Scotty Cowling, WA2DFI

P O Box 26843, Tempe, AZ 85285; **scotty@tonks.com**

Hardware Building Blocks for High Performance Software Defined Radios

The author explores some alternative hardware for software defined radio use.

Low-cost, highly capable digital hardware is proliferating everywhere. Do names like Arduino, Beagle Board, Raspberry Pi, BeMicro or SoCkit mean anything to you? Is this some kind of secret code that has to do with very small Italian dogs that eat spherical fruit for breakfast? Not hardly! While the first three are names of low-cost embedded microcontroller (MCU) boards, the last two represent their counterparts in the world of programmable hardware. Wait, you say; microcontrollers *are* programmable hardware. This is true, but what makes BeMicro and SoCkit boards different is that they each contain a Field Programmable Gate Array (FPGA). See the "MCU Versus FPGA" sidebar for a look at the differences between an MCU and an FPGA.

An MCU executes instructions from a pre-defined instruction set in sequential order; the hardware is fixed, but the sequence of instructions is programmable. An FPGA, on the other hand, has no fixed instruction set or sequence of instructions, operates on data in parallel and has programmable logic and interconnections. Software defined radio (SDR) implementations can benefit greatly from the FPGA parallel hardware architecture.

And Now for Something Completely Different — A Software Defined Radio!

A few RF boards can be added to standard, off-the-shelf digital development kits to build a high-performance software defined radio. What these SDRs lack in polish, they make up for in performance. When you assemble a collection of boards, some of which were designed for a completely different purpose than building an SDR, what you end up with may not look pretty. If performance is your goal, however, you will not be disappointed.

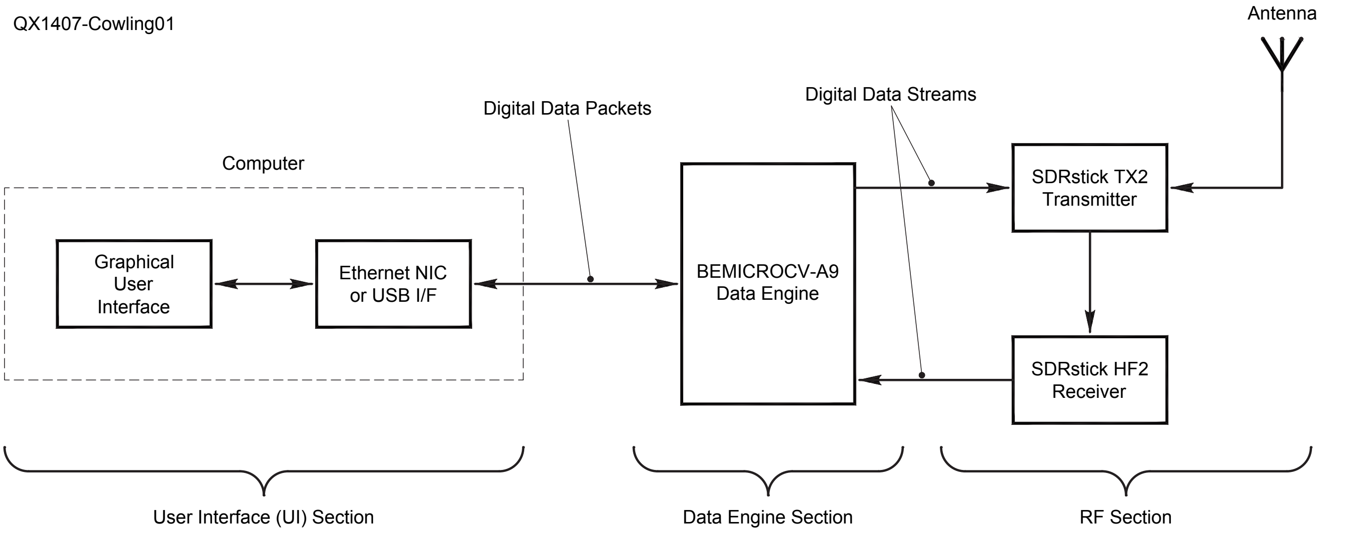

Figure 1 — An example of functional SDR sections.

I will present a hardware overview of the BeMicro series (BeMicro, BeMicroSDK, BeMicroCV and BeMicroCV-A9) FPGA development boards and the SoCkit System-on-a-Chip (SoC) FPGA development board, along with the SDRstick™ RF front-end boards necessary to transform them into a full-fledged digital down conversion receiver or a digital up conversion transmitter, or both at the same time!

Together, these boards make several different configurations of very high-performance digital down conversion/digital up conversion radios possible for only moderate cost.

System Functional Categories

We can break an SDR system down into three main functional categories (see Figure 1 for an example), starting from the operator and working our way toward the antenna:

1) User interface (UI)

2) Data engine

3) RF section (receiver front-end, transmitter strip)

This is definitely a simplified view, but still useful. Common SDR processes such as decimation, modulation and demodulation, digital filtering, data formatting and so on, are not necessarily confined to any one of the above broad functional categories. We will touch on these processes later on, but we will not dwell on the technical details. We will piece together our SDR and leverage the work others have already done to make it work.

The User Interface — 31 Flavors?

The user interface comes in many varieties: for different radios (such as openHPSDR, RF Space SDR-IQ™, FlexRadio Systems Flex-6000™); for different operating systems (*Windows*, *Linux*, *MAC OS*, *Android*); and for different specialized uses (such as GNU Radio, CW Skimmer, QtRadio). I have lumped these all into one category because they have one thing in common: they are the interface between the human and the radio. In reality, these software programs and the computers that run them perform many more tasks than just their one self-described function. In general, the user interface also performs modulation and demodulation, digital filtering, time-domain to frequency-domain conversion (for waterfall and panadapter displays), and control of all radio hardware and hardware DSP functions. As we shall see, some of these functions may be performed in the data engine, or split between the user interface and data engine. In either case, the functions are already coded and working. All we have to do is connect them properly.

The Data Engine — Heavy Lifting

The data engine is the bridge between two data domains. On one side is the freshly digitized data from the receive portion of the RF section or the digital data stream destined to be converted to analog RF in the transmit portion of the RF section. On the other side is the data to and from the user interface, which is typically some type of computer. Notice that there are no analog components to the data; the conversion between analog signals and digital data takes place in the RF section (and sometimes in the user interface section in the case of microphone or receiver audio). The data engine deals strictly with digital data.

Since the data coming from the RF section (samples from the receiver's ADC, for example) is in a different format than the data that is destined for the user interface section, the data engine must perform this conversion. The same is true for the outgoing data stream coming from the user interface section and ultimately destined for conversion to an analog transmit signal by the transmit DAC. The communications interface on the user interface side of the data engine in common SDRs today is either USB or Ethernet. Both USB and Ethernet use packets to transfer data, and since the raw data from ADCs or to DACs are in streams, packetization and de-packetization of the data must also be performed by the data engine.

The data engine may simply re-format raw data into packets (or packets into raw data), but often times the available data bandwidth on the user interface side does not match the required bandwidth on the RF section side. In the case of a low rate receiver ADC sending data over a high speed interface to the user interface, or a high speed interface from the user interface sending data to a low rate transmit DAC, the solution is easy since the packetized nature of USB and Ethernet allow idle periods where no data is sent. In the opposite case (high rate ADC to low speed interface or low speed interface to high rate DAC), receive data will be lost (user interface over-run), or the transmitter DAC will be starved for data (RF interface under-run). For example, in the HF2 digital down conversion receiver described later, the 16-bit ADC clocked at 122.88 MHz produces almost 2 Gbit/s of raw data. This data rate far exceeds the capacity of both high speed USB 2.0 at 480 Mbit/s and Gigabit Ethernet at 1 Gbit/s. The fact that both USB and Ethernet have packet overhead only makes the problem worse. The solution is implemented in the data engine hardware.

In the receive path, the data engine simply throws some of the data away. This is done in the digital domain by a process known as decimation. The key to building a useful receiver is to know which data to keep. That is determined by the center frequency of the receiver display bandwidth. This is the digital equivalent of the local oscillator in a super heterodyne receiver. An explanation of the mathematics behind decimation is beyond the scope of this article, but there are excellent references.[1, 2]

In the transmit path, the raw data from the user interface is digitally up-converted to a full-bandwidth data stream and sent to the DAC in the RF section. This is accomplished using digital mixing, again determined by the center frequency of the transmit bandwidth. This center frequency is analogous to the local oscillator in a mixer type transmitter.

The RF Section — Still Analog After All These Years

The RF section is the interface between the outside world of RF and the digital world of, well, the rest of the radio! Any receiver front-end protection, filters or attenuators are included in this section, along with transmitter filters, power amplification and antenna switching. Most of these functions may also be performed in the digital portion of the SDR, at least to a limited extent. There are some things, however, that must be done in the analog hardware. An SDR is supposed to be a *digital* radio, so why can't we do everything *digitally*?

One reason is simple, and it also applies to conventional all-analog radios. *The maximum ratings of the devices connected to the antenna port of the radio must not be exceeded.* The other reason applies to SDR receivers, but generally not to all-analog receivers. *Any signals above the Nyquist frequency must be prevented from reaching the input to the receiver ADC.* (The Nyquist frequency in a digital down conversion receiver is half the ADC sample clock frequency.) What happens when either of these rules is violated depends on the particular SDR design, but performance is invariably compromised in some way.

Electrostatic discharge protection devices should be used at the antenna input to protect front-end components, and internal analog attenuators can prevent large input signals from overloading analog low-noise amplifiers (LNAs) or ADC inputs. Of course, attenuators reduce the amplitude of all signals, making them less useful than filters in some situations.

There are two reasons to use analog filters in an SDR. The first and foremost is to prevent any components in the front end from saturating, including active attenuators, low-noise amplifiers or ADC inputs. Once saturation occurs, devices behave in a non-linear fashion; no amount of digital

[1] Notes appear on page 40.

processing is likely to correct this. Filters used for this purpose can be either internal or external, or both. SDRs operated in the presence of strong in-band signals (such as near a powerful broadcast station) or connected to very large antennas may need external filters to prevent front-end overload, but many SDRs need no front-end overload filtering at all.

The second reason to use analog filtering is to prevent any signals above the Nyquist frequency from reaching the input to the ADC. Image signals above the Nyquist frequency will fold back, or alias, onto signals below the Nyquist frequency. Once this happens, the two signals (the desired signal and the image signal) will become indistinguishable from each other. This type of filter is called an anti-aliasing filter, and is most often internal to the SDR.

Data Engines

The SDRstick™ RF boards currently support three different data engines, and more if you include other FPGA development kits with HSMC expansion connectors. The data engines that I will discuss are all manufactured by Arrow Electronics.

BeMicro — Not Quite Enough

The introduction of the first Field Programmable Gate Array (FPGA) System Development Kit created the potential for inexpensive, easy to build (read: plug together) Software Defined Radios. While I can only guess at which one was *the* first one, the BeMicro FPGA development kit (Figure 2) was one of the first.

The original BeMicro FPGA development kit used an Altera EP3C16 Cyclone III FPGA containing about 16 K logic elements (logic elements). The small BeMicro PCB also sported a 16 MHz clock oscillator, 256 K × 16 bit static memory (SRAM), 8 user-programmable LEDs, a USB programming/power port and an 80 pin micro-edge connector (MEC) for I/O. Arrow designed the BeMicro as a showcase for the Altera NIOS II embedded soft-core CPU (see the side bar), and even provided lab materials with design examples to help beginning FGPA users get started quickly. The design tools are all free and downloadable from the Altera website: **www.altera.com/products/software/sfw-index.isp**. The major drawback of the original BeMicro was the lack of any kind of high-speed communications interface. The only two external interfaces on the BeMicro are the USB port and the 80 pin micro-edge connector.

The BeMicro USB interface uses an FTDI interface chip. The FTDI interface chip can only operate at USB low or full-speed bit rates (1.5 or 12 Mbps, respectively). It connects directly to the dedicated FPGA serial programming (JTAG) port rather than general purpose FPGA pins. This makes it difficult for the FPGA programmer to use the USB port for a communications interface.

The BeMicro general-purpose I/O interface on the 80-pin micro-edge connector could be used for a high-speed communications interface. As you will see later, however, we need all 80 of these pins for the interface to the RF portion of our SDR. All in all, the BeMicro qualifies as a resounding "almost enough" hardware platform.

Along comes BeMicroSDK

The successor to the BeMicro, called the BeMicroSDK (the suffix stands for System Development Kit), added enough functionality to the somewhat limited original BeMicro to advance it from the "almost enough" to "now it is possible" SDR data engine category.[3] Let's see what the designers added to make BeMicroSDK (Figure 3) a viable SDR data engine.

For starters, BeMicroSDK sports a 10M/100M Ethernet port. We now have a relatively high-speed communications interface, at least compared to full-speed USB. Ethernet supports routable data packets and is as ubiquitous as USB on modern desktop and laptop computers. The memory size is increased from 512 Kbytes to 64 Mbytes, and changed to DDR SDRAM. A micro-SD memory card socket and a temperature sensor have been added, as well as three push-button and two slide switches. These last few items are not of much use to us in building an SDR data engine, but the Ethernet and DDR memory sure are! In the excitement, I almost forgot to mention the FPGA upgrade: the 16 K logic elements Cyclone III has been replaced by the newer and larger 22 K logic elements Cyclone IV, an EP4CE22 device. The BeMicroSDK became the data engine for the first SDRstick™ SDR.

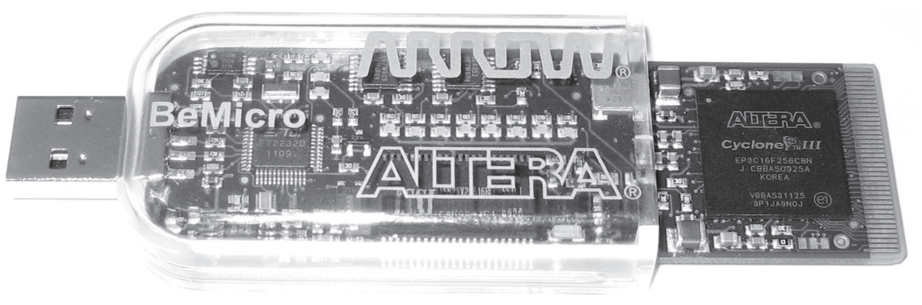

Figure 2 — The Original BeMicro board — not quite SDR material.

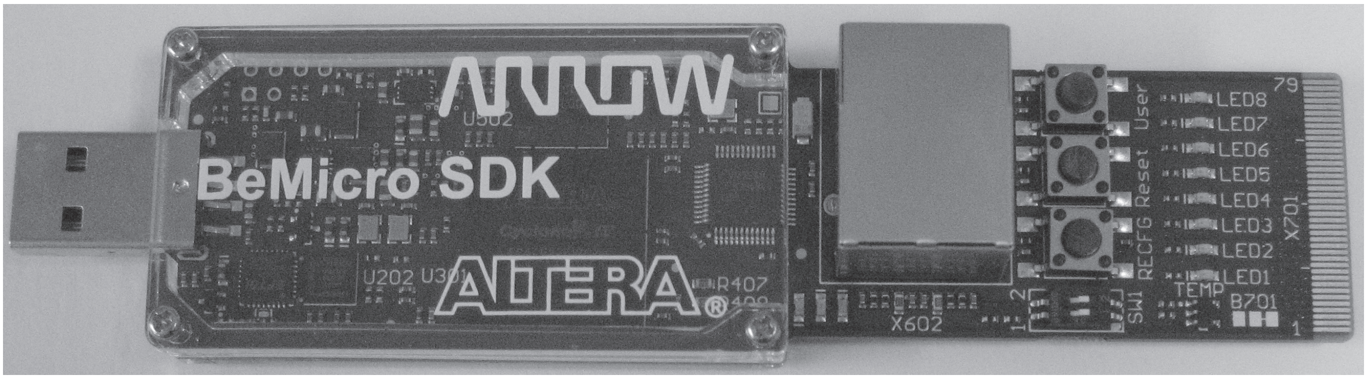

Figure 3 — The BeMicroSDK, with an Ethernet port, makes a good SDR data engine.

BeMicroCV-A9 - More Logic Elements, Anyone?

The evolution of the BeMicro line continued with the introduction of the BeMicroCV (where the suffix stands for Cyclone V).[4] While this is an interesting board (see Figure 4), in many ways it is a step backwards from the BeMicroSDK. It has a newer, but barely larger 25 K logic elements Cyclone V 5CEA2 device in place of the older Cyclone IV. The biggest problem for SDR use is the substitution of 64 GPIO pins on two 40 pin headers for the Ethernet port. While lots of I/O pins are useful for many things, even 64 of them cannot make up for the loss of the Ethernet port. If the BeMicroCV is not suitable for use as an SDR data engine, why do I even mention it? The answer lies in the successor to the BeMicroCV, called the BeMicroCV-A9.

The BeMicroCV-A9 (Figure 5) is different from the BeMicroCV in only two ways, but these two differences make the A9 (as it is called for short) an ideal SDR data engine. The first change is the re-purposing of 19 of the GPIO pins to add a Gigabit Ethernet port. This is a blazing fast channel for communicating with the user interface. The other change is to the FPGA. The A9 board uses (not surprisingly) a Cyclone V 5CEA9, with *301 K logic elements!* It is the largest member of the Cyclone V CE series, and perfectly suited to the job of SDR data engine. Now that we have two candidates for the job of data engine, you might think that we can move on to the RF section. But wait, not so fast; we have one more board that has something to offer the SDR builder.

CPU+FPGA - SoCkit to me!

There is more to life than bread alone, and more to an SDR data engine than just the raw number of logic elements in its FPGA. There is a new kid on the block that offers the best of both the FPGA world and the MCU world. It is called the system on a chip field programmable gate array, mercifully abbreviated SoC FPGA. This relatively new class of programmable device integrates both an FPGA and an ARM processor on the same silicon chip. The processor portion of Altera's SoC FPGA is called the hard processor system, or HPS for short. The term *hard* refers to the fact that the processor is hard-coded into the silicon, and not built by interconnecting programmable logic elements the way a soft-core processor is made. The hard processor system runs much faster than a soft-core processor and requires a smaller area on the silicon die. The hard processor system of the Cyclone V SoC FPGA in our next data engine runs at 800 MHz, whereas most soft-core processors are limited to a clock rate of around 300 MHz.

In a clever combination of the SoC abbreviation and the last word in the term system development *kit*, Arrow Electronics' SoC FPGA board is called SoCkit.[5] This somewhat obscure reference to a 1960s TV comedy show sketch does not escape older readers, I am sure. Young squirts can check Note 6.

The SoCkit board is built on a much larger form factor than the BeMicro series of data engines (see Figure 6). SoCkit's larger physical size allows room for far more features than we have seen so far on our candidates for SDR data engine duty. Here is a partial feature list:

- Altera Cyclone V SoC FPGA with 110 K logic elements and dual-core ARM Cortex-A9 CPU
- Two banks of DDR3 SDRAM (2 GByte total)
- Gigabit Ethernet port
- High Speed Mezzanine Connector (HSMC) expansion connector
- MicroSD Card connector
- USB serial port
- USB 2.0 OTG port
- USB Blaster programmer
- 128 × 64 pixel LCD display
- Video DAC with VGA connector
- Audio CODEC with line out/line in/mic in connectors
- 8 each: LEDs, pushbutton switches, slide switches

This is an impressive list of features for a board that costs under $300![7] Although Gigabit Ethernet is included, as well as a microSD card socket, there is no micro-edge connector like there is on every BeMicro series board. We will have to connect our RF boards to the high speed

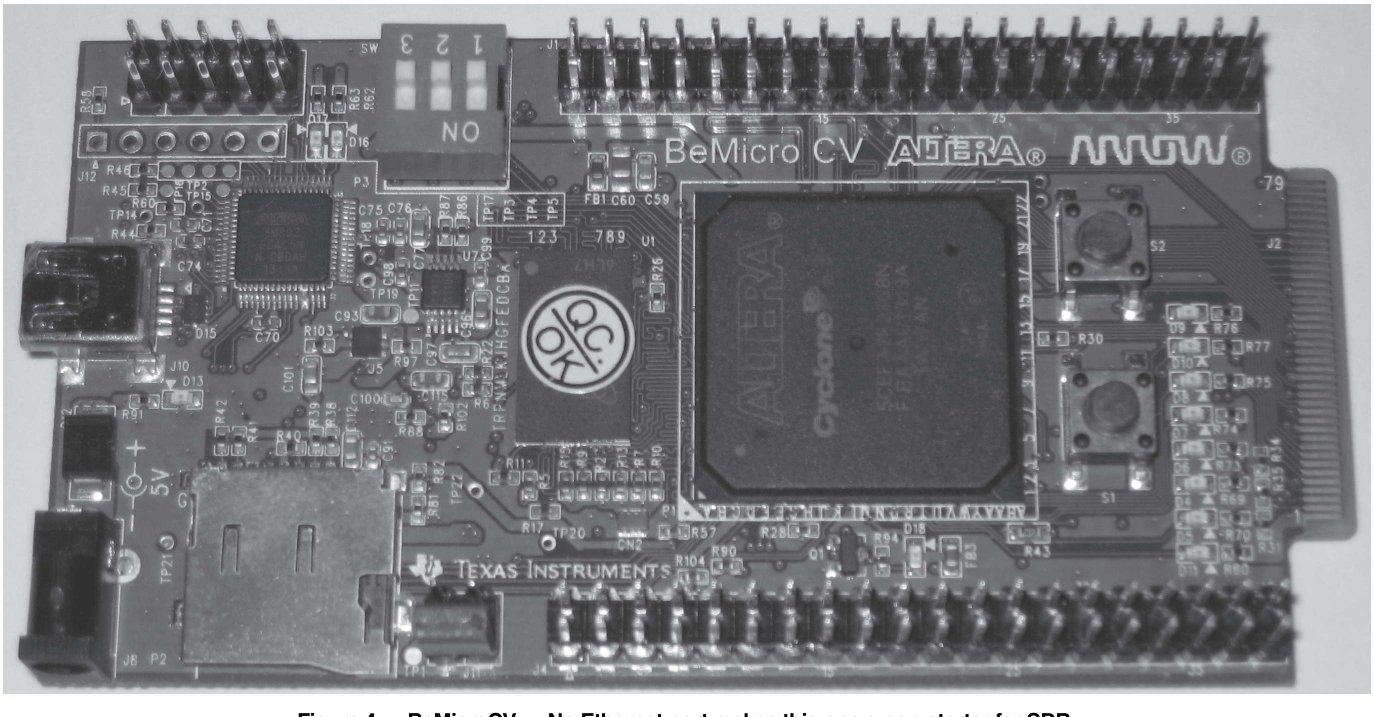

Figure 4 — BeMicroCV — No Ethernet port makes this one a non-starter for SDR.

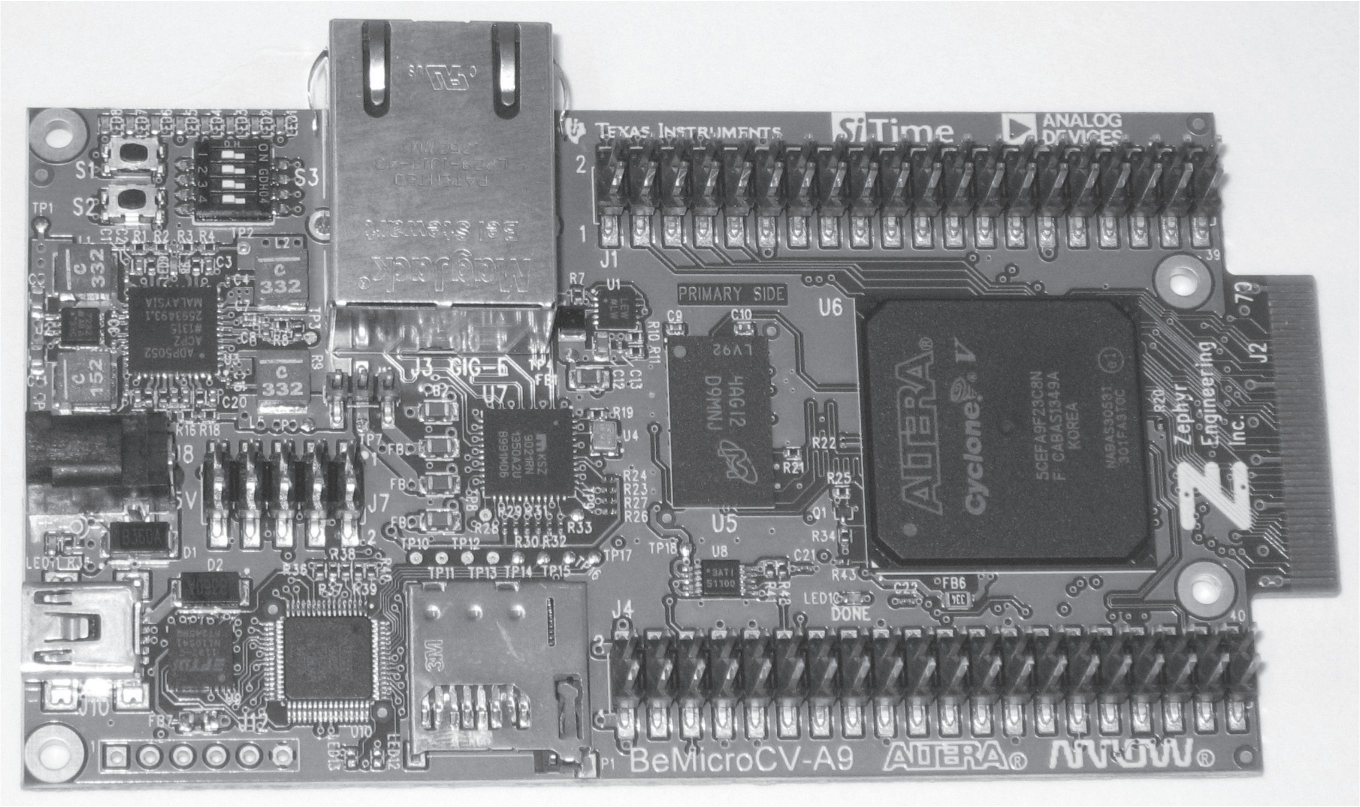

Figure 5 — BeMicroCV-A9, The Ultimate SDR data engine?

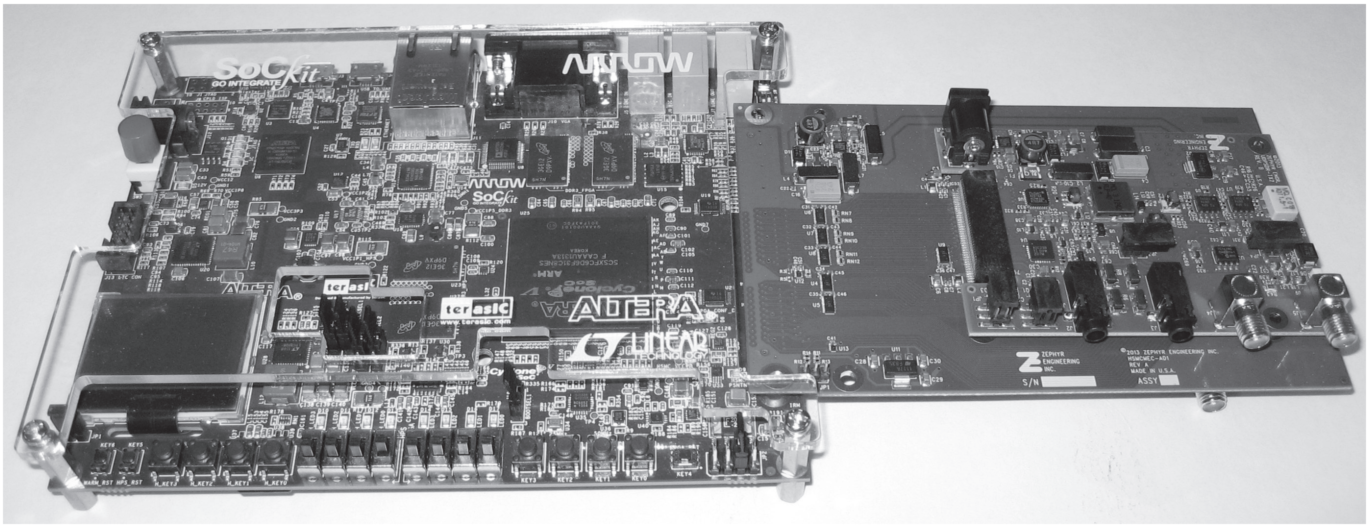

Figure 6 — SoCkit board with AD1 adapter, HF2 receiver and TX2 transmitter.

mezzanine connector (HSMC) instead. The HSMCMEC-AD1 adapter card (AD1 for short) was designed for this specific purpose (see Figure 7). The AD1 adapter plugs onto the high speed mezzanine connector port of virtually any development board (many off-the-shelf FPGA development boards have them) and provides two micro-edge connectors, one male and one female. The male, or board-edge connector accepts HF1 or HF2 receiver boards, while the female socket accepts the TX2 transmitter board. (More on the RF boards in the next section.) The AD1 performs level translation, since high speed mezzanine connector voltage levels can be 1.8 V, 2.5 V or 3.3 V, while the micro-edge connector RF boards all use 3.3 V signaling. The AD1 also provides separate receive and transmit connectors, allowing the TX2 transmitter to be used with the HF1 or HF2 receiver boards. Speaking of receivers, let's move on now to discuss the RF section of our SDR.

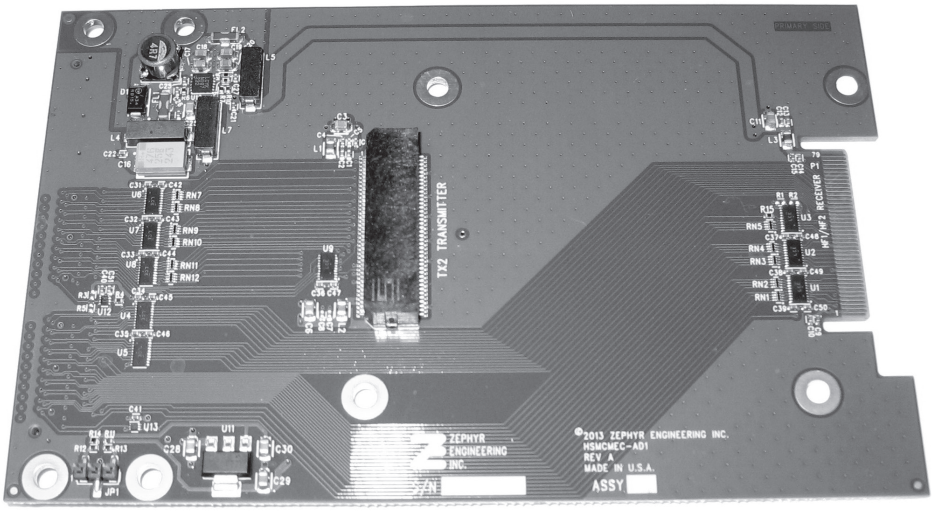

Figure 7 — The AD1 high speed mezzanine connector to micro-edge connector adapter board.

Receivers

There are two models of SDRstick™ micro-edge connector-compatible receiver boards: UDPSDR-HF1 and UDPSDR-HF2. The less expensive HF1 front-end (Figure 8 and Figure 9) performs decently, and employs a 14 bit ADC sampling at 80 mega samples per second. Paired with a BeMicroSDK or BeMicroCV-A9, the HF1 makes a fully functional 100 kHz to 30 MHz receiver. The TX2 transmitter cannot be used to make the system into a transceiver, since there is no micro-edge connector expansion connector on the HF1 receiver.

The high-performance HF2 receiver (Figure 10 and Figure 11) incorporates the same components and a similar architecture to the receive section as the openHPSDR Hermes board.[8, 9] A Crystek CVHD-950 extremely low phase noise oscillator clocks the 16-bit LTC2208 ADC at 122.88 MHz. A programmable 0 to 31 dB step attenuator (Mini-Circuits DAT-31-SP+) ahead of the LTC6400-20 20 dB gain differential ADC driver helps prevent overload. The HF2 receiver board has a transmit expansion micro-edge connector for the TX2 transmitter board described in the next section. The HF2/TX2 combination (along with a suitable data engine) makes a complete transceiver. Table 1 shows a comparison of the features of the two receiver boards.

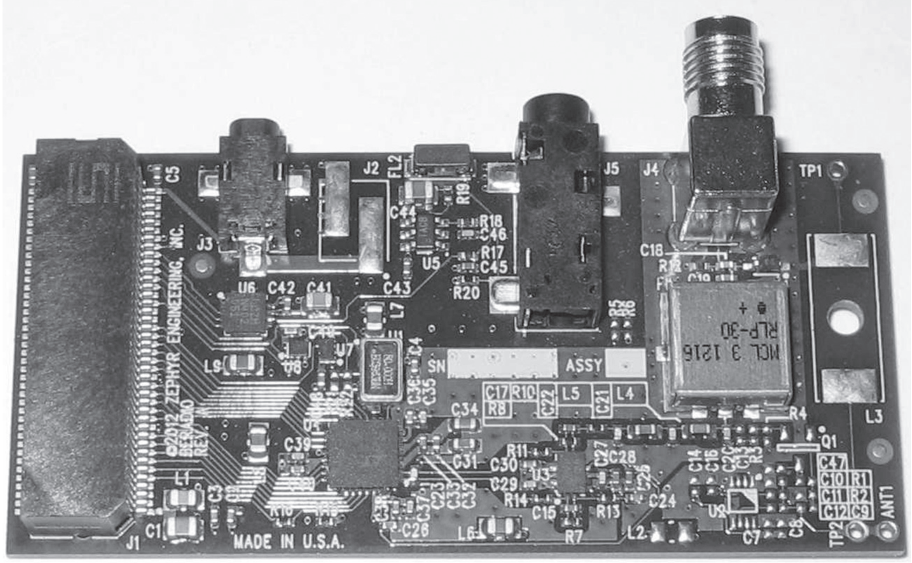

Figure 8 — The HF1 100 kHz to 30 MHz receiver.

Transmitter

The TX2 transmitter (Figure 12 and Figure 13) uses the same components and a similar architecture to the transmit section of the openHPSDR Hermes board (see Note 9). The TX2 uses an Analog Devices

What is a Virtual Receiver?

SDR receivers are built from parallel FPGA logic and software programs that perform mathematical operations (digital signal processing, or DSP) on a raw stream of data from an ADC. The processed digital data stream can be routed to client software for further processing or converted back to the analog domain to drive a speaker. The same raw data stream may feed multiple instances of DSP hardware and software, and each DSP instance can be independently configured (for example, center frequency, modulation type, filtering, and so on). Each one of these instances of DSP hardware and software is called a virtual receiver (FlexRadio Systems calls it a "Slice Receiver.") It is *virtual* because the receiver exists as a series of programmed processes rather than analog hardware, such as mixers and oscillators. Even though it is *virtual*, a virtual receiver is still implemented in hardware; it uses digital logic in place of analog components.

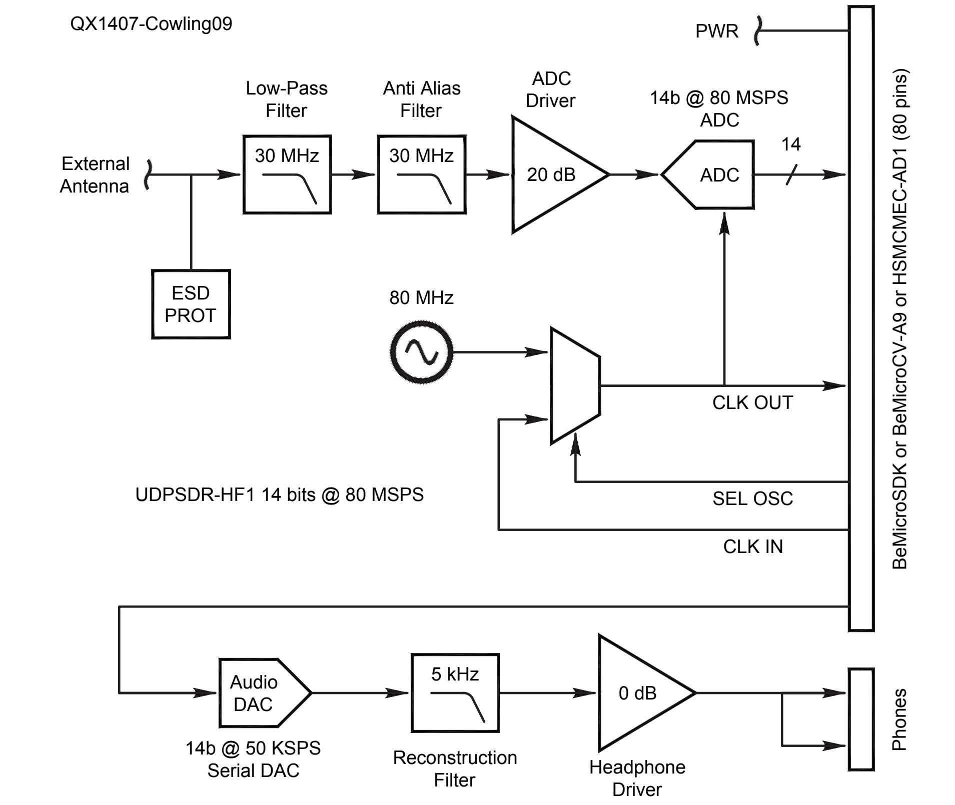

Figure 9 — HF1 receiver block diagram.

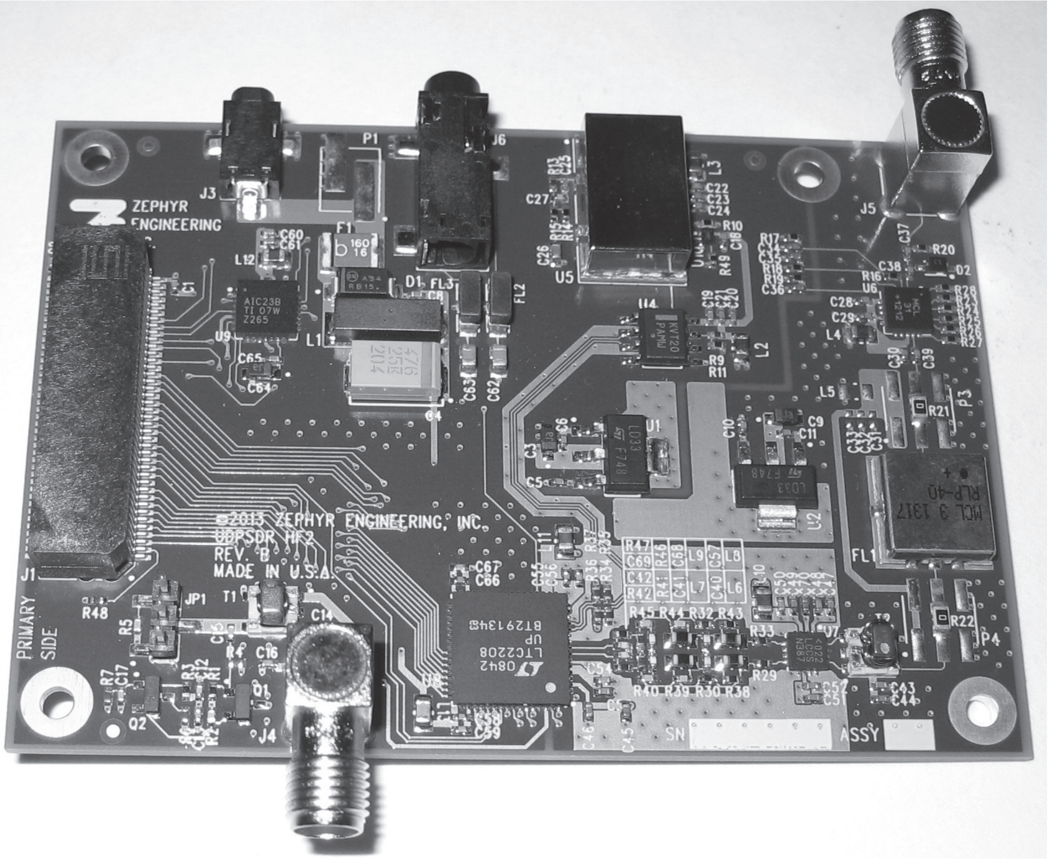

Figure 10 —The high-performance HF2 100 kHz to 55 MHz receiver board.

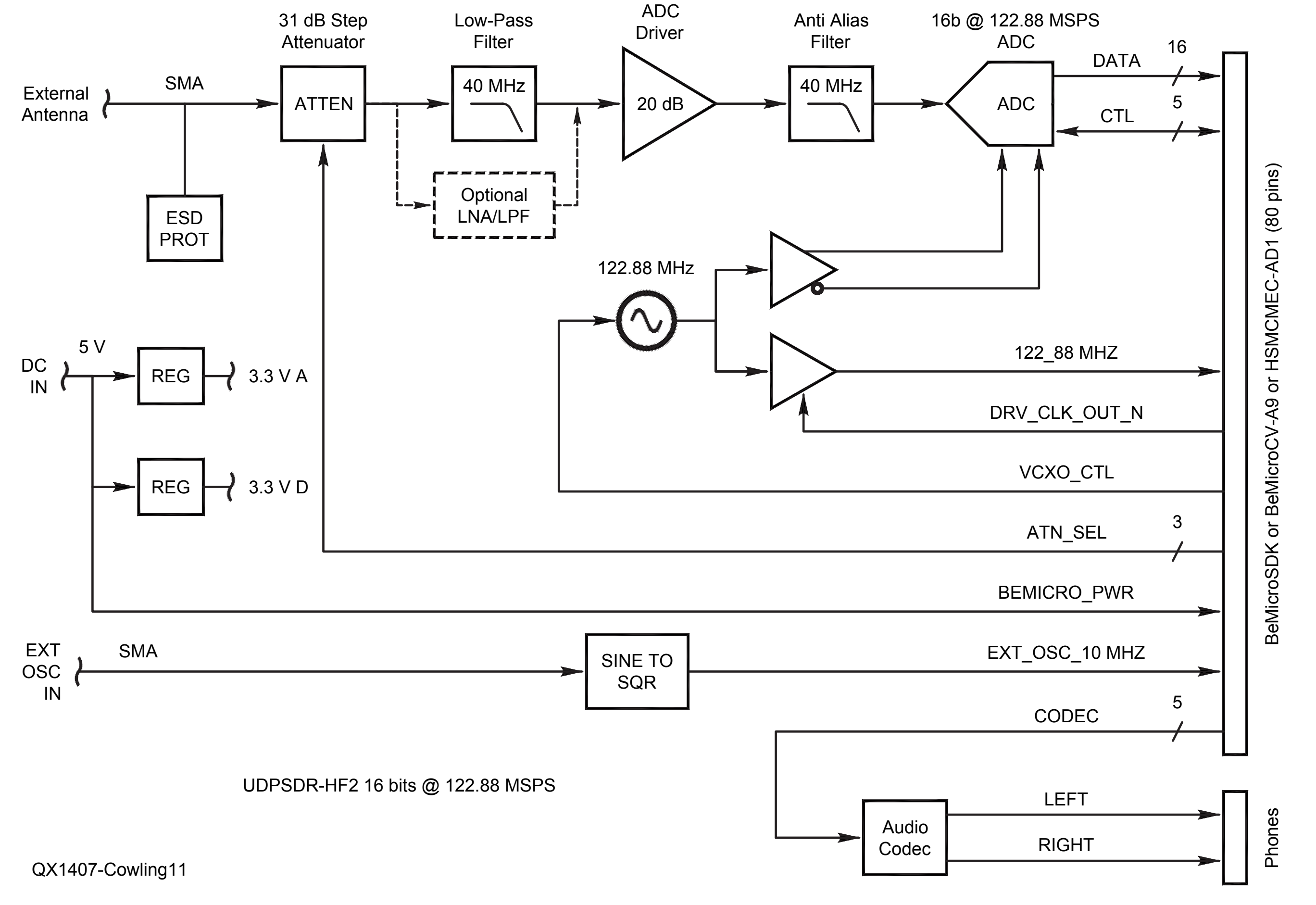

Figure 11 — HF2 receiver block diagram.

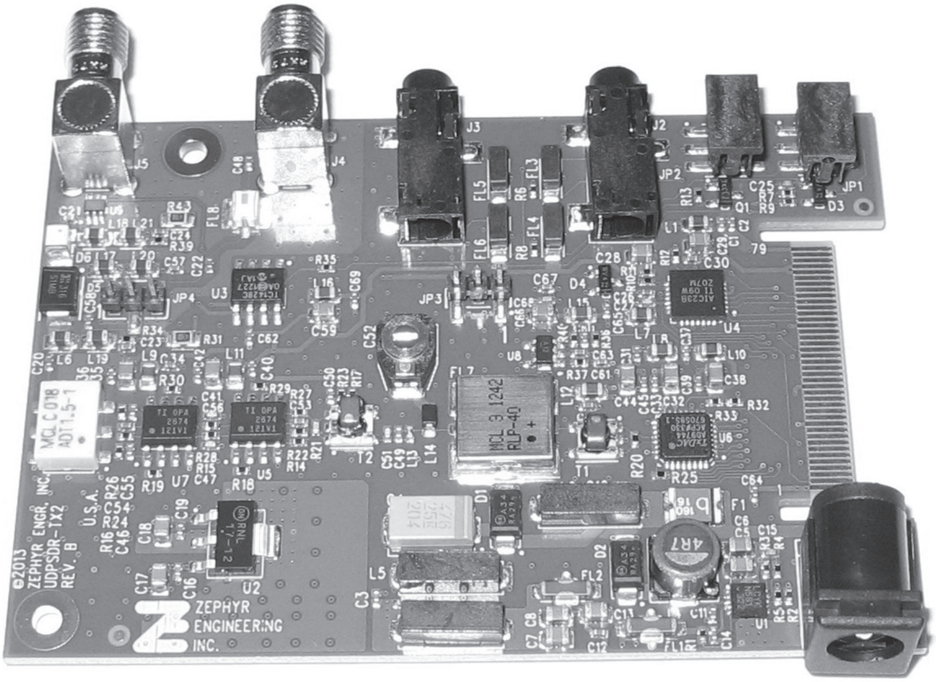

Figure 12 — The TX2 transmitter plugs into either the AD1 adapter or the HF2 receiver.

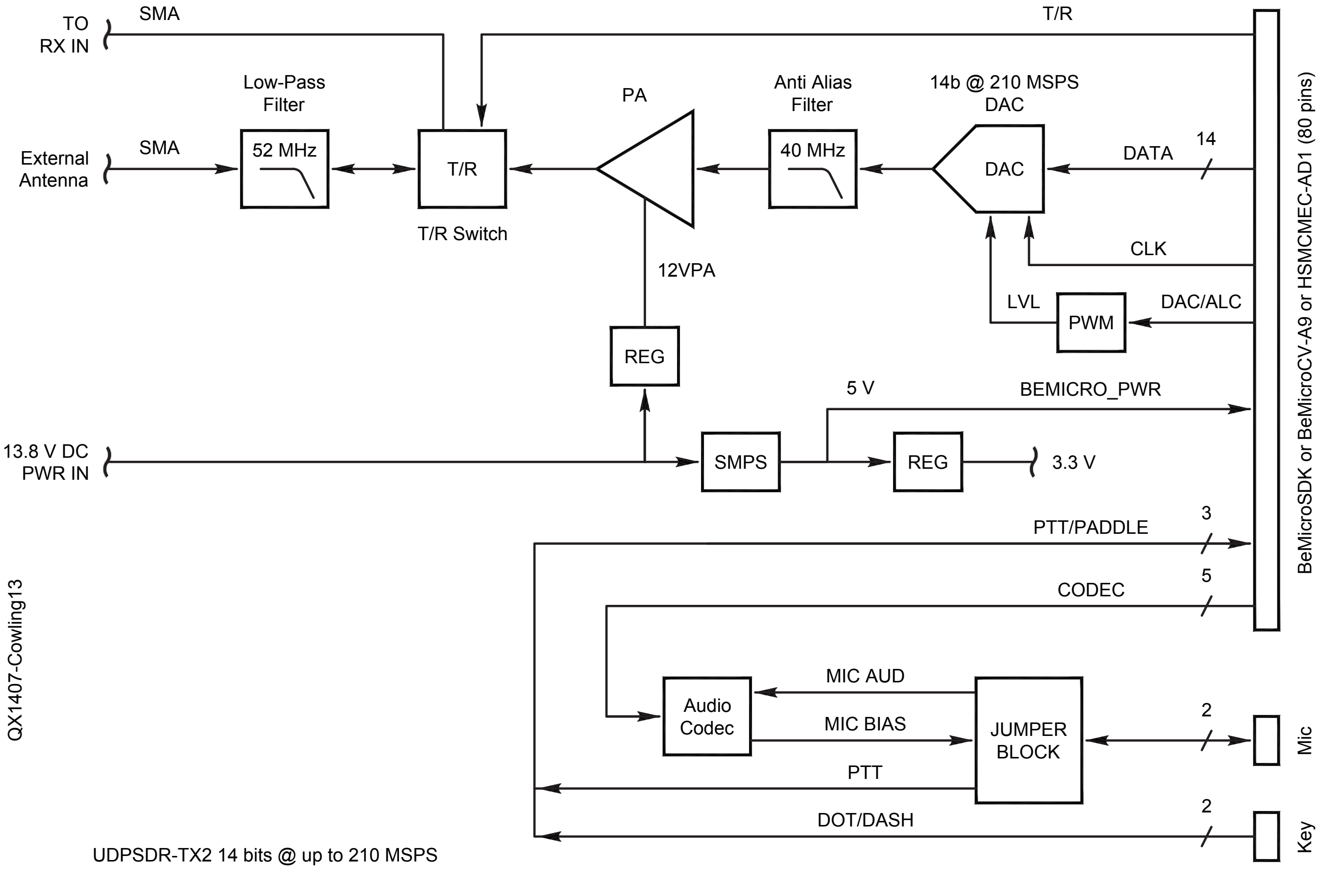

Figure 13 — TX2 transmitter block diagram.

AD9744ARU 14 bit, 210 mega samples per second differential-output-current DAC. A two-stage transformer-coupled differential PA supplies 500 mW to the output connector through a 52 MHz low pass filter and a PIN-diode T/R switch. An on-board +5 V switch mode power supply accepts a wide range DC input (9 V to 18 V) and provides enough current to power an HF2 receiver and either a BeMicroSDK or BeMicroCV-A9 data engine. The TX2 sells for $179 from iQuadLabs.[10]

Amplifiers

The QRPp level of 500 mW may be enough some of the time, but there are other times when you need to trade in your slippers for a pair of boots (speaking in the RF sense, of course). The two options that I describe here are by no means the only viable ones. The first QRO solution is the Hardrock-50 HF power amplifier kit.[11] This kit (Figure 14) is a 5 W input, 50 W output 160 m through 6 m amplifier, but you can add a 10 dB gain, 5 W pre-driver board to make it a perfect match for the 500 mW output of the TX2 transmitter. There is an internal auto tuner in the works, and harmonic filtering is

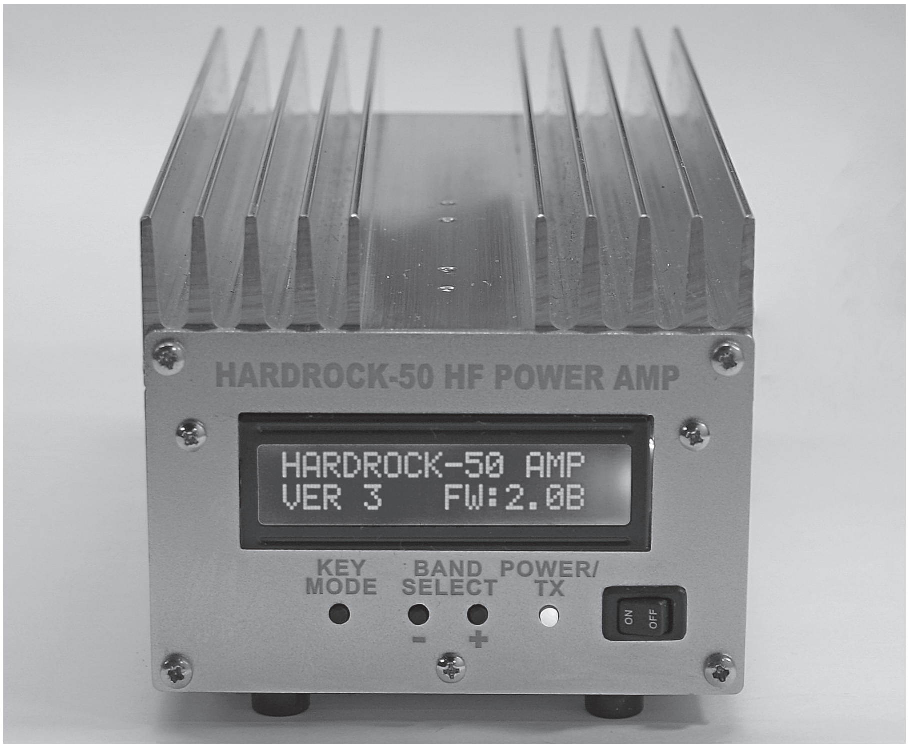

Figure 14 — The Hardrock-50 HF power amplifier kit covers 160 m through 6 m.

already designed into the amplifier. The TX2 transmitter has an amplifier keying output that can be used to switch the Hardrock-50, or it can be done via RF detection (carrier operated).

TAPR offers the Pennywhistle 18 W power amplifier as a kit and the Alex TX low-pass filter board assembled. Pennywhistle (Figure 15) will produce 16 W to 20 W of power from 500 mW of drive from 160 m through 6 m.[12] Pennywhistle requires harmonic filtering at its output, which can be provided by the Alex TX low-pass filter board (Figure 16).[13] The TX2 amplifier keying output can be used to control Pennywhistle, but there is no control port to control the Alex filter selection. An external manual controller or an interface to a computer or the data engine must be built to perform this function. Both the SoCkit and the BeMicroCV-A9 have extra I/O pins that can be used for this purpose; the BeMicroSDK does not. If you are clever, you can figure out a way to re-purpose unused LED or switch I/O pins on the BeMicroSDK, since you only need two outputs to control Alex-TX.

Systems

Making an SDR

Now that we have discussed all of the pieces, let's assemble them into an SDR. I will examine three SDR configurations, each built from the components described above. To do this, we must focus not on the individual blocks, but on section interconnections, shown as horizontal lines between the three sections shown in Figure 1.

In all three example SDRs, the physical connection between the user interface and the data engine is Ethernet cable. Physical connection is easy, but to get the user interface and the data engine to *understand* each other over this connection requires a bit more effort. To make this work, we will use a standard protocol over the wire called User Datagram Protocol (UDP). A datagram is made up of a header (which contains routing information and a checksum, among other things) and a block of user data. It is up to the originator of each datagram (the source) to place the bytes of user data into the datagram

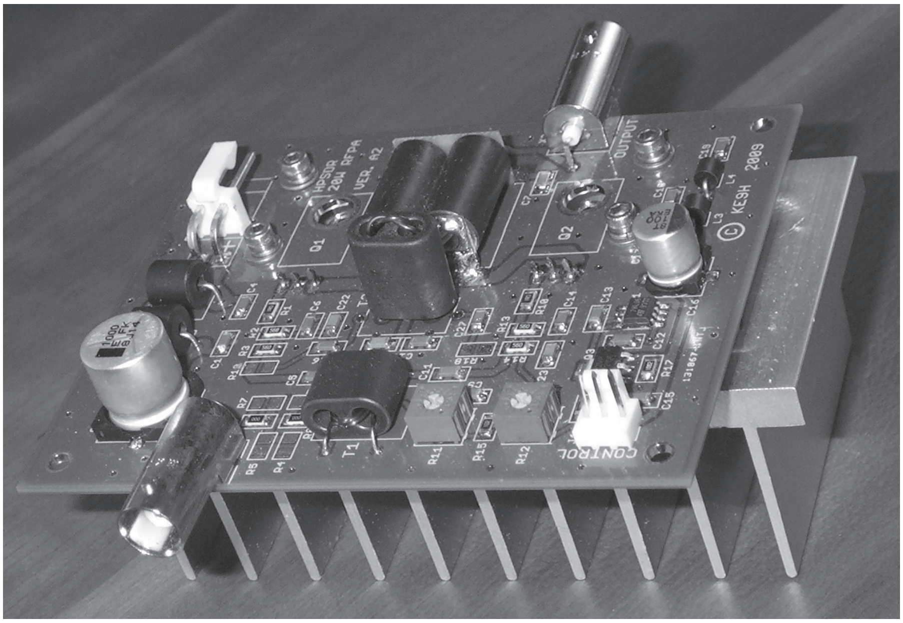

Figure 15 — The Pennywhistle power amplifier requires only 500 mW of drive for 18 W output.

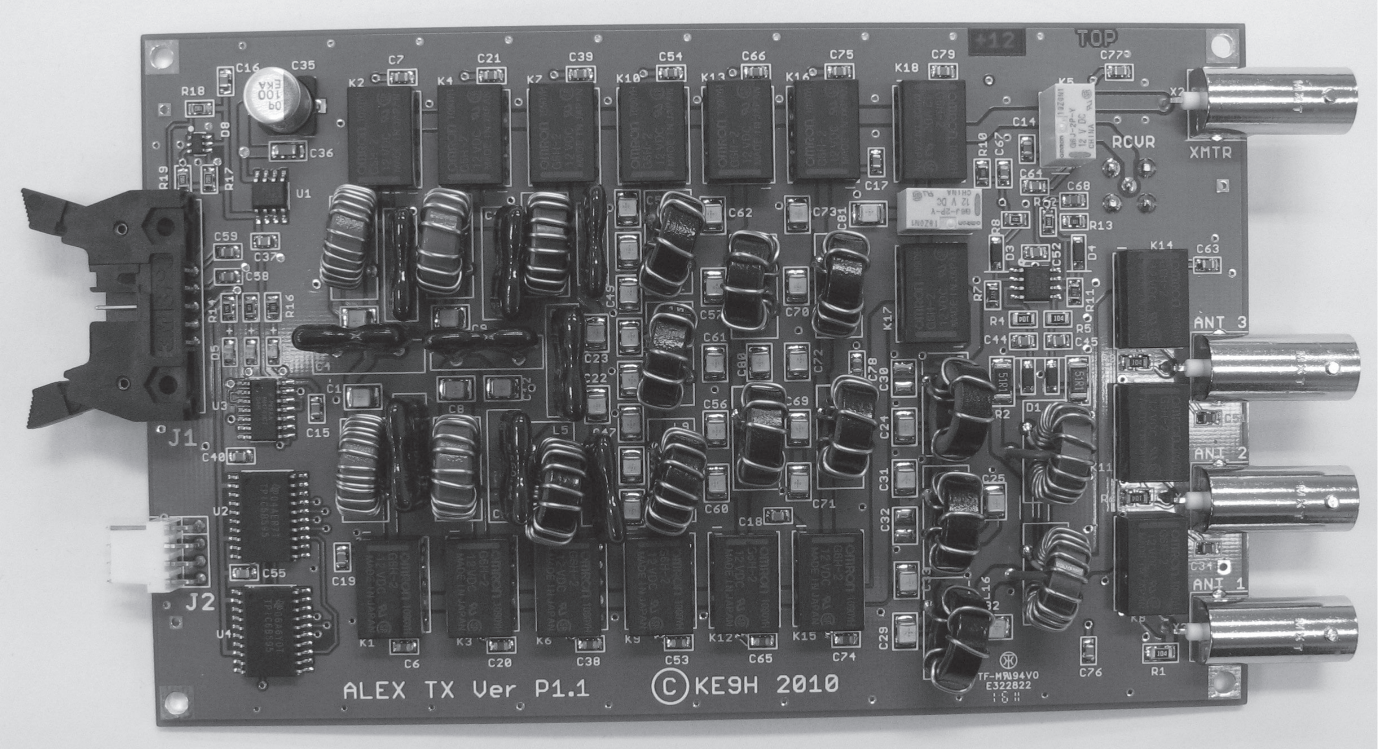

Figure 16 — The Alex transmit low-pass filter board handles harmonic suppression.

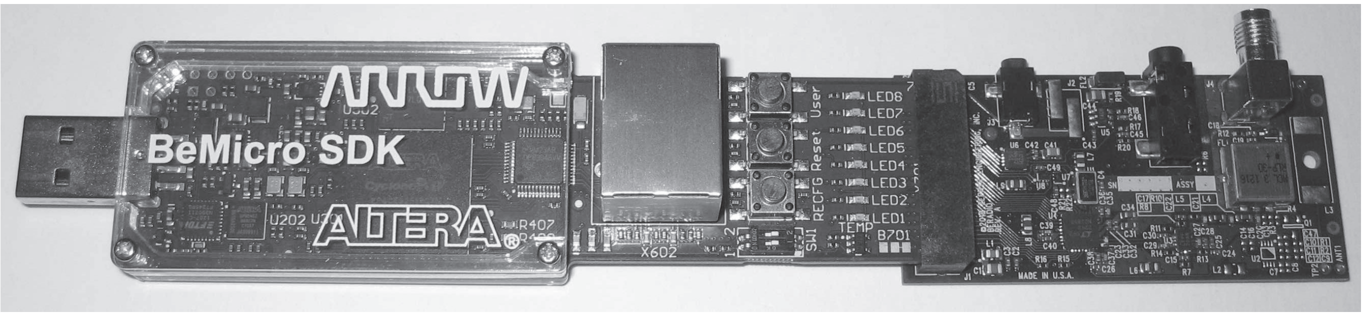

Figure 17 — A complete cost-effective SDR receiver: BeMicroSDK with HF1.

Table 1
Comparison of the SDRstick^{TM} HF1 and HF2 Receiver Boards

	HF1	*HF2*
Receive frequency range	100 kHz to 30 MHz	100 kHz to 55 MHz
Antenna connector	SMA	SMA
Front-end attenuator	--	0-31dB, 1dB steps
LPF	RLP-30+ and anti-alias	RLP-40+ and anti-alias
Pre-ADC gain	20 dB	20 dB
ADC	LTC2249, 14-bit at 80 Msps	LTC2208, 16-bit at 122.88 Msps
ADC clock source	TCXO or FPGA	Crystek CVHD-950 VCXO
Receive audio	LTC2641 Audio DAC	TLV320AIC23B CODEC
GPSDO reference input	--	10 MHz
Transmitter expansion micro-edge connector	--	yes
Cost[10]	$169	$399

in the order that the receiver of the datagram (the destination) expects. For this to work, the designers of the user interface and the designers of the data engine agree to organize the data within the block of user data in a pre-defined order. Just to confuse things even more, this definition is also called a protocol. For example, the openHPSDR version of PowerSDR^{TM} uses openHPSDR Ethernet Protocol formatted data blocks. The upshot of all of this is that the data engine must build UDP packets that the user interface can understand. There are many different user interfaces and many different data engines, so how can this possibly work? As Paul Simon said, "would you explain about the 50 ways…"[14] Let's look at *three* ways.

The ExtIO.dll Method

The High Definition Software Defined Radio (HDSDR) software user interface is a good example of a straightforward way for a single user interface to talk to many different data engines.[15] The designers of HDSDR defined their data block format and wrote their user interface to recognize it. They wrote their code to accept an extension program called ExtIO.dll. This extension sits logically in between their user interface and the data engine and reformats data blocks from the data engine into standard HDSDR format data blocks. Each SDR designer writes a simple ExtIO.dll to convert his data format to HDSDR data format (and back) and includes this software with the SDR. The SDRstick^{TM} radios come with an ExtIO.dll for HDSDR.

Customize Your Data Engine

Another way for one data engine to talk to many different user interfaces is to just program it that way. The "P" in FPGA stands for "programmable," so just program the data engine to format the data blocks for whatever user interface you want to use today. If the FPGA in the data engine has enough resources (like the BeMicroCV-A9 has, for example), program it to recognize the user interface and automatically format the data for that particular program. The SDRstick^{TM} does this, too! When HDSDR or PowerSDR^{TM} user interfaces are started (other user interfaces do this, too), they broadcast a special packet to the network called a Discovery Packet. SDR data engines are designed to reply to this Discovery Packet by returning their network address and radio ID back to the user interface. Since the Discovery Packet also identifies the user interface, (and thus, the data format type that it understands), the data engine can use this information to decide what format to send back. For example, if PowerSDR^{TM} sends a Discovery Packet to an SDRstick^{TM} data engine, SDRstick^{TM} responds with openHPSDR Ethernet format packets. If HDSDR sends a Discovery Packet to an SDRstick^{TM} data engine, SDRstick^{TM} responds with its native format packets, which are then converted to HDSDR native format by the SDRstick^{TM} ExtIO.dll converter.

Customize Your user interface

A third way to hook the data engine and user interface together is to write your own user interface. While this may seem like a difficult solution (and it is not trivial), software can come to the rescue. The GNU Radio project might be the solution that you are looking for. For a GNU Radio receiver, the data engine formats data in its native mode. The SDR designer writes some software called a GNU Radio Source Block that converts the native mode data format into a standard GNU Radio format. Once the data is in this format, it can be used anywhere within GNU Radio to build custom radio software. (In the transmit direction, the interface is called a Sink Block.) GNU Radio has a learning curve to it, but it is an extremely powerful tool with which to build custom SDR user interfaces and applications. SDRstick^{TM} radios come with GNU Radio Source and Sink Blocks. Please refer to "Digital Signal Processing and GNU Radio Companion" by John Petrich, W7FU, and Tom McDermott, N5EG, in this issue of *QEX* for Part 1 of an in-depth look at GNU Radio.[16]

Data Engine to RF Hardware

Connecting the RF front ends to the data engine is simple, since the boards are designed to plug together. A brief description of the interface follows. The HF1/HF2 receiver interface from the ADC is made up of parallel data, an ADC clock and an overflow bit. The TX2 transmitter interface to the DAC is also parallel data and a DAC clock. The CODEC (for receive audio) and the RF step attenuator on the HF1/HF2 receivers and the CODEC (for microphone audio) on the TX2 transmitter are each digitally interfaced to the data engine. The data engine uses several general purpose input/output (GPIO) pins for things like PTT, paddle dot and dash inputs and ADC and clock buffer control. The HF1 (Figure 9),

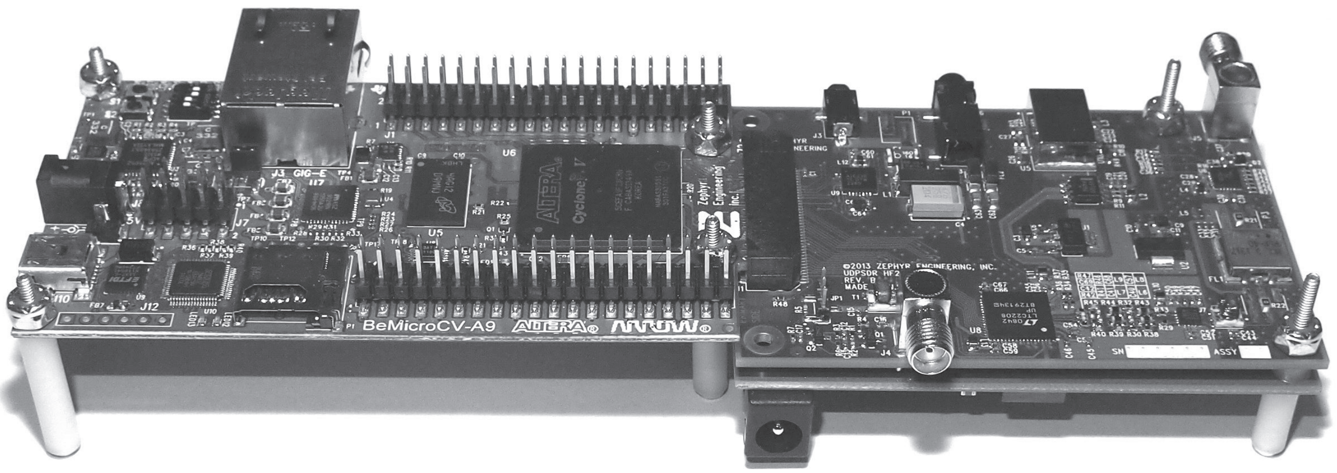

Figure 18 — A complete high-performance SDR transceiver: BeMicroCV-A9 with HF2/TX2.

HF2 (Figure 11) and TX2 (Figure 13) block diagrams show the connections.

Receiver with BeMicroSDK and HF1

The most cost effective SDR platform is the receive-only BeMicroSDK data engine paired with the HF1 receiver (Figure 17). This creates an Ethernet-based receiver that covers 100 kHz to 30 MHz with 1.25 MHz receive bandwidth. This is wide enough to see the entire MW broadcast band on-screen in the panadapter display. It is also wide enough to fully display any of the Amateur Radio bands below 28 MHz, or enough of the 10 m band to display all of the activity on any one mode. This receiver costs just over $200, and is a good place to get your feet wet in FPGA-based SDR. This is by far the least expensive Ethernet-based, broadband SDR available. The FPGA program available for BeMicroSDK/HF1 does native UDP format at 1.25 MHz and 384 kHz receive bandwidths and Hermes UDP format at 384 kHz bandwidth, both receive only.

Transceiver with BeMicroCV-A9, HF2 and TX2

The most exciting SDR platform is the High-performance HF2 receiver married up with the TX2 transmitter and the BeMicroCV-A9 data engine (Figure 18). Truly remarkable SDR functionality becomes possible with the sheer amount of logic present in the A9 data engine's FPGA and the Gigabit Ethernet interface to move data. To put this into perspective, the A9 data engine contains nearly eight times as many logic elements as the Hermes FPGA. While Hermes is limited to about seven virtual receivers, the A9 data engine has no such limitation. (See the "What is a Virtual Receiver?" sidebar.)

The current FPGA program available for BeMicroCV-A9/HF2/TX2 does native UDP format at 1.92 MHz and 384 kHz receive bandwidths, and Hermes UDP format at

MCU Versus FPGA

An embedded microcontroller (MCU) consists of many logic building blocks that are each designed to perform one function. For example, it has an Arithmetic Logic Unit (ALU), which carries out the mathematical operations specified by the instructions in the computer software. It has a Program Counter that keeps track of where in memory the current instruction is located. It has a Memory Management Unit (MMU) that controls accesses to main memory. There are many of these blocks, and each block is a collection of logic gates, memory cells, and transistor switches, each hard-wired to perform one function, and only that function. These small blocks are wired up into a large structure in order to make a functional MCU. This is, of course, a simplistic explanation of how millions of transistors are wired up to form a microcontroller, but it illustrates one main point. The MCU logic and interconnections between these pieces of logic are fixed. The hardware is designed to fetch an instruction and carry it out, fetch the next instruction and carry it out, repeating this process forever. Modern MCUs do this job *very* fast, but they can only perform the operations hard coded into their fixed instruction set. For example, an MCU might have instructions for addition, subtraction or writing to main memory, but it will not have an instruction to perform every complex mathematical operation that might be needed. The programmer writes software to break down these custom, complex mathematical operations into small sequential steps that each can be performed by a pre-defined instruction from the MCU's instruction set.

An FPGA, on the other hand, has very little fixed logic and interconnections. To illustrate this concept, let's imagine that we can take all of the gates and memory elements (small groups of these are called logic elements or LEs) that make up the MCU, disconnect them from each other and spread them out in a "sea of logic elements." If we provide a way to connect these logic elements together in any order we like (in other words, program the FPGA), we can create just about any function we need. In fact, we can connect them back up just the way they were connected in the MCU, and we have (guess what?): an MCU! This is what is called a soft-core processor. One FPGA manufacturer — Altera — has a pre-programmed soft-core processor called NIOS II, but it is not the only one that we can make out of our sea of gates. A soft-core processor is not as efficient as an MCU, since all the logic interconnections take up space on the FPGA chip, making it bigger, and thus, more expensive to make. All the programmable logic interconnects also slow the soft-core processor down because they introduce more delay than the fixed logic interconnects of the MCU.

Soft-core processors are interesting and useful, but they are not the main attraction of FPGAs. Remember that MCUs execute instructions serially? FPGAs can perform their logic functions in parallel. Imagine that I need to perform 10 additions. Even with in-line coding (no loop), it will take the MCU 10 instructions to do this, and more if the 20 addends must be fetched from memory first. If the MCU runs at 100 MHz (10 ns per clock cycle), and we assume that each instruction takes one clock cycle, it will take at least 100 ns to perform the 10 additions. If I program 10 adders into the FPGA, I can perform all 10 additions at the same time, requiring only one clock cycle to obtain all 10 sums. This is a simplistic example, but consider that even small FPGAs have tens of thousands of logic elements, and logic elements number in the millions in large FPGAs. FPGA hardware parallelism creates remarkable capability to implement algorithms that can benefit from this parallelism.

384 kHz receive bandwidth. This only hints at what is possible. For instance, eleven virtual receivers can be designed into the FPGA logic, one for each of the Amateur bands, 160 m through 6 m. The full spectrum of every HF Amateur band (and the lowest VHF one, too!) can be simultaneously displayed. Using appropriate software, eleven users could connect to this radio, and each user would have a virtual receiver. Admittedly, such FPGA code and software does not yet exist. Now that hardware is available to support such features, however, the firmware and software are possible. The BeMicroCV-A9 is a prototype now, but should be available by the time you read this.

Transceiver with SoCkit, AD1, HF2 and TX2

The most flexible SDR platform replaces the BeMicroCV-A9 with the SoCkit development board and the AD1 adapter (Figure 6). The SoCkit board FPGA contains fewer logic elements than the A9 board (110 K versus 301 K), but it has something the A9 does not: a dual-core ARM processor. With this processor (and the other SoCkit on-board resources), we can run an embedded operating system, such as *Linux*. *Linux* brings with it things like a full TCP/IP stack, software control of packet data formatting and easy application development (compared to FPGA applications), among other things. The current FPGA program available for SoCkit/AD1/HF2/TX2 does native UDP format at 1.92 MHz and 384 kHz bandwidths, and Hermes UDP format at 384 kHz bandwidth.

The same virtual receiver scenario is possible with the SoCkit data engine that is possible with the A9, but other possibilities open up with the addition of *Linux* to the system. For example, we could write a server application to serve data up directly to remote clients. We can run this application right on the SoCkit board's local processor, eliminating the computer normally necessary to perform this task. We have made an "NAR," or Network Attached Radio! While I have just coined this term, you can bet that the concept is already here!

Conclusion

Advanced, high-performance hardware is available off-the shelf at reasonable cost. FPGA code is currently available to perform basic functions, while more advanced features are either planned or left to the user to implement. Some open-source FPGA example code is available, and can be used as a starting point for developers and experimenters.[17] There are lots of SDR user interfaces to choose from, many under current development and some are open source.

Scotty Cowling, WA2DFI, was first licensed in 1967 as WN2DFI, and has been continuously active since that time. An Extra Class licensee and ARRL Life Member, Scotty is active while mobile on HF CW and on APRS. He is an advisor for Explorer Post 599, a BSA affiliated ham club for teens in the Phoenix, Arizona area. He also enjoys minimalist QRP operating. He has participated in every ARRL Field Day since 1968!

Scotty has been involved in the openHPSDR project for the last 8 years, and has served on the TAPR Board of Directors (2006-2012) and as TAPR Vice President (2011-2012). He is active in the production of openHPSDR components and with other TAPR projects. He is a co-founder of iQuadLabs, LLC, a supplier of openHPSDR systems and other Software Defined Radio components, and is President of Zephyr Engineering, Inc, an engineering consulting firm.

Scotty's professional specialty is FPGA and embedded systems hardware design. He designed his first project with a microprocessor in 1975 and his first FPGA project was in 1987. He holds a BSEE from Rensselaer Polytechnic Institute and an MSEE from Arizona State University.

Notes

[1] Steven W Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, ISBN 0966017633, available for free download at: **dspguide.com**.

[2] Many DSP references can be found here: **dspguru.com/dsp/links/books/online**.

[3] Arrow Electronics BeMicroSDK information: **arrownac.com/solutions/bemicro-sdk**.

[4] Arrow Electronics BeMicroCV information: **parts.arrow.com/item/detail/arrow-development-tools/bemicrocv**.

[5] Arrow Electronics SoCkit board information: **arrownac.com/solutions/sockit**.

[6] Comedians on the TV show *Rowan and Martin's Laugh-In* used the term "sock it to me," typically followed by a dousing with a bucket of water. See **en.wikipedia.org/wiki/Rowan_&_Martin%27s_Laugh-In**.

[7] SoCkit ordering information: **parts.arrow.com/item/search/#st=sockit;renMeR**.

[8] Scotty Cowling. WA2DFI, "The High Performance Software Defined Radio Project," *QEX*, May/June 2014, pp 3-13.

[9] Hermes SDR information: **openhpsdr.org/wiki/index.php?title=HERMES**.

[10] HF1, HF2, TX2 and AD1 boards may be purchased from **iQuadlabs.com**.

[11] Hardrock-50 amplifier information: **hobbypcb.com**.

[12] TAPR Pennywhistle kits: **tapr.org/kits_pw**.

[13] TAPR Alex Filter boards: **tapr.org/kits_alex**.

[14] Paul Simon, 1975: **en.wikipedia.org/wiki/50_Ways_to_Leave_Your_Lover**.

[15] HDSDR web page is: **hdsdr.de**.

[16] John Petrich, W7FU, and Tom McDermott, N5EG, "Digital Signal Processing and GNU Radio Companion: An Easy Way to Include DSP in Your Radio Projects," Part 1, *QEX*, Jul/Aug, Part 2, *QEX*, Sep/Oct 2014.

[17] Hermes FPGA code is open-source: **svn.tapr.org**.