James Ahlstrom, N2ADR

221 Pleasant Plains Rd, Stirling, NJ 07980; jahlstr@gmail.com

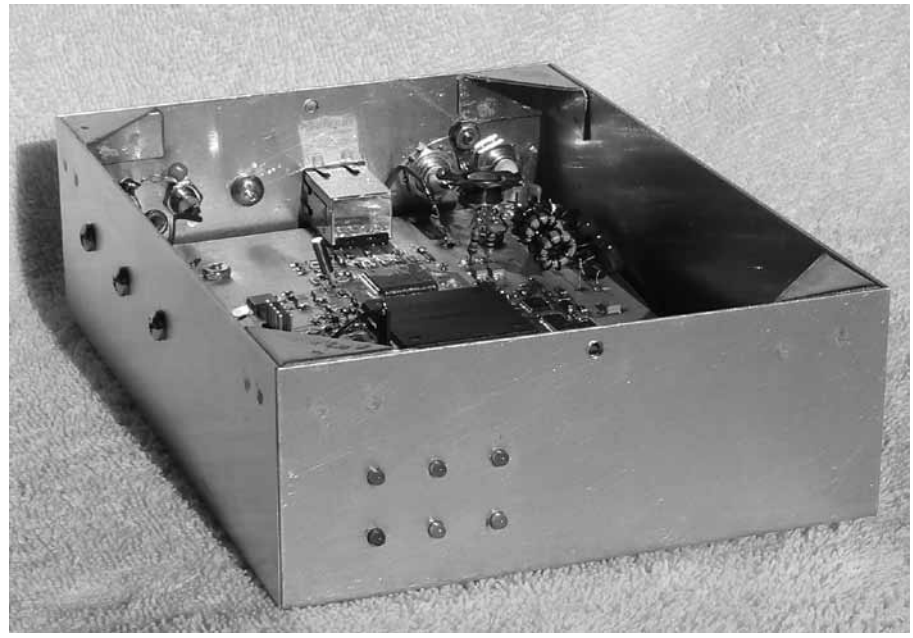
An All-Digital Transceiver for HF

Build your own HF transceiver using an FPGA and software.

In a previous article, I described a digital SSB exciter that used a personal computer (PC), a field programmable gate array (FPGA), and software.[1] That design used an Ethernet controller connected to a microcontroller (MCU) and then connected to the FPGA. The microcontroller ran the Ethernet and IP code, leaving the FPGA free to do the digital signal processing. The microcontroller was a bottleneck that slowed down the data speed to barely acceptable levels. I wanted to work at higher sample rates and clock speeds, so I started this new project.

At first, I tried to connect the Ethernet controller chip to both the MCU and the FPGA, but this proved awkward. In the end, I decided to eliminate the MCU and connect the Ethernet controller directly to the FPGA. This resulted in a high data rate but meant that I had to write new Ethernet and IP code for the FPGA and throw my MCU code away. This was worrisome, as I had never written a large FPGA program before. I purchased some new Verilog books that proved to be highly useful, thought about it for a while, and then dove in.[2,3]

With the MCU gone, I could change to a faster 10/100 megabit per second Ethernet controller. The FPGA would now be at the center of the design with the digital to analog converter (DAC) transmitter chip connected to it. Then I realized I could just add an analog to digital converter (ADC) for receive, and I would have a complete transceiver.



The Basics

For background, you might want to read my original article. For digital signal processing basics, be sure to see the book by Lyons[4] and the ARRL digital radio Web site[5]. But if that is not handy, here is a crash course in digital radio.

An ADC is an analog to digital converter; it converts your antenna voltage to a number. I am using a 14-bit ADC running at 122.88 MHz, and the numbers it measures range from –8192 to +8191. These numbers will be crunched by software to ultimately result in received audio. To transmit, you need a DAC: a digital to analog converter. You take microphone audio, convert it to numbers with a (different) ADC, crunch those numbers and send them to the DAC for direct conversion to RF. Then, amplify and filter the RF and send it to your antenna. The numbers we are talking about are actually pairs of numbers, an in-phase and a quadrature number, or an I/Q pair. These are most easily thought of as a complex number in the form x + jy. Or you could think of them as representing both the amplitude and phase in one number.

The software mostly consists of filters and mixers just like in an analog radio. A digital mixer is the multiplication by $\cos(2\pi ft) + j \sin(2\pi ft)$. You need to generate this I/Q number somehow, and the CORDIC algorithm is a good choice.[6] To tune numbers with a digital mixer you feed the I/Q pairs of numbers into a CORDIC module, and get new I/Q pairs out the other end. The new numbers represent the original signals shifted in frequency by a constant frequency; that is, they have been "mixed". The advantage that I/Q number pairs have over plain numbers is best seen in a digital mixer. Analog mixers always have images; they produce sum and difference frequencies. These digital mixers have no images. They produce either a sum or a difference frequency but not both.

The digital filters we are most interested in are low-pass FIR and CIC filters. The FIR (finite impulse response) filters can have very nice pass-band and stop-band responses, but they require hardware multipliers. The FPGA does have hardware

[1]Notes appear on page 00.

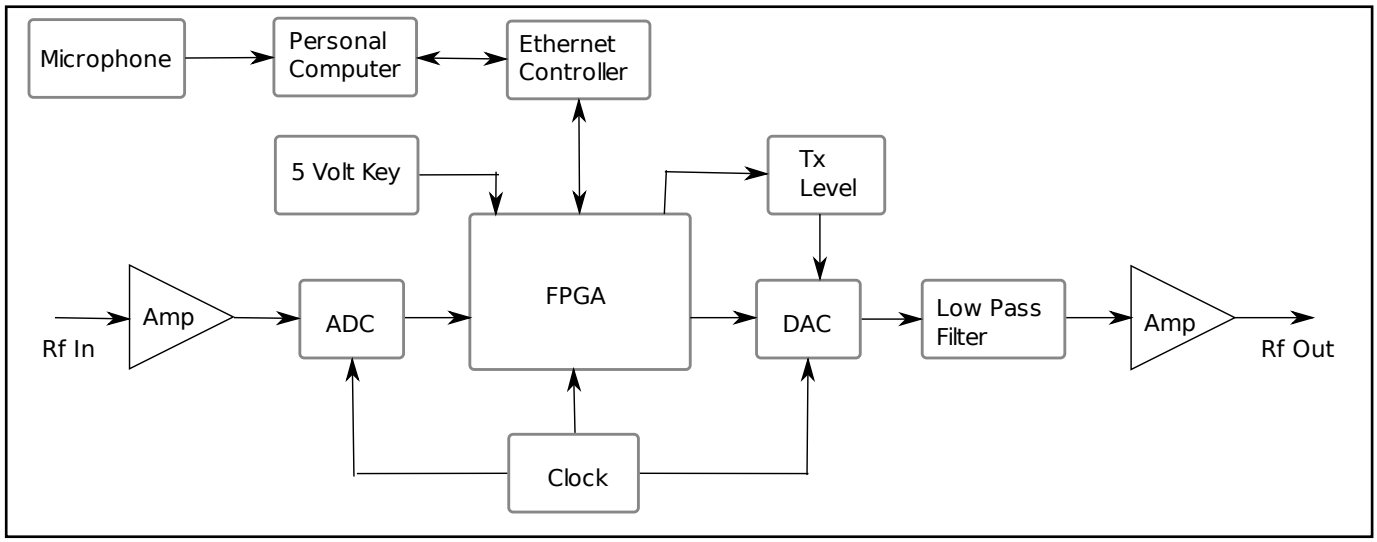


Figure 1 – A block diagram of the transceiver.

multipliers but not enough to make a very big filter. So we use some CIC filters, also. These CIC (cascaded integrator-comb) low-pass filters are only effective near zero hertz. As we move away from zero hertz to perhaps 1% or 10% of their bandwidth, the remainder of the stop-band has poor attenuation. But their advantage is that they require no multiplication at all, are very fast, and are easy to program in an FPGA. So a typical filter plan is to use a few CIC filters and then follow these by a single additional FIR filter to achieve the total filtering required.

Another important digital radio concept is sample rate reduction, or "decimation". Our ADC runs at 122.88 MHz and generates 14-bit samples, so its data rate is 1,720 megabits per second. We will not have much luck sending that through a 100 megabit per second Ethernet connection, so we will need to reduce the sample rate substantially. To do that, we first tune the original spectrum of signals to "baseband"; that is, toward zero hertz. Then, we low-pass filter the signal. We can then reduce the sample rate by eight times, for example, by keeping only every eighth sample. We use decimation on the receive path to reduce the sample rate. A reduction by 512 times gives 240,000 samples per second. On the transmit side we need to increase the sample rate ("interpolation") from 48 kHz to 122.88 MHz, an increase of 2,560 times. To increase the sample rate by eight times, we send a sample and then send seven zero samples. Both decimation and interpolation are usually done in stages with a rate change of two to maybe 40 instead of all at once. They both require low-pass digital filters at every rate change. I still find it amazing that the FPGA can handle all these calculations at the raw data rate of 1,720 megabits per second.

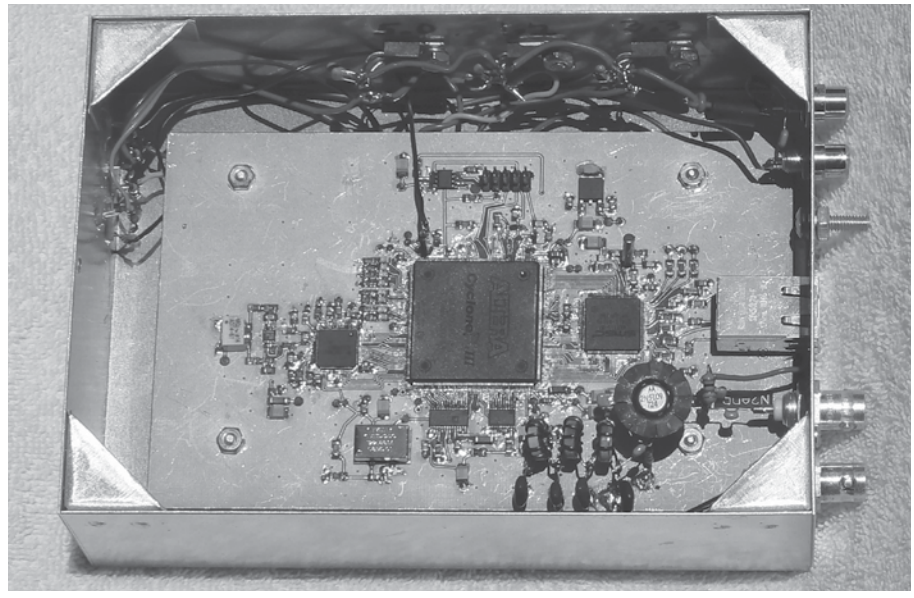


Figure 2 – The printed circuit board.

Hardware Design

The hardware design is very simple. Its block diagram is shown in Figure 1, a photograph of the PC board is shown in Figure 2, and a list of the major parts is shown in Table 1. The FPGA is physically and logically the center of the design with the other chips grouped around its edges. The Ethernet controller is on the right in Figure 2 and is connected to the FPGA with a 16-bit data bus and some control signals. The Ethernet controller has its own 25 MHz crystal, some coupling and bypass capacitors, some load resistors, and an RJ45 jack with built-in magnetics (transformers). The controller is a LAN9115 by SMSC and it operates full or half duplex at 10 or 100 megabits per second. It is a complete Ethernet solution and includes its own PHY (correct physical interface) and its own MAC (media access control) with speed negotiation capability. It has 16k bytes of memory shared between transmit and receive, and it stores received Ethernet packets until the FPGA gets around to processing them.

The transmit DAC is located below the FPGA in Figure 2 together with a smaller 8-bit DAC. The transmit DAC is 14 bits and operates at the 122.88 MHz clock frequency.

The 8-bit DAC generates 0.125 to 1.25 V DC to control the output level of the transmit DAC. I use this to reduce power to 50 watts on 60 meters, and to compensate for lower gain at higher frequencies in my amplifier chain. The output of the transmit DAC goes through a low-pass filter to a 2N5109 feedback amplifier.[7] Note that these are built directly on the board using "ugly" construction. Maybe next time I will make traces for them. The low-pass filter is designed for an input impedance of 180 Ω to reduce the loss from the impedance matching.

Referring to Figure 1, on SSB transmit the microphone audio is processed by the PC into in-phase and quadrature samples (I/Q) at 48 kilohertz, and sent via Ethernet to the FPGA. The FPGA interpolates (increases sample rate) and tunes these to the output frequency as described below, and sends the digital data to the DAC. The analog RF from the DAC is low-pass filtered and amplified, and sent to the power amplifier chain. For CW, note the 5 V key input. This is high for key down and low (0 V) for key up. It is used to directly generate CW in the FPGA to eliminate any pauses or delays that may be introduced by the PC.

The receive circuitry is located to the left of the FPGA in Figure 2. Input RF goes to a Minicircuits 1:4 unbalanced to balanced transformer, and then to an LTC6405 balanced preamp. The preamp output goes through a low-pass filter to the ADC. The low-pass filter is used to reduce noise above the Nyquist frequency of 61.44 MHz. This noise would fold back into the pass-band and degrade our noise figure. The ADC is a 14-bit Texas Instruments ADS5500IPAP operating at the 122.88 MHz clock. If the preamp and filter were omitted, the ADC could be used up to its analog bandwidth limit of 750 MHz, but that would compromise the performance at HF. The ADC connects to the FPGA and sends 14 bits of data plus a clock that is synchronous with the data. Referring again to Figure 1, the ADC samples go to the FPGA where they are tuned to baseband, decimated (sample rate reduced) and sent to the PC via Ethernet.

Below and to the right of the DAC is the Crystek 122.88 MHz low phase noise clock oscillator. Note that this connects directly to the ADC and DAC as well as the FPGA. This creates some problems, but I was worried about degrading the phase noise if I ran the clock through the FPGA to the other chips. The strange clock frequency was chosen because it is an integer multiple of 48 kHz, and the part was available from Digikey. We eventually need to play received audio at a rate that the sound card can handle, and 48 kHz is a good choice. If the original clock were not an integer multiple of the play rate, we would need non-integer decimation, and that is much more complicated.

Table 1
Major Parts List

Item	Part Number
FPGA	Altera EP3C25Q240C8N, 240 pins
FPGA Flash Memory	EPCS16SI8N in 8-SOIC
Ethernet Controller	LAN9115 by SMSC in 100-TQFP
Clock	Crystek 122.88 MHz CVHD-950-122.880
Receive ADC	125 MHz 14-bit Texas Instruments ADS5500IPAP in 64-TQFP
ADC Preamp	Linear Technology LTC6405 in 8-MSOP
Transmit DAC	Analog Devices 14-bit AD9744 in 28-TSSOP
Transmit Level Control DAC	Analog Devices AD7801 in 20-TSSOP
RJ45 Ethernet Jack	Stewart SI-50170-F
Ferrite Chips	Murata BLM41PG102SN1L in 1806

Directly above the FPGA is its flash memory program storage chip and a 5X2 header to program the flash. Unlike a microcontroller, an FPGA lacks its own flash, so it reads its program out of the memory chip on power up. You may notice a bank of eight small LED's above and to the right of the DAC. These were used for debugging during development, and enabled me to output a byte of data. Now they are used to indicate various error conditions. On the left wall of the box are six LED's. Three connect to the Ethernet controller, and indicate link, activity, speed and full duplex. The other three connect to the FPGA and indicate ADC clip (over range), errors, and on-the-air status. On the top wall of the box are three low dropout voltage regulators for 5 V, 3.3 V and 1.2 V. There is another 2.5 V regulator on the board to the top right. Input power to the box is 6.0 V.

The PC board layout was done with Eagle[8] and all the files including a schematic are available from the authors Web site[9] or from the ARRL's *QEX* binaries site.[10] The parts used were chosen for performance, but also for availability and ease of construction. I chose parts with feet (OK, small feet) because I was not quite ready to try to solder leadless packages.

To me, the most interesting thing about this hardware design is that it doesn't do anything in particular. It has an Ethernet to FPGA link that could be carrying anything, an RF output from a DAC that could generate anything, and an RF input to an ADC that could sample anything. Just looking at the hardware, it could be a transceiver. But it could also be a spectrum analyzer with a tracking generator. To make it a transceiver we need software.

FPGA State Machines

FPGA software is intrinsically parallel, and, left to itself, it will execute every line of code at once. That can be useful, but it is not always what we want. For example, reading from a memory bus requires setting the address on the bus, waiting, asserting the read control line, waiting, reading the data from the bus, waiting, and finally de-asserting the read line. These operations must be sequential and correctly timed. Suppose we receive an Ethernet packet. We must examine the packet headers to decide what block of code should handle the packet, and this also requires sequential operations.

To obtain sequential operation from an FPGA, we use a state machine. A state machine has a state variable that can assume one of a number of values. Based on that value, we perform certain operations. To perform different operations in fixed order, we change the value of the state variable to the next state. For example, to program the memory bus read described above, we would start in the "set address" state. This state would set the address on the bus, set the "next state" variable to "assert read," and set the state to "wait." At the next clock, the state would be "wait." In the "wait" state we increment a counter, wait for a timeout, and then set the state to the value of "next state." In our example, after the timeout, the state would change to "assert read." At the next clock, our state is "assert read," so we assert the read control line, set "next state" to "read data," and set the state to "wait." After another wait interval, the state would change to "read data." At the next clock, our state is "read data," so we read the data from the bus and store it somewhere, set "next state" to "de-assert read," and enter another wait state. After the wait, the "de-assert read" state de-asserts the read control line, and changes the state to an "idle" state. Much of the time the FPGA has nothing to do, and is waiting to receive an Ethernet packet, or to have an ADC data block available to send. The "idle" state typically checks to see if there is any work to do. If there is, it changes the state according to the job to be done.

FPGA Software

There are three major blocks of software running in the FPGA: handle Ethernet reads and writes, write transmit data to the DAC, and read receive data from the ADC. All three blocks run simultaneously. All FPGA code is written in the Verilog language. The FPGA has a fixed IP address of 192.168.2.196, and uses three UDP ports. Port 0xBC77 receives control for the ADC samples and sends sample from that port. Port 0xBC78 receives control data such as transmit frequency and mode. And port 0xBC79 receives the transmit audio.

The simplest block of software reads the 14-bit data from the ADC at the clock rate of 122.88 MHz. It then tunes the data to baseband using CORDIC, and reduces the sample rate by a factor of eight with the first CIC filter. The data then goes to a second CIC filter that reduces the data rate by a factor of 2, 4, 5, 8, 10, 20 or 40 as programmed by a control byte sent from the PC. The data then goes to a final FIR filter that reduces the rate by eight. We now have a final sample rate of 48 to 960 kHz. Samples consist of two 24-bit numbers (I and Q), and are written to a memory buffer for transmission to the PC via Ethernet. The data rate at 960 kHz is 46.08 megabits per second or a little higher if we include packet overhead. That is a significant part of the 100 megabit Ethernet bandwidth and will stress all but the fastest PC's and Ethernet switches. At this rate we can see almost a megahertz of spectrum, and that is not much use on HF. But it could be useful on microwave frequencies if the hardware is used with a transverter. I usually set my sample rate to 240 kHz as a compromise between seeing a good part of the band, and having the signals be wide enough to tune accurately with the mouse.

The next block of code is used for transmit. For SSB transmit, the PC sends audio samples at a rate of 48 kHz, and these are stored in a memory buffer in the FPGA. When the buffer is half full, the FPGA starts reading the samples; it then interpolates them by factors of 20, 16 and 8 in CIC filters and sends them to the DAC. The buffer provides a store of samples in case the PC samples are delayed. For CW transmit, the PC is not directly involved. Instead, the FPGA reads its key input pin. When the key goes down, a counter counts up to a maximum value, and when the key goes up, the counter counts down to zero. The counter is the input to the CORDIC mixer that tunes the DC counter to the output frequency. The result is a shaped key envelope with no delays from the PC.

The most complicated block of FPGA code is used to manage the Ethernet controller. First we initialize a register in the FPGA to zero. Then on power up, we start counting this register until we reach a timeout value. Until we reach timeout, we assert a reset pin connected to the Ethernet controller. Actually this reset pin is connected to all chips that can be reset, and is monitored by software routines that need initialization. In this way, we insure that power is stable before we start operating.

Next, we program some registers in the Ethernet controller by writing fixed byte values on the bus, and then we change the state of the FPGA state machine to "idle," In the "idle" state, the FPGA first checks whether there is ADC data ready to be sent to the PC. If there is, it transmits an Ethernet packet with 42 bytes of header, a sequence byte, a status byte that includes the key up/down state, and 1,440 bytes of data consisting of 240 samples. If there is no ADC data, we check to see if an Ethernet packet was received. If no packet was received, we check the Ethernet controller for error conditions, and turn on the on-board LED's for any errors found. Then we enter a short wait and repeat.

If we receive an Ethernet packet, we read the first 80 bytes into a buffer, and examine the buffer to see what to do next. If the packet is an ARP request for our IP address, we send an ARP response. The ARP protocol is used to associate Ethernet addresses with IP addresses, and it makes the transceiver play nicely on your network. If the packet is a ping request to our IP address, we reply with a ping response. Ping isn't completely necessary, but it is useful and should be included in any Ethernet appliance. If the packet is a UDP packet addressed to us we continue to process it. Otherwise it is silently discarded, and we read and discard any remaining bytes in the Ethernet controller to clear it for the next packet.

If the UDP packet is addressed to our ADC sample port and has data 0x7272, we record the return address and use it to send our ADC samples. That way we do not have to program the IP address of the PC into the FPGA. If the data is 0x7373, we stop sending ADC data. If the UDP packet is addressed to our control port, we read the transmit frequency, receive frequency, the transmit level (for the 8-bit DAC), the transmit mode (CW

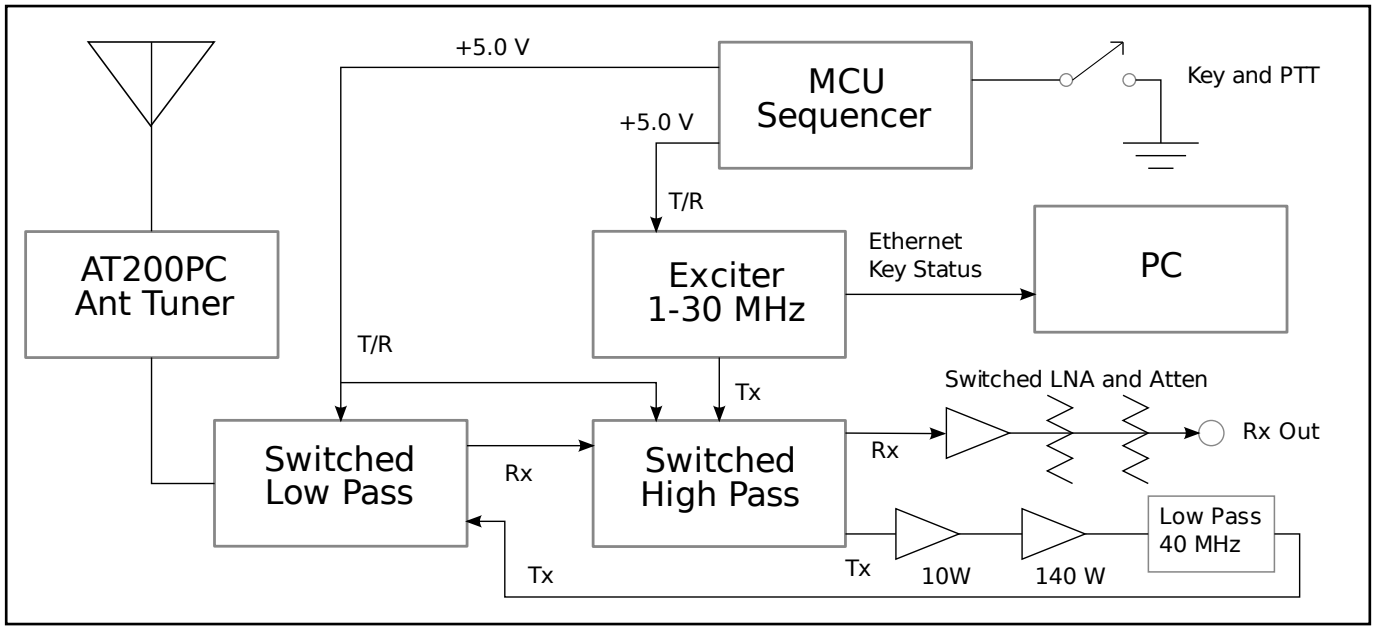


Figure 3 – A block diagram of my station control.

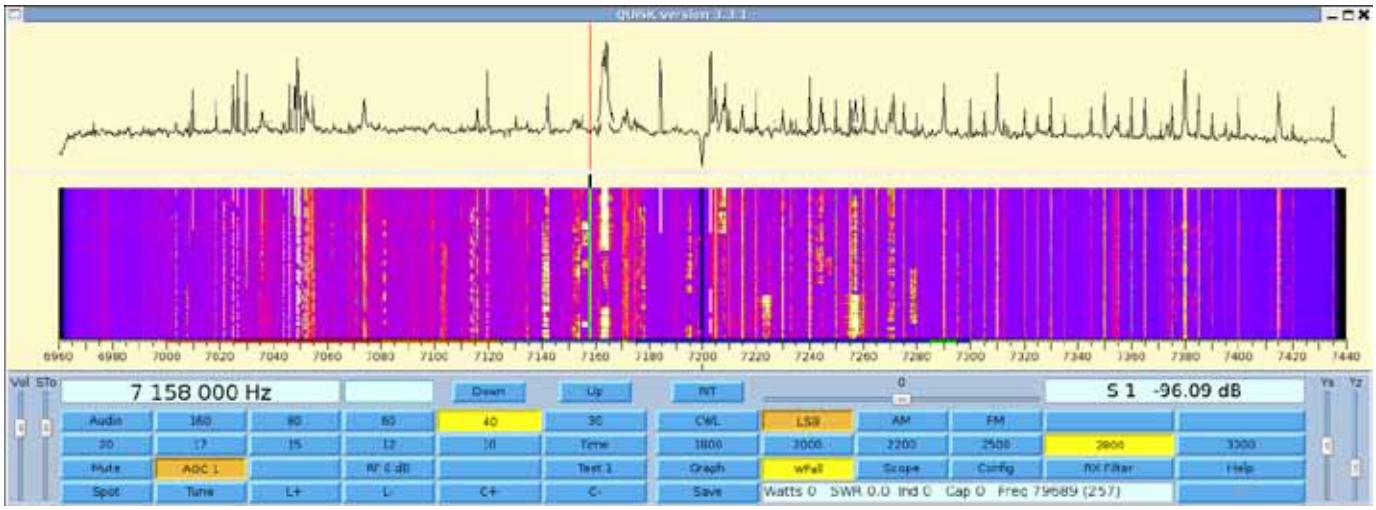


Figure 4 – A screen shot of Quisk running on my PC.

or SSB) and the requested stage two decimation for the ADC. We then echo the control packet back to the sender as an acknowledgement. If the UDP packet is addressed to our transmit audio port, we skip the two byte sequence number, write the audio from our buffer to memory, and then copy the remainder of the audio from the Ethernet controller to memory. The data is two 16-bit numbers (I and Q) at 48 kHz, or 1.536 megabits per second (plus overhead), a modest rate even on 10 megabit Ethernet.

Station Control

To make a complete station the exciter has to be combined with a power amplifier chain and filters. A block diagram of my station is shown in Figure 3. The antenna tuner is an AT200PC by LDG controlled by the PC through a USB to serial adapter. There are two filter boxes consisting of conventional LC filters switched by relays. The low-pass filters are the usual high power (150 W) transmit filters. The high-pass filters are low power LC filters. The combination results in a band pass response on both receive and transmit.

On receive, the signal goes through the low-pass filter, the high-pass filter, and then possibly through a switched preamp or attenuators. On transmit, the RF output from the exciter goes through the high-pass filter, then to a series of power amplifiers, then through a TVI filter, and finally through the low-pass filter to the antenna. My current antenna is a fan dipole for 60, 40 and 20 meters.

The relays require sequenced switching, so the key is connected to a microcontroller that generates keying signals for the relays and exciter. When the key goes down, the relays are switched first, followed by the exciter, and the reverse on key up. Note that there is no key line to the PC even though the PC needs to know the key state so it can mute the audio on transmit and substitute a sidetone on CW. The key status is sent to the PC using Ethernet. It is encoded in the second byte of data in an ADC sample block. At one time I used a pin on the parallel port, but newer PC's often don't come with a parallel or serial port.

The station control shown is quite generic, and has been used with several past versions of my station. It worked just as well with my SDR-IQ receiver and 2008 exciter. I am continuing to work on the power amplifier chain, as I am having trouble keeping the gain flat with frequency and the IMD low.

Quisk Software

My Quisk software running on a PC controls my whole station. My PC is an old 3 GHz Pentium 4 with one core. The PC controls the AT200PC antenna tuner through a USB to serial adapter; it controls the filter boxes through Ethernet and an I2C gateway; and it directly controls the transceiver by Ethernet. A screenshot of Quisk is shown in Figure 4. It shows the whole 40 meter band sampled at 480,000 samples per second. Quisk software is available on my Web site.[11]

Quisk was originally designed for CW QSK operation using the sound card as the receiver, and a DDS chip for transmit. Gradually I added the capability to control more hardware: an SDR-IQ receiver by RfSpace, my 2008 exciter, an AOR receiver with an IF output (as a panadapter), and now my transceiver. Quisk uses an external hardware file written in the Python programming language to control its hardware. To use different hardware, you just change this file. You can use one of the hardware files that comes with Quisk, or write your own. There is also a way to add additional controls to Quisk by adding a custom *Python* "widgets" file. *Python* is a powerful language that is very easy to learn. I use it for Quisk and for nearly everything else. Its complex data type is handy for impedance calculations, for example.

The top level of Quisk is written in *Python*, and controls the graphical user interface (GUI), the graph and waterfall screens, and any custom hardware. The rest of Quisk is written in the *C* language, and this code is responsible for the sound cards, the USB interface, the Ethernet IP ports, the FFT (fast Fourier transform) and the digital signal processing. Linking *Python* with *C* in this way is very common. The main program quisk.py is 2300 lines, and the *C* code is 4500 lines. This is a very small program.

Quisk was never intended to be a "product" in the sense that it had every feature imaginable and could run any hardware right out of the box. I always saw it as a starting point for other people's homebrew projects; something that would be simple and understandable, and easy to adapt to different hardware and software designs. That is one reason that Quisk runs on Linux. Linux comes with *Python* and *C* compilers, and it provides a rich software development environment for free. Being simple, small and easy to change is one of the design goals of Quisk. But even though Quisk is simple, and after using other available radio software, I find I prefer Quisk. Some of that is no doubt pride of authorship, but I want my radio to look like a radio, not a Windows program, even though it is an image on a computer screen. And I find that Quisk is easy to tune and has the features I need.

Although I use Quisk to control my transceiver, that is not a requirement. The trans-

ceiver is simply sending and receiving I/Q samples just like any other software defined radio. Any software can provide the data the transceiver requires. All that is needed is an Ethernet IP/UDP interface, and that is simple to supply.

Quisk Internals

Quisk runs in two threads, a GUI thread that runs the user interface, the screens and the FFT, and a second thread that is responsible for reading and writing the sound and performing digital signal processing. When used with my transceiver, Quisk operates as follows. On receive, the sound thread reads a UDP port at timed intervals and collects any ADC samples available. These samples are first copied into an array for the FFT. When enough samples are available, the GUI thread will perform the FFT, average the results, and eventually update the screen. The sound thread then tunes the signal to baseband in a digital mixer, reduces the sample rate, filters the signal and divides it into an I/Q pair and demodulates it. For FM, there are additional filters at this point. There may be more decimation or interpolation depending on the sample rate and source. Finally, AGC is applied and the sound is played on the sound card.

For SSB transmit, the sound thread reads the microphone and boosts the high frequencies in a digital filter. It then filters the audio and divides it into an I/Q pair. This signal is mixed to a higher frequency, clipped, re-filtered, and mixed back down to baseband. These audio samples are sent to the UDP port and then to the transceiver.

There are a few other interesting features. The S-meter in Quisk is calculated by squaring and averaging the correct number of FFT bins to equal the indicated filter width. This means that Quisk has a true RMS voltmeter with a known noise width that can be used to make noise measurements. Quisk can generate a two-tone test pattern to make IMD measurements. Quisk generates these tones in the PC software and sends them as an SSB signal. And Quisk has a spot switch that sends a CW carrier in SSB mode. That is not too exciting, except that since everything is digital, the level is guaranteed to be at PEP.

Problems

I am very happy with this transceiver hardware design, but there are still some unresolved problems and directions for future work. I measured the noise figure at 23 dB. Although this is usable, I was expecting a noise figure closer to the preamp noise at 200 Ω of 8 dB. I found this very confusing, but thanks to an email[12] from Jeff Millar, WA1HCO, I now understand the high noise figure. He calculated the noise figure of the ADC alone to be 30 dB. The preamp gain is not sufficient to dominate the noise figure, and using the usual formula F = F1 + (F2 - 1) / G1 gives the net noise figure I observe. I do not want more preamp gain, as that would degrade the dynamic range. I have an additional preamp external to the transceiver that I can switch in for 20 meters and up. The purpose of the preamp is to make it easier to drive the receive input.

I am also worried about the clock integrity. I connected the clock oscillator to three IC's, but there is no spec for how many IC's the clock will drive. I could connect the clock to the FPGA and run the clock straight through, but that might degrade the jitter. I could use a clock distribution IC, but the ones I saw had worse specs than my oscillator. I don't have any evidence that the clock is questionable, but I don't have any equipment to measure the clock noise either.

And then there is the power amplifier chain. The IMD spec out of the DAC is exemplary, but it quickly degrades with amplification. On 10 meters, it winds up about 23 dB below one tone, and I am working on improving this. Even good quality commercial ham equipment seems to have IMD specs in the low 30 dB range below PEP. If I can get a cleaner power amplifier chain, then maybe I can bend the response curve in software to improve the IMD.

Conclusion

It has been great fun working with these new digital IC's and getting a lot of hands-on experience with digital signal processing. I am not convinced that an all-digital radio is better than the best analog radio, but I think it is easier to homebrew one, especially since all the software is readily available for free. When I get on the air with it and say that my rig is homebrew, my ham friends take an interest, and that is very gratifying. I would also like to thank the hams that have emailed me and expressed an interest in my work.

I also learned that even in this digital age, some things in radio never change. As pointed out by Wes Hayward et al, building a receiver is a great way to learn humility. If there were a magic bullet formula for a receiver, they would all be built the same way, but instead, receiver design remains a challenge. And filters remain at the heart of radio whether digital or analog. And then there are antennas. It was amazing to me how much better my first digital receiver worked when attached to a good antenna! When I tune across the bands, I hear hams with digital rigs, analog rigs, and some with classic Collins or Drake gear, but one thing I know for sure; they definitely have an antenna.

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]James C. Ahlstrom, "An All-Digital SSB Exciter for HF", *QEX* May/June 2008, pp 3-10.
[2]J. Bhasker, *Verilog HDL Synthesis, A Practical Primer*, Star Galaxy Publishing, 1998.
[3]J. Bhasker, *A Verilog HDL Primer*, Third Edition, Star Galaxy Publishing, 2005.
[4]Richard G. Lyons, *Understanding Digital Signal Processing*, Second Edition, Prentice Hall, 1996.
[5] **www.arrl.org/software-defined-radio**
[6]Ray Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers", **www.andraka.com/files/crdcsrvy.pdf**.
[7]Wes Hayward, Rick Campbell and Bob Larkin, *Experimental Methods in RF Design*, ARRL, 2003, ISBN: 0-87259-879-9.
[8]**www.cadsoftusa.com**.
[9]**www.james.ahlstrom.name/transceiver**
[10]**www.arrl.org/qexfiles**
[11]**www.james.ahlstrom.name/quisk**
[12] Jeff Millar, WA1HCO, personal communication