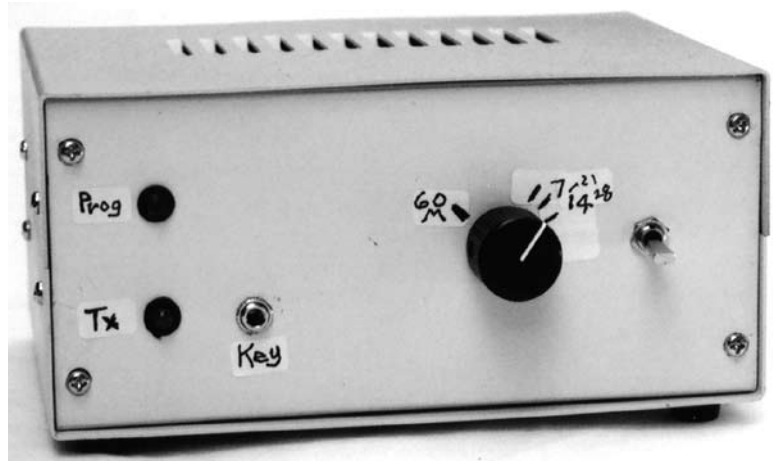
James Ahlstrom, N2ADR

221 Pleasant Plains Rd, Stirling, NJ 07980; jim@interet.com

An All-Digital SSB Exciter for HF

Build your own SSB exciter to go with your software defined radio receiver by using an FPGA and software.



These are exciting times to be involved with radio. The digital revolution that has transformed music and photography, and that will soon transform television, is finally making its way to the HF bands. Each year brings new and better digital integrated circuits. Now more and more of radio "construction" can happen in software rather than in hardware. Instead of drilling a panel and mounting knobs and switches, the radio "panel" can be a computer display. Tuning, modulation, demodulation and most filtering can be software too. As the hardware part shrinks, it also becomes more generic, and major modifications can be done in software without altering the hardware at all. The software can be shared using the Internet. I think that software defined radio (SDR) and digital signal processing (DSP) techniques provide the easiest way to homebrew your own equipment, even considering the need to work with surface mounted parts. Home construction can again be a viable and enjoyable part of ham radio.

In 2004 I started building a software defined radio receiver. I was inspired by the work of Leif Åsbrink, SM5BSZ; Gerald Youngblood, AC5OG; Pieter-Tjerk de Boer, PA3FWM, and many others.[1, 2, 3, 4] I had been licensed as a teenager, and I built some of my own equipment, as was common at the time. My license lapsed while I was raising my family, however. I always enjoyed electronics projects, and when I got my ticket back, I wanted to homebrew my own station. Of course electronics had changed substantially and there was much to learn. By 2006 I had a working CW station that offered full break-in (QSK) operation. The receiver hardware was based on Gerald Youngblood's design, but the software was my own. I named the software QUISK, a more pronounceable version of QSK. The CW exciter was the receiver VFO, and used the Analog Devices AD9953. Then it was time to design an SSB exciter.

The project described in this article is my current (October 2007) SSB and CW exciter. I have been operating it on 40 meter CW and phone. Signal reports have been favorable, with good audio and no reports of clicks or noise.

This design is completely independent of my receiver and old CW exciter. In the past, I would have tried to share components between the transmitter and receiver to reduce cost and complexity, but as pointed out by Wes Hayward and others, there are advantages to a design in which the receiver and transmitter are independent.[5] This is especially true for those of us without extensive test equipment, because the receiver can be used to listen to and test the transmitter. A block diagram of a digital transceiver will show few common components except for the VFO anyway, and there are no expensive crystal filters to share.

Although my receiver is based on Gerald Youngblood's design, I decided to move his transmit mixer to the digital domain and use a second independent VFO. That way I could continue to operate CW while building my new exciter, and I could test it with my existing receiver. So, my receiver and new exciter share nothing but the power supply.

The digital mixer and VFO are both provided by a field programmable gate array (FPGA). In case you don't "get" FPGAs, let's take a few minutes for a brief introduction.

What is an FPGA?

The heart of this exciter is the FPGA, and learning to work with it was one of the most fun parts of this project.[6] I used to think that an FPGA was like one of the old PAL chips; a way to replace a few gates and maybe a shift register with one chip and an embedded design. Progress in FPGA technology has been just crazy, though, and the FPGA has evolved into a fast general-purpose computing engine; one more than capable of digital signal processing tasks. [7]

The FPGA consists of a few thousand "cells" that can function as four-input, one-output gates. The logic function for each gate is programmed into a lookup table. For each input, the table determines the output. So the cell can be an AND gate, an OR gate, or any other sort of gate. The cells can also be programmed as adders, or as registers of any width. Complicated digital circuits can be built up as combinations of these cells. The FPGA has a couple dozen dedicated hardware multipliers to speed up multiplication,

[1]Notes appear on page 10.

and it has several kilobytes of memory. The FPGA has busses and connection resources to connect everything, and these act like programmable wires. The FPGA "program" is a list of what cells to use and how to connect them. The FPGA has a hundred input-output (IO) pins, and the IO connections are programmable too. The FPGA is fast. It can operate at a clock rate of 150 MHz, which is much faster than a microcontroller. Of course these numbers are rough. FPGAs come in sizes with greater or lesser amounts of resources.

Programming an FPGA is different from programming a microcontroller or a personal computer because the program does not describe a sequence of instructions for a CPU; it describes the wiring of a digital design. Standard sequential languages like *C* or *Python* are not appropriate for this, and generally you would write an FPGA program in the *Verilog* or *VHDL* languages. The FPGA program for this project was written in the *Verilog* language. If you are used to programming computers, keep in mind that an FPGA is a parallel device. An FPGA does not execute its program a line at a time. It executes multiple lines, and maybe even all lines, simultaneously. With a computer it is difficult to get several things to happen at once, but with an FPGA it is hard to get them to happen sequentially. The ability to write several independent code modules, compile them into different parts of the same FPGA, and then execute the modules in parallel gives the FPGA great computational power. This is what CPU manufacturers are doing with the new multi-core processors, but with an FPGA you get as many special purpose cores as you care to program. FPGAs lend themselves to pipelining. That means you can convert a 50-line program into 50 one-line programs that run in parallel. Once the pipeline fills up, you get one output per clock tick.

FPGAs provide great computational power, but programming them takes a while to get used to. *Verilog* bears a superficial resemblance to the *C* language, and that caused me some trouble because *Verilog* does not act like *C*. It helps to think in terms of digital design elements like adders, multiplexers and registers because that is what the FPGA program represents, and the *Verilog* compiler writers were digital designers who think in those terms.

In the computer business it sometimes happens that a language or program is pressed into service in an area it was not designed for. For example, *Java* started out as a language to program set top boxes. *Verilog* started out as a language to model digital circuits. A standard *Verilog* reference book is full of language features that are illegal in an FPGA program, such as the ability to enter gate delays. To program an FPGA, we are using a subset of the *Verilog* language in a special-purpose way, and this is called "*Verilog* for synthesis." Maybe someday there will be a good book on this, but I haven't seen one yet. Meanwhile, you will have to rely on the documentation provided by the FPGA manufacturer.

Of course, to duplicate my SSB exciter you don't need to program the FPGA. The object code is provided. If you do want to program it, the source code is also provided. The program is a modest one, and consists of only a few hundred lines of *Verilog* code.

Apparently, an FPGA is a very advanced piece of silicon. But it has (at least) two features that make it a very friendly component for the amateur homebrewer. Typically, we are limited to one or two sided printed circuit

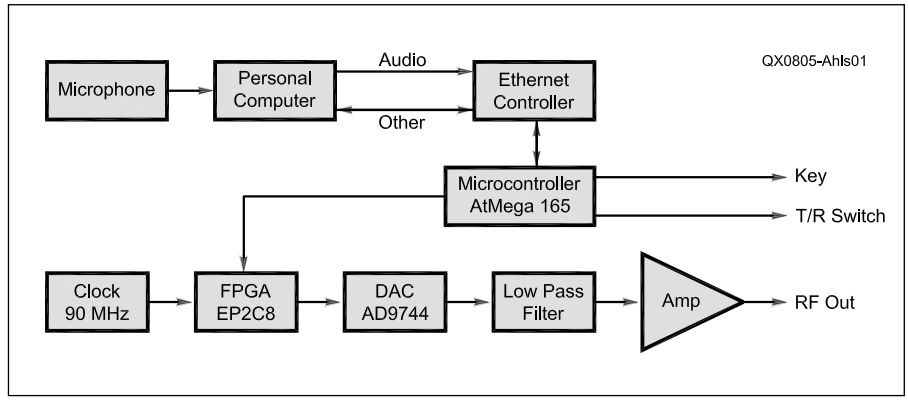


Figure 1 — This is a block diagram of the exciter.

PC Software

Microphone samples at 48 kHz, 16-bit samples
SSB speech processing
- High pass IIR filter to shape passband
- Bandpass filter (FIR) and separate into I and Q
- Tune up to 12 kHz
- Clip audio 12 dB
- Bandpass IIR filter to reestablish bandwidth
- Tune back down to baseband
- Normalize amplitude (AGC)

Send 16-bit I and Q audio samples using UDP over Ethernet
Send FPGA program and FPGA transmit frequency
Receive key status to mute audio and substitute sidetone

Microcontroller Software

Fixed IP address 192.168.1.44
Known UDP ports 0x553A, B, C
Process all Ethernet traffic
- Ethernet Address Resolution Protocol (ARP)
- ICMP echo request (Ping)
- FPGA program: SSB, CW, etc.
- Audio data — copy to FPGA
- Frequency control — send to FPGA

Send key up/down status back to PC

FPGA Software

Write audio samples to memory
Memory holds 4608 samples, 96 milliseconds
When memory is half full, turn on transmitter
- Read audio from memory; write to memory continues
- Interpolate to 90 MHz
- Tune to output frequency with CORDIC
- Send RF data and clock to 14-bit DAC

Audio will eventually stop
When memory is empty, turn off transmitter

Figure 2 — The exciter software outline.

boards. These are the only boards that can be made at home, and even then only one side can be accurately etched. Multilayer boards are very expensive in small quantities, and the cheapest two sided commercial prototype board I have found costs about $70, as much as all the advanced ICs combined. Maybe some day I will understand why the box and the circuit board cost more than the contents. Anyway, some of the connections on a homebrew board may be longer than desired or may be made with an actual wire instead of a board trace. These lines may "ring"; that is, you may have multiple pulse reflections on them. With a dedicated chip you are out of luck, but with an FPGA you can program around the problem. For example, you can count up a counter when a line goes high, and count down when it goes low. You don't accept the value of the line until some counter value is reached; that is, after a known delay. Essentially you have de-bounced your faulty line, just as you might de-bounce a mechanical switch.

The other homebrew advantage of an FPGA is the programmable IO pins. Suppose you need to connect an eight-bit port to another chip. With a dedicated chip it might happen that the port pins go up on one chip and down on the other, so if you connect them with eight parallel traces on the board, the port bits will be reversed. This never happens with an FPGA. You simply connect all the traces to any convenient pin on the FPGA, and program the pins as desired.

Exciter Block Diagram and Schematic Diagram

The block diagram of the exciter hardware is shown in Figure 1. Figure 2 is an outline of the software. Figure 3 shows the exciter schematic diagram. In this discussion, I will be giving some background on the design decisions and why they were made, in case you want to use this project as a starting point for your own design.

We start with the microphone. I am using a Logitech USB microphone from RadioShack, catalog number 33-101. This plugs into the computer and appears as another sound card. It can capture audio data but not play back. The mic captures 16-bit samples at 48 kHz, a standard rate for most sound cards. This is greater than CD quality, but the CIC filters (see below) need this high rate. An alternative is a sound card with a mic input, but the mic inputs of some sound cards are noisy, and you may need an external preamp or volume control to adjust the level. My receiver uses my only sound card, so I am using the USB mic. The computer filters and clips the mic audio and divides it into in-phase (I) and quadrature (Q) signals (I/Q signals). Other designs would send the I/Q signals to sound card left and right channels, and then to an analog I/Q mixer to generate RF. My design uses exactly the same I/Q signals, but they remain digital data.

The software I use to generate these I/Q signals is part of my QUISK receiver, and is programmed in *Python* using a Tkinter graphical user interface (GUI), and also in *C*. You are welcome to use QUISK if you can run *Linux*. If you want to run *Windows*, the GUI is portable and the *C* is quite generic, so only the sound card access *C* code would need to be changed. You could also cut the relevant code (microphone.c) out of QUISK and use it in your own software. The I/Q signals required are the usual ones generated in other software, such as that used with the SDR-1000. Many software packages can generate them.

We now need a way to get the digital audio data out of the computer and into the exciter hardware. The data is two 16-bit samples (I and Q) at a 48 kHz rate, a bit rate of 1.5 megabits per second. Lots of people use USB, and I see that the new FlexRadio 5000 uses firewire. Although I like USB when someone else does the programming, USB is difficult to program even with a special controller IC. On the exciter side, the initial handshaking between the exciter and the host computer is complicated. On the computer side, the problem with USB is the drivers. I use both *Linux* and *Windows*, and they require different USB drivers. It always seems that the Linux drivers are missing or are an afterthought. Once you have the drivers, the problem is sharing the device between two or more programs on the computer. Only one program can use a USB interface at once unless you write the multiplexing (sharing) code yourself.

In this design I decided to use Ethernet, or, more specifically, UDP packets within the Internet Protocol (IP) carried by an Ethernet local area network. This is what your computer is no doubt using now for Internet connectivity, and you may know it as TCP/IP. But we are using UDP (another IP data type) instead of TCP. To connect the exciter and the computer, you just plug their Ethernet connectors into the same hub or switch. Figure 4 shows the back panel of my exciter, with the various connectors, including the Ethernet connector.

The programming is very simple. On the exciter side there is no startup code for UDP. Data packets just arrive at the Ethernet controller, and the microcontroller processes them. On the computer side, the code for UDP uses the "sockets" interface and the same *C* code can be used for *Windows* and *Linux*. UDP socket code can be written in many different computer languages, not just C. I have UDP code in C and in Python. Any code written in *Python* automatically runs on *Linux* and *Windows*, including UDP code. Unlike USB, multiple programs can talk to the interface at once, just by using different UDP ports. UDP can also be routed over the Internet. In case anyone thinks UDP is not suitable for audio data, remember that it is the protocol used by voice over Internet protocol (VOIP) providers such as Vonage and Skype.

We are using the Ethernet connection for other data besides the audio data. Since the exciter is on an IP network, it answers ping requests. Pinging the exciter is the first debug step to see if it is alive. The exciter also participates in the address resolution protocol (ARP), a way for the computer to discover its Ethernet address given its IP address. If you don't know what ARP is, don't worry; it just means the exciter is working politely with your local area network. The computer uses Ethernet to send the frequency tuning data to the FPGA, and to request a status reply. The exciter uses Ethernet to send the key up or down status to the computer, to mute the receiver during transmit. Note that the key is connected to the microcontroller. In CW, key up/down messages also substitute a sidetone.

Ethernet is also used to program the FPGA. Since the FPGA does not have permanent memory, the FPGA must be programmed on every power up. A program for an FPGA is a several hundred kilobyte file. I have three FPGA programs available on my computer; one for SSB, another for CW and a third for a two-tone test. The chosen program is downloaded from the computer to the microcontroller using UDP, and the microcontroller programs the FPGA. An alternative design would be to store the program in a special memory chip connected to the FPGA. My design avoids the small cost of the memory chip, however, and the several hundred dollar cost of the special memory programming cable. There are also other advantages. You can debug the microcontroller and Ethernet code first without having to deal with the FPGA. It makes it easy to write new FPGA programs while still using the old ones. I can imagine programs for PSK31, FM and other modes being added in the future.

The microcontroller now copies the audio samples to the FPGA. But before we discuss that, let's take a look at the FPGA clock. If you are used to microcontrollers it will be no surprise that an FPGA has a clock, too, and that operations occur on clock edges. In fact, FPGAs have special dedicated clock inputs, and you must use these to keep clock skew to a minimum. In our case, the FPGA clock is also used for the digital to analog converter (DAC), and that means you need a really good clock. Any noise on the DAC clock

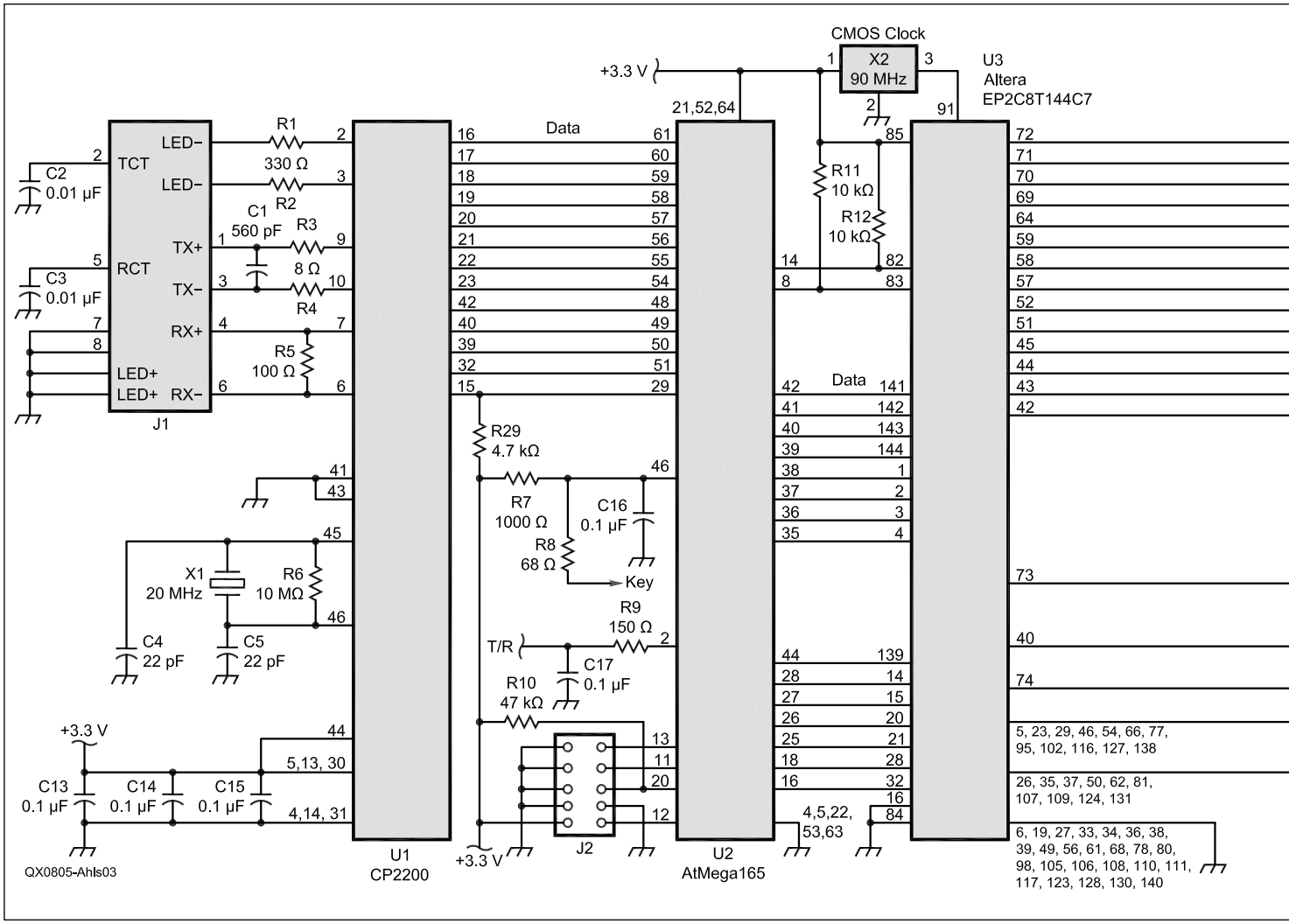


Figure 3 — This drawing shows the exciter schematic diagram.

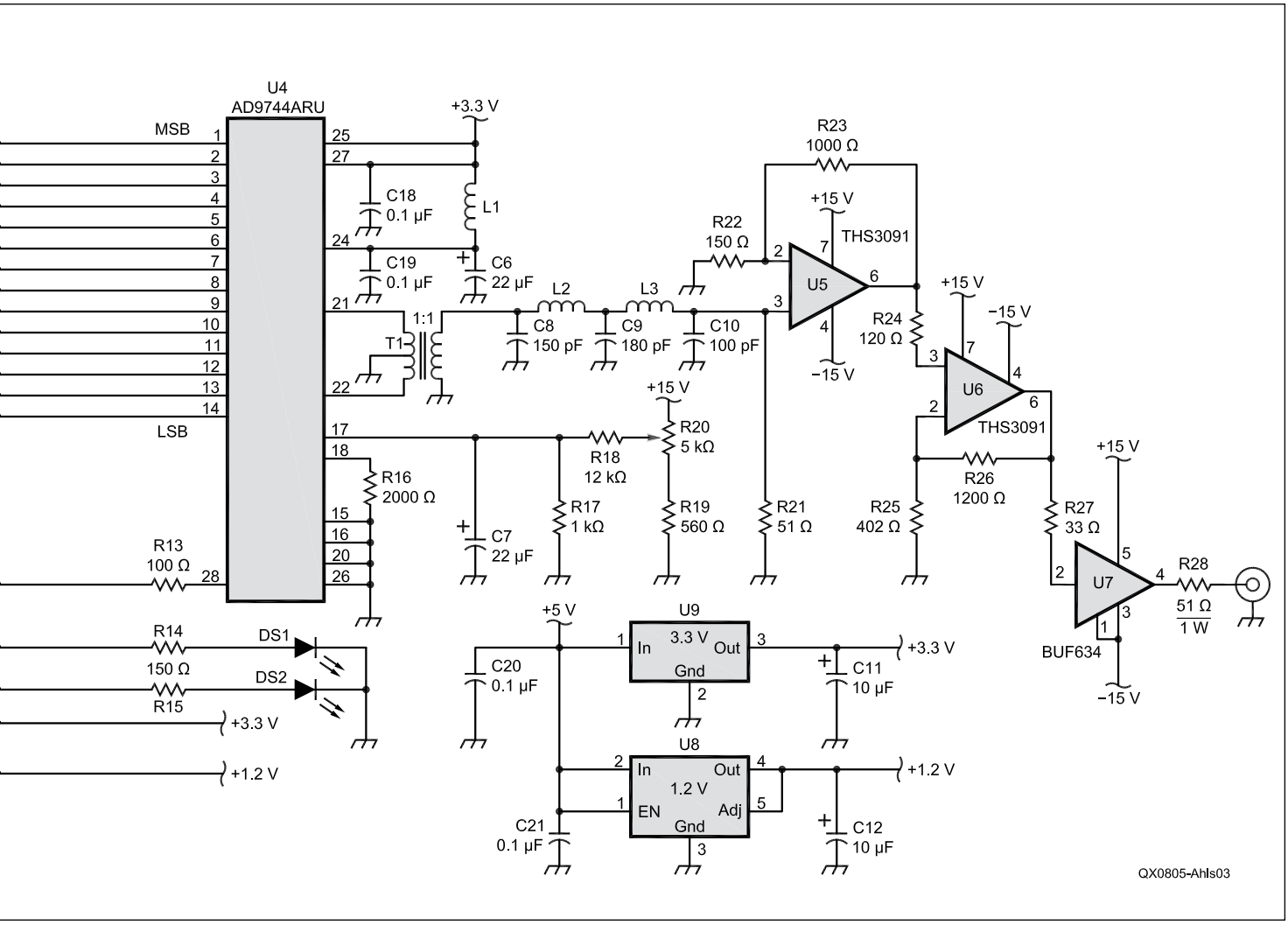
Parts List

All Capacitors are 50 V ceramic 1206 SMD unless otherwise indicated.

C1	560 pF
C2, C3	0.01 µF
C4, C5	22 pF
C6, C7	22 µF Tantalum 10 V SMD
C8	150 pF
C9	180 pF
C10	100 pF
C11, C12	10 µF Tantalum 6 V radial
C13-C21	0.1 µF 50 V ceramic 0805 SMD bypass. Not all bypass capacitors are shown.
DS1	LED, Green
DS2	LED, Red
J1	Mag Jack, Stuart SI-50170, DigiKey 380-1086-ND
J2	5 × 2, 0.1 inch pin header
L1	7 turns no. 32 enameled wire on ferrite bead
L2	359 nH, 10 turns no. 24 enameled wire on T37-6 core
L3	327 nH, 10 turns no. 24 enameled wire on T37-6 core

All Resistors are ¼ W, 5%, 1206 SMD unless otherwise indicated.

R1, R2	330 Ω
R3, R4	8 Ω
R5, R13	100 Ω
R6	10 MΩ
R7, R17, R23	1000 Ω axial
R8	68 Ω axial
R9, R14, R15	150 Ω axial
R10	47 kΩ
R11, R12	10 kΩ
R16	2000 Ω
R18	12 kΩ
R19	560 Ω axial
R20	5 k Ω pot, DigiKey CT2204-ND
R21	51 Ω
R22	150 Ω
R24	120 Ω axial
R25	402 Ω
R26	1200 Ω axial
R27	33 Ω
R28	51 Ω, 1 W axial
R29	4700 Ω
T1	6 bifilar turns no. 32 enameled wire on BN-43-2402 core
U1	CP2200
U2	AtMega165, DigiKey AtMega165-16AV-ND
U3	Altera EP2C8T144C7, DigiKey 544-1457-ND
U4	Analog Devices AD9744ARU, DigiKey AD9744AR-ND
U5, U6	THS3091DDA, DigiKey 296-15789-5-ND
U7	BUF634, DigiKey BUF634T-ND
U8	MIC29302WT adjustable voltage regulator, DigiKey 576-1124-ND
U9	MIC29300-3.3WT 3.3 V voltage regulator, DigiKey 576-1119-ND
X1	20 MHz crystal, DigiKey 300-8446-ND
X2	CMOS Clock, 90 MHz, Fox F4105



will appear at the output, and any clock jitter will appear as phase noise. You could build your own crystal clock, and if you do I suggest using the differential clock inputs on the FPGA. If you use a CMOS clock oscillator, as I do, avoid those meant for general use, and find one specifically designed for communications use — one with a specified jitter. For DAC use, the clock is a big deal.

The I/Q audio samples at 48 kHz are read from the Ethernet chip by the microcontroller, and then copied to the FPGA. The job of the FPGA is to change the sample rate to 90 MHz ("interpolate") and tune the signal to the desired output frequency in a digital mixer. An alternative design would use an ASIC for this, one known as a "digital up converter" (DUC). There are some problems with a DUC, however. Some use a ball-grid package, and I haven't figured out how to solder those yet. The Analog Devices AD6622 has a serial interface, and it is hard to move the data through the microcontroller fast enough. Besides, FPGAs are fun. See Note 6.

The interpolation (sample rate increase) step is simply a matter of adding zero samples to the signal. We have a 48 kHz rate that must be multiplied to 90 MHz, an increase of exactly 1875 times. So we just take one sample, add 1874 zero samples, and repeat. Actually it is better to do this in two stages, first multiplying by 25 then 75. But there is a catch. There will be repeats, or "aliases" of the original signal every 48 kHz, and these aliases must be filtered out by an anti-alias filter. The well-known way to do this is to use a cascaded integrator-comb (CIC) filter.[8] The CIC filter is especially fast and simple, uses no multiplications, and it interpolates and filters all at once. Note that we are using a narrow 25 times interpolation stage followed by a wider 75 times stage. This is because a CIC filter can only work with a signal that is a small fraction of the initial sample rate. In our case that is an SSB signal in 0 to 3 kHz divided by 48 kHz. Very wide (30%) bandwidths will not be filtered effectively. Also note that interpolation is an integer multiplication, and you wind up with a multiple of the original sample rate of 48 kHz. If you want to raise the clock frequency to a higher standard value, the next available clock rates are 120 MHz (times 2500) and 150 MHz (times 3125).

After interpolation we have a 90 MHz sample rate, but the data is still audio. The last job of the FPGA is to mix the baseband SSB signal to the output frequency in a digital mixer. There is no magic to a digital mixer. It just relies on the rule that you multiply two exponentials by adding their exponents. For example:

$10^2 \times 10^5 = 10^7$

Next, recall that if you put a sine or cosine wave into a Fast Fourier Transform (FFT) you get two spikes, one at plus frequency and one at minus frequency. But if you use a pure wave, the complex exponential

$e^{i\omega_0 t}$

you get only one spike. That is the reason to use complex signals; your mixer only produces one output, not the sum and difference

a regular mixer does. A digital mixer is just a multiplier. In our case, the first exponential is the I/Q SSB signal:

$e^{i\omega t}$

the second is a pure wave

$e^{i\omega_0 t}$

and we multiply these together to get the product

$e^{i\omega t} \times e^{i\omega_0 t} = e^{i(\omega+\omega_0)t}$

We see that the product is the original signal plus the tuning frequency, ω_0, just what we want from a mixer. The only problem is how to generate the tuning signal — a complex exponential — at, for example, 7.25 MHz, that will tune our zero hertz SSB signal to the 40 m band. This can be generated by the CORDIC algorithm.[9] This algorithm calculates a complex exponential by successive rotations of a vector, to produce a known phase angle. The phase angle is increased at each clock pulse by a constant amount, the "phase increment," and that increment determines the output frequency. The phase increment is sent from the computer to the microcontroller, and then to the FPGA.

After the digital mixer, the FPGA has a 7.25 MHz (for example) signal sampled at 90 MHz. It sends this signal to a 14 bit digital-to-analog converter (DAC), an Analog Devices AD9744. From now on the signal is analog. The next step is an analog low pass filter to remove high frequency images. This is only a five pole low pass filter, but additional filtering is provided later. The signal is then amplified by a THS3091 current feedback operational amplifier (op-amp). Originally this was the final output stage, but this chip overheated when driving a 50 Ω load. I wanted as high an output as I could get from the op-amps. The op-amps have very low IMD products, and the more output I could get from op-amps, the less distortion I would have from subsequent amplifier stages. So I added a second THS3091 and a BUF634 output stage. This combination can drive 50 Ω at about 1 W, but since I have filters following the exciter, I added a 50 Ω output series resistor. The output is 300 mW, up to about 25 MHz, and decreasing to 175 mW at 30 MHz. This decrease in output should not occur, but I have not had time to track it down.

I measured the IMD performance on my SDR-IQ, and got –62 dB at 5.3 MHz, –50 dB at 21.2 MHz, and –41 dB at 29 MHz.[10] These measurements are in dB below one tone, not PEP. The SDR-IQ does not have an IMD specification, so I am not sure where the IMD is generated. The IMD is not a strong function of output level until the output clips. The IMD performance is very good indeed.

Operation on CW

The above description is applicable to SSB. For CW, the hardware is identical, but the FPGA runs the CW program instead of the SSB program. Note that the key connects directly to the microcontroller, so generating CW does not use a computer. The FPGA uses the same CORDIC algorithm, but the input is no longer audio samples, it is just a number. A constant CORDIC input of 310000 would produce a sine wave at full amplitude, but we need to shape the CW envelope to reduce key clicks. When the key goes down, a counter slowly counts up to 310000 and stays there. When the key is released, it counts down to zero. The counter value is the input to the CORDIC algorithm, and the result is a ramp up and down of the CW envelope with a time of seven milliseconds. The T/R relays are also controlled by

Figure 4 — This photo shows the back panel of the exciter, with the various input/output connectors.

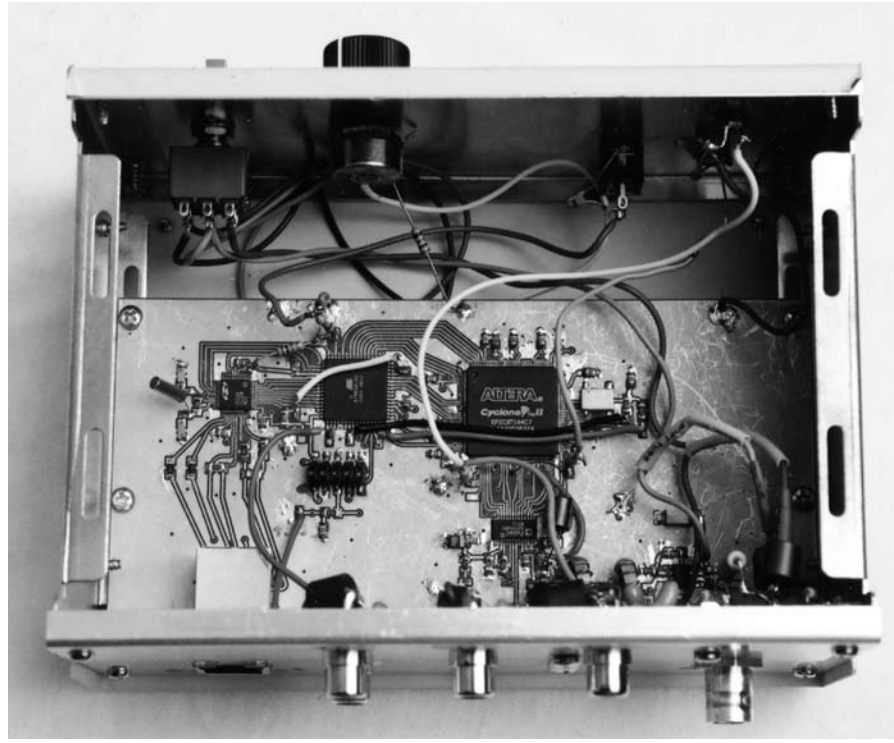


Figure 5 — Here is a look inside the project box. Note the construction style, which is a mix of circuit traces and "ugly" construction.

the microcontroller. On key-down, the T/R relays are closed and the FPGA is keyed 12 milliseconds later so the relays have time to switch. On key-up, the relays are opened after a 12 millisecond delay to allow time for the CW waveform to go to zero.

Notes on Construction

I generally build all my projects using the "ugly" construction method. The ground leads of the parts are soldered directly to an un-etched circuit board, and the remaining connections are made directly above. The un-etched board provides a good ground plane. I like this method because it is easy to experiment with the circuit, and to replace sections of the circuit that do not work. But when there are surface mounted parts involved, the board must be etched to accommodate them. We still need a good ground plane, so only a minimum of the board can be etched. I etch the pads for the surface mount components, the power supply bypass chip capacitors, most direct connections between integrated circuits, and some of the power traces. There is a limit to how many traces can be routed on the board because there must be a very good solid ground plane in these RF and digital circuits. Even routing just power traces can break up the ground plane too much, so I make many connections with wires. It helps to use double-sided board and solder the top and bottom surfaces together with wires passed through small holes. The bottom side remains un-etched. As space permits, I attach unused IC pins to pads so wires can be added later if required. The result is still "ugly" construction but with surface mounted parts incorporated.

The board for this project was laid out using *Eagle* under the assumption that I would make the board myself.[11] The board turned out with too many bridged traces. The 0.5 mm pin pitch parts are at the limit of what I can manage with a homemade board. Perhaps I can make it work by making the traces narrower, as the pre-sensitized circuit boards I am using have trouble with narrow spaces between traces. In the end, I decided to order a commercial two-sided prototype circuit board. These are quite workable but come without solder mask or silkscreen. Figure 5 is a photo of the circuit board inside the project box.

Digital Spurs

No transmitter based on DSP is complete without a search for spurious emissions, or "spurs." All mixers generate unwanted outputs at frequencies removed from the desired output. In the case of DSP circuits, a desired sine wave output will be accompanied by small unwanted signals that bear no obvious relationship to the carrier. Part 97.307 of the FCC rules specifies that the maximum spur for transmitters below 30 MHz must be at least 43 dB below the desired output, and that the spurs must be reduced "to the greatest extent practicable." Taking a look at the data sheet for the AD9744, we see that it has spurs of –74 dBc at an output frequency of 25 MHz and a sample rate of 65 MHz. Spurs generally rise with the output frequency and decrease with the sample rate.

The FPGA digital mixer can also produce spurs due to the inaccuracy of integer arithmetic. In this design we are using wide widths for the calculations; 32 bits for phase and 20 bits for amplitude. If spurs are found to result from the FPGA it can be programmed for even wider data widths.

Digital spurs can be located far removed from the output frequency. For example, the AD9953 DDS VFO I am using for my receiver shows –60 to –70 dBc spurs at 20, 160 and 180 MHz when operating at 80 MHz. Some data sheets will specify close-in spur performance that is significantly better than total bandwidth performance. We are using a narrow bandwidth filter for each band on transmit to suppress these wide spurs.

I searched for spurs from the exciter using my receiver, but the largest spurs I could find clearly came from the receiver and not the exciter. So I recently purchased an SDR-IQ receiver and spectrum analyzer from RfSpace to search more carefully for spurs. (See Note 10.) The specified spur-free range is 80 dB below full scale worst case, and 100 dB typical. I set my two-tone test signal level to 20 dB below full scale and searched several megahertz around the signal frequency. The largest spurs I could find from the exciter were 77 dB below one tone on the 10 m band. Since this is 97 dB below full scale, I cannot be sure the spurs are due to the exciter and not to the SDR-IQ. The SDR-IQ shows only 190 kHz of the band at once, so it takes a while to set the exciter to several frequencies in each band and search many megahertz around each frequency. Even then, I didn't try every possible operating frequency. Still, the purity of the output was gratifying. It was difficult to find any spurs at all, even small ones.

The Software

The software for this project is available on my Web site, and is licensed under the general public license (GPL) as an open source project.[12,13] Complete project files, not just source files, are provided. The FPGA software is in the Altera directory, and the microcontroller software is in the AVR directory. This software should not require changes. The computer software is in the quisk directory, but this includes the complete receiver software too. The QUISK software runs under the *Linux* operating system, but only the quisk/microphone.c file is relevant to the exciter, and it is generic *C* code. It should run on *Windows* by just changing the sound card access.

The software for this project was written in *Python*, *C*, *Verilog* and *AVR* assembler language. I especially enjoy *AVR* assembler language. It is a textbook example of a RISC processor instruction set, and is very different from the high level languages I use in my day job. This was my first *Verilog* project, so I hope to get suggestions for improvement.

Future Work

This is version one of the exciter, and there are some things I would do differently the next time. Copying audio data through the microcontroller is a bottleneck. The current design works, and does have the needed throughput, but with little room to spare. If a faster data rate is required in the future, the microcontroller could not keep up. And it would never be able to operate at the full 10 megabit rate of Ethernet. The Ethernet controller CP2200 used here has an 8 bit interface that is perfect for the microcontroller. I planned to use the RXAUTORD interface in the CP2200 to read successive bytes, but I could not make that work. Instead I am setting an address for every byte read, and this has slowed down the throughput. Next time I would try connecting the Ethernet controller to both the microcontroller and the FPGA. The microcontroller could get the FPGA program and handle routine UDP packets as in the current design, but the FPGA would read its audio data directly from the Ethernet controller. This should provide close to the maximum 10 megabit data rate. It would probably be easier to do this with a different Ethernet chip such as the ENC28J60.

The drop in output at 30 MHz should not happen. I am sure this could be fixed. I would also try to use one THS3091 instead of two. Using two was just the result of adding an extra buffer after the board was done. It would also be interesting to raise the clock rate to 120 or 150 MHz instead of 90, although 90 MHz is clearly sufficient for 30 MHz output.

I need to continue to search for spurs. Perhaps I will write a computer program for the SRD-IQ to search over its frequency range and automatically find spurs. Maybe I should port QUISK to *Windows*, but first I need some time to just operate my station!

The Rest of My Station

The exciter output goes to a BNC connector and then to a separate filter box, ampli-

fiers, the transmit/receive (T/R) switch and the antenna. The filter box is in a separate enclosure and has a band-pass filter for each band, to provide spur reduction on transmit and protection from strong out of band signals on receive. After that the signal is amplified and sent to a set of low pass filters, also one per band. The signal then goes through a final 35 MHz low pass filter and then to the antenna. The final power output is about 100 W. Transmit/receive (T/R) switching is provided by relays driven by the microcontroller, and the microcontroller provides suitable sequencing. My filter box currently has filters for the 60, 40, 30, 20 and 17 m bands with one remaining position. This meets my current requirements, as I lack antennas for 160 and 80 m, and my receiver is limited to 20 MHz and below. Of course, the SSB exciter will work on all HF bands.

Conclusions

Operating this exciter on the air has been gratifying. Audio reports have been good, and if I can hear a station I can usually talk to them, too. CW operation is especially nice with smooth full break-in operation.

It seems to me that the digital techniques used in a software receiver and a software transmitter are so different that there is no point in trying to share parts between them. In a receiver the problem is dynamic range and avoiding clipping. In a transmitter the problem is to maintain high signal levels to maximize power output and numerical accuracy. My old CW exciter that shared the VFO with the receiver never did have a very good keying envelope, and the switch in VFO frequency between transmit and receive made full break-in harder to achieve. This current design is much better.

I hope this project inspires others to build their own equipment. You are welcome to duplicate this project, or, better yet, improve on it. I know some people dread working with surface mounted parts, but this is a solvable problem with proper magnification and a decent soldering iron. And there is no alternative if you want to use modern ICs. These modern ICs pack a punch, and they eliminate the need for endless lists of discrete parts. Just look at this board. There are only five integrated circuits, and most of the rest of the parts are power supply bypass capacitors. To me, this is easier to construct than the hundreds of discrete parts that constitute the alternative.

I would like to say a special word of thanks to Leif Åsbrink, SM5BSZ. His Linrad pages are a treasure trove of practical information and hard-won insight. (See Note 1.) Near the end of this project, when I was stuck on how to do SSB speech processing, I revisited his Web site. And there it was: his recently added insights on just that topic. Thanks Leif!

James Ahlstrom, N2ADR, was first licensed as KN3MXU in 1960. He received a BS in physics from Villanova University in 1967 and a PhD in physics from Cornell University in 1972. He then moved to New York to work in the financial business. He is currently a one-third owner of Interet Corporation, Millburn, New Jersey. Interet publishes software to analyze leveraged equipment leases. His license lapsed while raising his family, and he was re-licensed in 2006. He currently holds an Amateur Extra class license. Besides Amateur Radio, he enjoys bird watching, skiing, music and working out at the gym.

Notes

[1]**www.nitehawk.com/sm5bsz/**

[2]Gerald Youngblood, AC5OG, "A Software-Defined Radio for the Masses," *QEX* Jul/Aug 2002, pp 13-21, Sept/Oct 2002, pp 10-18, Nov/Dec 2002, pp 27-36, Mar/Apr 2003, pp 20-31.

[3]**wwwhome.cs.utwente.nl/~ptdeboer/ham/sdr**. [Note that this URL is correct as printed. There is no period between www and home. — Ed.]

[4]**www.arrl.org/tis/info/sdr.html**

[5]Wes Hayward, Rick Campbell and Bob Larkin, *Experimental Methods in RF Design*, ARRL, 2003, ISBN: 0-87259-879-9; ARRL Publication Order No. 8799. ARRL publications are available from your local ARRL dealer, or from the ARRL Bookstore. Telephone toll-free in the US 888-277-5289, or call 860-594-0355, fax 860-594-0303; **www.arrl.org/shop; pubsales@arrl.org**.

[6]**www.fpga4fun.com**

[7]U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, Second Edition, Springer-Verlag, 2004.

[8]Richard G. Lyons, *Understanding Digital Signal Processing*, Second Edition, Prentice Hall, 1996.

[9]Ray Andraka, A Survey of CORDIC Algorithms for FPGA Based Computers, **www.andraka.com/files/crdcsrvy.pdf**.

[10]**www.rfspace.com**.

[11]**www.cadsoftusa.com**.

[12]**www.james.ahlstrom.name/quisk**.

[13]The software files for this project are also available for download on the ARRL Web site for our readers' convenience. Go to **www.arrl.org/qexfiles** and look for the file **5x08_Ahlstrom.zip**. The authors Web site may have updated listings available for download.

QEX