Sivan Toledo, 4X6IZ

School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel, stoledo@tau.ac.il

A Driverless Ethernet Sound Card

*Sivan describes a high performance Ethernet interface to a
PC that is portable across operating systems.*

A growing class of radios relies on a laptop or desktop computer to perform signal processing and to implement the radio's user interface. Examples include Flex Radio's transceivers (starting from the SDR-1000[1] and now including the Flex-3000 and Flex-5000), the Softrock kits[2], and others. In all of these radios, which I will refer to as software-defined radios (SDRs[3]), a radio front-end is connected to a generic personal computer (PC). This connection carries a complex baseband signal, radio status information, and commands to the radio.

Many radios in this class transfer analog baseband signals to/from the PC. The PC digitizes the incoming signal and produces the outgoing signal using a sound card, either built-in or external. The status and command communication in early radios used the serial or parallel port of the PC, but recent radios tend to use a USB interface. The overall architecture is shown in Figure 1.

[It is the consensus of the author and our technical reviewers that Hertz is an improper unit to describe sample rate. There is no SI unit for sample rate, so we have to choose something appropriate. [We will use "SPS" to mean samples per second.—Ed]

Some PC sound cards have high dynamic range and moderate sampling rates (up to 192kSPS). Therefore, using a sound card is a reasonable low-cost solution, but it has some disadvantages. One possible disadvantage is the difficulty of finding high-quality reasonably-priced sound cards. The archives of SDR user groups' mailing lists suggest that finding a suitable sound card is challenging for many. Another disadvantage of using a consumer sound card is the possibility of rapid obsolescence. Cards that come with a special driver, which is the case with many high-end cards, only work on operating sys-
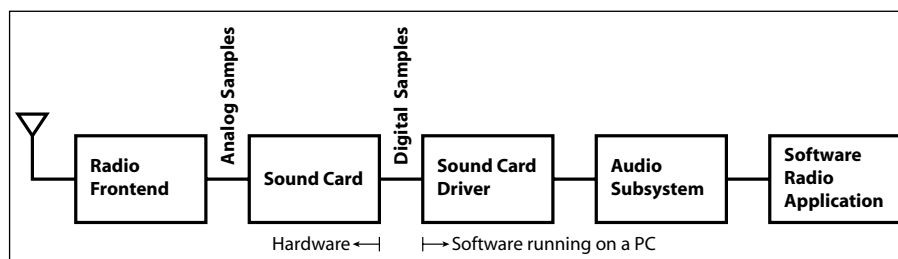

Figure 1 – General SDR architecture when using a PC Sound card.

tems for which the vendor supplies drivers. A sound card that works fine under *Windows XP*, for example, may not be supported by the vendor on *Windows 7*, or it may not be supported on *Linux*. The third disadvantage of relying on sound cards is their limited bandwidth, at most 192kSPS.

To address these issues, some recent radios include a sound card or non-audio analog-to-digital and digital-to-analog converters (ADCs and DACs). Such radios include some Flex Radio products[4], which include a built-in sound card, the HPSDR project, and the SDR-Widget project (which aims to build a low-cost high-performance sound card for softrock-type radios), and others. Radio front-ends that sample faster than 192kSPS also include ADCs and DACs; these include the USRP radios, the radios built by Pieter-Tjerk de Boer (PA3FWM), and others[5,6].

This article presents a novel way to communicate baseband samples to/from a PC. More specifically, I will describe a sound-card prototype that uses the most ubiquitous Internet communication protocol, TCP/IP, to communicate these samples. This approach has significant advantages over the common approach which uses a USB connection.

From USB to Ethernet

Most external sound cards use a USB connection to transfer audio samples to/from the PC. These cards usually use the standard USB audio protocol[7], but they can also use custom USB protocols. Most operating systems support the USB audio protocol. Therefore, cards that use this protocol do not need a card-specific driver, and they work with all operating systems. The USB audio protocol uses a USB data transfer mode called *isochronous transfer*[8]. The isochronous mode allows the PC to choose a sampling rate for which it can ensure sufficient bandwidth on the USB bus. The PC dedicates a fixed fraction of each USB frame to the audio data transfer. Isochronous transfers are not reliable: missing or corrupt packets are not retransmitted. This is a reasonable and common design decision for real-time data. Sound cards that use a specialized driver can also use isochronous data transfers.

USB sound cards suffer from several problems. One problem is limited support for high sampling-rates and for 24-bit samples. Release 1.0 of the USB Audio standard did not support high-end cards, which forces high-end cards to use card-specific drivers. This problem may disappear over time, since release 2.0 of the standard allows higher sampling rates and 24-bit samples, but operating system support for this version is still patchy. A second problem is

that the USB driver transfers the samples to/from the operating system's audio subsystem. The software interfaces (APIs) that the audio subsystem presents to applications vary widely between different operating systems. Some operating systems have multiple audio interfaces with different capabilities (e.g., DirectSound and ASIO in *Windows*). This makes it difficult to write portable SDR applications. (There is a good reason for this complexity: multimedia applications like games and video editing need low latency control of audio streams, but this is less important for SDR applications.)

An Ethernet connection to the sound card solves these problems. If the sound card sends and receives samples using UDP or TCP, it communicates directly with the SDR application without passing through the operating system's audio subsystem. The data still passes through device drivers on the PC, but now these are the networking drivers, which are built into all operating systems. Furthermore, the software interfaces to networking functions are essentially the same across all operating systems, unlike the interfaces to the audio subsystem. This makes it easy to write portable SDR applications.

The Ethernet frames carrying the baseband samples can be transported on a direct Ethernet cable connecting the sound card and the PC or through an Ethernet switch. They can also be transported though a USB connection, using a standard Ethernet-over-USB protocol; *Windows*, *Linux*, and *MacOS* come with built-in support for USB Ethernet cards, so no card-specific driver is needed. The possible interconnect architecture is shown in Figure 2.

The sound card itself contains digital-to-analog (DAC) and analog-to-digital (ADC) chips, or a chip with both (usually referred to as a CODEC) and a microcontroller. Low-end USB sound cards use a single special-purpose chip that serves both as a CODEC and as a USB interface, but in this article the focus is on higher performance designs with a microcontroller. Until recently, microcontrollers with a USB interface were significantly cheaper than microcontrollers with an Ethernet interface, and there was a wider selection of the former. Today, however, there are reasonably-low cost microcontrollers with a complete 100Mb/s Ethernet interface, including the physical signaling. The main external components required are the Ethernet socket and the isolation transformer, which can be built into the socket. The prototype described in this article uses such a microcontroller, the Texas Instruments LM3S9B96. This microcontroller also has an interface to audio DACs, ADCs and CODECs, making it suitable for sound cards.
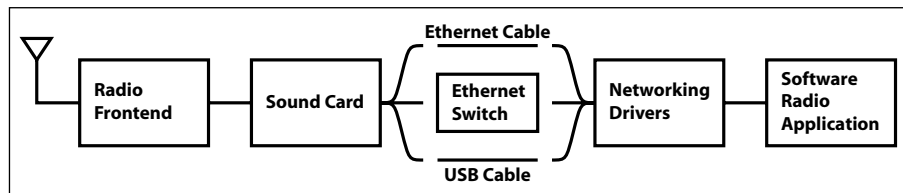

Figure 2 – SDR architecture when using Ethernet connection to the PC.

The Prototype: Hardware and Software

The Ethernet soundcard prototype is built around a simple microcontroller evaluation kit called EK-LM3S9B96 (Figure 4). [The EK-LM3S9B96 was a limited edition kit that is no longer available. The EK-LM3S9B92 uses an MCU with the exact peripherals and capabilities of the LM3S9B96 but does not include SafeRTOS or IEEE1588 in its Flash ROM—Ed.] The board contains the microcontroller and its oscillator crystals, a voltage regulator, Ethernet and USB sockets (I did not use the USB connection), and programming and serial-port connections to a small interface board. The interface board connects to a PC through a USB cable, allowing the PC to program the microcontroller and allowing the microcontroller to send back diagnostic or debugging information.

The evaluation board makes some of the microcontroller pins available through 0.1" headers, including all the CODEC interface pins. I mounted the evaluation-kit board on a large prototyping board that also had space for the CODEC, a Texas Instruments TLV320AIC23B. The line-in port and headphones output ports of the CODEC are connected to standard ¼" stereo jacks.

The software on the microcontroller is based on example code that came with the microcontroller. The networking software I used is *lwIP* (light weight IP), a small and free implementation of IP, TCP, and UDP. *lwIP* provides several programming interfaces. I used the raw interface, which is the lowest level interface, since it provides maximum performance and control.

I wrote the PC software in Java. I tested it on *Windows XP*, but it should run unmodified on *Linux*, too. The rendezvous or enumeration mechanism that allows the PC application and the sound card to find each other is described later in this article.

Overcoming Scheduling Jitter

The main challenge that I faced in this project was to drive the sound card's DAC. The common wisdom is that UDP is the appropriate Ethernet protocol for transporting real-time data, such as audio[9]. UDP is an IP protocol that is used to send so-called *datagrams*, or discrete limited-length packets of data between applications. UDP is not

reliable; the protocol itself does not acknowledge packets and does not retransmit lost or corrupt packets, although applications can implement acknowledgements and retransmission for important data.

In spite of the common wisdom, it turns out that UDP does not work well when one side in the communication channel is a microcontroller with a limited amount of RAM. The problem is not with receiving data from the sound card, but with sending data to it; the problem turns out to be more severe under *Windows* than under *Linux*.

To understand the problem, let's consider what happens when the sound card sends and receives baseband samples using UDP packets. We will assume that the PC processes incoming baseband samples into audio that it sends to an internal sound card, and that it sends baseband samples that are derived from its internal sound card's microphone input.

Let's start with the direction that does work well, sending data from the sound card. The microcontroller receives a baseband sample from the ADC or CODEC every few microseconds and puts it automatically in a hardware queue. When the queue reaches a certain level, it generates an interrupt signal that causes the microcontroller to remove samples from the queue and place them in a buffer in RAM. When this buffer has enough samples to be worth sending in an Ethernet frame (usually up to 1500 bytes), the microcontroller sends them to the PC in one UDP packet. A packet every millisecond is a reasonable rate; at 24-bits per sample, two sampling channels, and 192kSPS, a millisecond of audio is 1152 bytes, small enough to transport in one packet. At lower sampling rates or lower resolutions we can either send shorter packets or send less frequently. When the PC receives a packet, its network card generates an interrupt that causes the operating system to process the packet. The operating system stores the packet in a buffer until the SDR application retrieves it. This operating system buffering response happens even if the PC happens to be running some other application when the packet arrives. At some later time the PC's operating system will switch to the SDR application. When the application will reach the point where it reads UDP packets, the packet will be delivered to it.

Table 1

Comparison of Task Latency Across Scheduling Parameters

Time in ms

	Windows XP Normal Priority		Windows XP Realtime Priority		Linux Normal Priority	
	Maximum	Typical	Maximum	Typical	Maximum	Typical
Idle	146	125	126	125	43	43
One Core Busy	155	125	129	125	51	43
Two Cores Busy	--	--	129	125	64	43

The PC's operating system can buffer lots of packets. If many packets are buffered when the SDR application is scheduled to run, it will receive all of these packets, process them, and create audio for the internal sound card. The buffering and the fact that processing is delayed do not prevent the software radio from producing continuous audio; they only mean that the audio that the user hears was received some time ago, say a quarter or half a second.

Now let's consider the other direction, which is more troublesome. The PC's operating system runs the SDR application once in a while. The application reads audio from the internal sound card. The operating system buffers this audio, so it is not lost even if the SDR application is suspended for hundreds of milliseconds. The SDR application processes this audio, produces baseband samples, and sends them to our sound card in UDP packets. If the SDR application is scheduled to run only every 100ms, say, what the sound card sees is a period of almost 100ms with no data at all, and then a burst of UDP packets. A burst after 100ms will contain about eighty 1500-byte packets at 24-bit and 192kSPS.

The large, infrequent bursts cause two problems. One is the need to buffer enough baseband samples to bridge over the periods in which no data is received. At 48kSPS and 16-bit samples, the microcontroller only needs to buffer 9.6KB to be able to feed the DAC for 100ms. But at 192kSPS and 24 bits, it needs to buffer 115KB. This is possible with an external RAM chip, but usually not with the internal RAM of a microcontroller (the LM3S9B96 has 96KB of RAM). The other problem is the high likelihood of lost packets. The PC can send the packets in such a burst much faster than a relatively slow microcontroller can process them, so many of these packets are likely to be lost.

I implemented such a protocol. It worked fine at 48kSPS but not at 96kSPS. At this rate, there were many gaps in the analog baseband signal produced by the sound card.

The severity of this problem depends on how often the PC's operating system schedules the SDR application. To measure this, I wrote a small application that simply reads (and discards) audio from the internal sound card. The program reads a fixed amount of audio every time. The amount of audio ranged from 1ms worth of audio to 100ms. The program measured the time periods between successive audio-read operations. Ideally, these periods would be close to t milliseconds when the program asks for t milliseconds worth of audio. But this is almost never the case.

Table 1 shows the actual periods that I measured for t=10ms. The experiments were conducted on a Dell D820 laptop connected to ac power. The laptop has a dual-core Intel T7200 processor running at 2GHz and it ran either *Windows XP* SP3 or *Linux* 2.6.31. The test program is written in Java, so exactly the same code runs under *Linux* and *Windows*. The results for other values of t, ranging from 1ms to 40ms, were similar; the results at 60ms, 80ms, and 100ms were similar under *Windows* but showed increased latencies under *Linux*. I ran the experiment when no other application was running (the *idle* row), when another application ran one CPU-intensive thread that used one of the two available cores (CPUs), and when the other application ran two CPU-intensive threads. The maximums are over experiments lasting 100s. The typical values are derived from the number of periods greater than 100ms in *Windows* and greater than 25ms in *Linux*; these numbers were always close to 800 and 2344, respectively.

The results show several interesting facts. The first is that both operating systems schedule the application at roughly constant intervals, even when the application asks for just a small amount of audio every time. *Windows* suspends the program for about 125ms, then runs it until it extracts all the available audio. At this point the program blocks while waiting for more audio. *Windows* suspends it again for 125ms rather than until the amount of audio it asked for is available. *Linux* does the same, but with a shorter scheduling interval of about 43ms.

The other interesting fact is that *Windows* does not run the application often enough when the CPU is busy. When only one core is busy, the maximum scheduling latency for processes with a normal priority rises a bit, but the program still runs often enough to process all the audio. When both cores are busy, *Windows* does not run the program often enough to collect all the audio even though this application presents a very small CPU load. When we increase the priority of the audio program to real-time priority (using the task manager), the problem disappears. *Windows* schedules the program often enough to process all the audio with the usual latency. *Linux* does not suffer from this problem and schedules the program often enough even when the CPU is heavily loaded.

The conclusion is that to receive data over UDP, the soundcard would need to be able to cope with gaps on the order of 150ms when *Windows* sends data and around 50ms when *Linux* sends the data, and with the packet bursts at the end of such periods. Current microcontrollers cannot do this.

Audio Transport over TCP

To address this problem, I designed a TCP-based[10] transport strategy. TCP is a reliable protocol. The receiver sends acknowledgements to the transmitter. Packets that are not acknowledged are retransmitted until they do. To avoid wasting bandwidth on packets that the receiver cannot accept due to its limited buffer space, the receiver informs the transmitter how much data it is willing to receive. This amount is called the *window*. TCP contains numerous other features, mostly to avoid network congestion. TCP is widely used on the Internet. For example, Web pages are delivered using TCP.

I initially tried a straightforward strategy for receiving data from the PC. The microcontroller allocated a buffer for incoming audio and announced the size of this buffer as the initial window. As incoming packets fill the buffer, the window shrinks. As the microcontroller uses up data from the buffer to send to the CODEC (thereby making space available for more data from the PC), the window grows. The microcontroller program updated the window every millisecond.

This simple strategy worked as long as no packets were lost. When a packet from the PC was lost, the loss triggered a weird behavior in *Windows XP* that caused a significant amount of audio to be lost. A TCP sender can

sense that a single packet was lost, because subsequent packets that are not lost trigger the receiver to send acknowledgments. These acknowledgements do not acknowledge the lost packet, of course, but the packet before. The repeated acknowledgements of the same packet allow the sender to conclude that one packet was lost, and it retransmits it. For some reason, *Windows XP* retransmitted only part of this lost packet (740 bytes out of 1460), not all of it. The microcontroller acknowledged what it received, but *Windows* apparently was waiting for an acknowledgement of the entire packet. At this point *Windows* stopped sending for about 200ms. After that period, it retransmitted the entire missing packet and things got back to normal. But by this time, audio has been lost, producing a gap in the analog output of the sound card. This sequence of events happened every few seconds; it was not a rare event. I discovered why the audio was lost using *wireshark*, a free TCP tracing program.

The moral here is not that *Windows XP* is defective (it eventually did transmit all the data, and there might be an obscure but good reason for the way it behaved), or that TCP is wrong for this application. The moral is that TCP implementations include many complex behaviors that fast computers with a lot of memory can easily cope with but that microcontrollers cannot always cope with.

I decided to use TCP in a simpler way that would not excite any complex TCP behavior. The new strategy also used a large circular buffer to store incoming audio, but it only advertises to the PC a window of one packet (1460 bytes). This allows the PC to send only one packet at a time. When the packet is received, the microcontroller copies its content to its audio buffer. Every millisecond, the microcontroller examines the amount of free space in this buffer. If it is greater than 1460 bytes, it opens the TCP window to one packet, allowing the PC to send another packet. This strategy results in one of two behaviors. When the buffer is fairly empty, say initially or after a period in which the PC did not send data, the microcontroller acknowledges every packet immediately, opening the window back to one packet. This allows the PC to send another packet resulting in a fast packet-acknowledgement ping-pong like behavior. When the buffer is nearly full, which is the way it should be most of the time, the microcontroller receives a packet, acknowledges it, but keeps the window closed. The microcontroller checks the state of the buffer every millisecond; once enough space to receive another packet becomes available, the microcontroller opens the window, allowing the PC to send another packet. At 192kSPS and 24 bits, this happens on average every 1.26ms. At lower rates and resolutions, this happens less frequently. In both behaviors, the microcontroller essentially throttles the PC, allowing it to send one packet at a time. This rules out complex TCP behaviors that occur when there are multiple outstanding unacknowledged packets.

Sending baseband samples from the sound card is easier. The microcontroller reads baseband samples from the CODEC into a circular 2-packet buffer. Every millisecond, the microcontroller checks whether one of these packets is full. If it is, it sends it to the TCP driver. I configured TCP with a send buffer of about 10 packets (16KB), so no data is lost if the PC acknowledges packets rather slowly. Normally the PC acknowledges packets immediately.

Figure 3 shows a simplified diagram of this microcontroller software architecture. TCP packets are received by the Ethernet device driver (an interrupt service routine), which passes them to *lwIP*, which in turn passes them to the sound-card's TCP receive handler function. This function places the data in a large circular buffer (30KB). The data is extracted from this buffer by the CODEC's interrupt service routine. If the buffer is empty, the CODEC's interrupt routine sends zero samples to the CODEC, but this should never happen when the sound card functions normally. The same interrupt service routine also removes samples from the CODEC's input queue and stores them in a separate two-packet circular buffer. Once a millisecond, a periodic function is invoked by *lwIP*. It checks if more than a full packet has been extracted from the large buffer. If so, it updates *lwIP*'s window to allow the PC to send another packet. It also checks if there is a full packet in the small circular buffer. If so, it sends the data to *lwIP*'s TCP send buffer, which is a circular buffer maintained by *lwIP* itself. I configured it to 16KB. If *lwIP* needs to retransmit a packet, it retransmits it from this buffer.

Results

The TCP mechanism works well without dropping audio up to at least 192kSPS. The CODEC only supports 96kSPS and lower, so to test at 192kSPS, I simply replicated each sample and verified that no data is lost.

I tested the sound card by sending audio to it from the PC's microphone and by receiving PC audio that the sound card digitized from the analog output of an iPod music player. I also tested the card with complex baseband samples from a 14MHz radio front end (a *Softrock Lite*). The baseband samples were processed by an SDR application (*Rocky*) that demodulated radio signals.

I tested the system using three different Ethernet interconnects and using the Dell D820 running *Windows*. One was simply an Ethernet cable connecting the sound card and the PC. In this configuration, both the PC and the sound card used a technique called *Auto IP* to obtain IP addresses. This interconnect provides the highest possible performance and lowest possible latency.
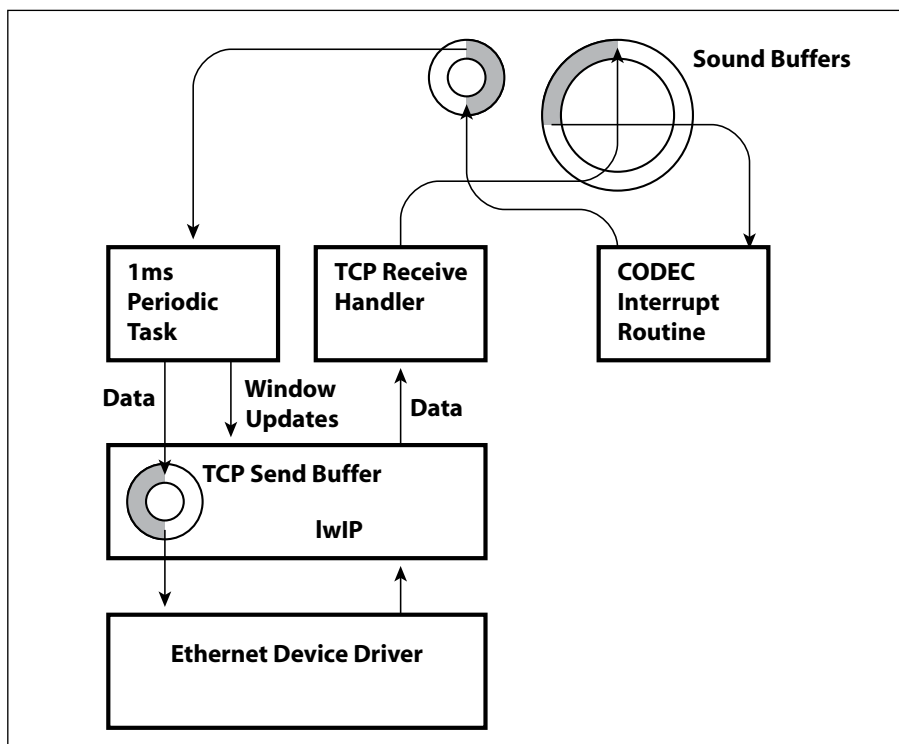

Figure 3 – Software architecture block diagram

In the second setup, the PC and sound card were both connected to an Ethernet switch. I tried both a D-Link switch designed for home use (more accurately, an ADSL modem with a built-in wired and wireless Ethernet switch, a DSL-2650U) and a large Cisco switch. In both of these setups, the sound card performed well without dropping any audio up to 192kSPS.

A third setup did not support high sampling rates. In this setup, the sound card was connected to the home switch through an Ethernet cable, but the PC was connected to the switch through a WiFi connection (802.11g, running at strong signal strength and 54Mb/s). In this setup, performance was good without dropped audio up to 48kSPS, but at 92kSPS and higher the PC-to-soundcard direction dropped a lot of audio. I eventually traced this to round-trip latency of about 2ms associated with the WiFi connection. That is, about 2ms pass between the time the PC sent a packet of data and the time the PC receives an acknowledgement with a window update allowing it to send another packet. This latency restricts the effective bandwidth between the PC and the sound card. Achieving high bandwidth in high-latency channels is exactly the reason that TCP normally uses large windows rather than the single-packet window that is used here; but as explained above, large windows, a small microcontroller, and high data rates do not work well together.

UDP Enumeration and Control

The sound card uses TCP for baseband (audio), but it uses UDP to rendezvous with the PC application and for command and control functions. I designed this UDP protocol earlier as a mechanism for PC programs to control and communicate with various technical gadgets. The protocol works not only with the sound card, but also with another microcontroller board, where the Ethernet frames are transported through a USB interconnect. The Ethernet-over-USB firmware supports both the proprietary *Windows* Ethernet-over-USB protocol, called RNDIS, and with the USB standard protocol, called CDC-ECM, which is supported by *Linux* (and by recent versions of *Windows*). In both cases, the drivers are built into the operating system.

When the sound card boots, it tries to obtain an IP address using the DHCP protocol. This succeeds in virtually any Ethernet environment except a direct cable connection. If the DHCP protocol fails, the card uses an auto-IP address. This behavior is part of *lwIP*. Now the sound card sends a broadcast UDP packet once a second announcing what kind of hardware it is. Since the packet is sent to a broadcast address, the sound card does not need to know the IP address of the PC that will control it.

The PC application listens for such packets. When it receives one, it sends back a point-to-point UDP packet acknowledging the broadcast. Once the soundcard receives this packet, it knows the IP address of the PC. It starts listening for an incoming TCP connection request from the PC.

The UDP connection remains useful for sending commands to the sound card and for receiving status information from it. In this protocol, commands are sent in both directions. The sender expects an acknowledgement for every command. A command that is not acknowledged is retransmitted until it is.

The UDP channel continues to send heartbeat messages in both directions once a second, unless there are other commands to send. If either side fails to receive these messages for a few seconds, it assumes that the other side has disconnected. In this event, the PC application reflects this on the user interface, and the sound card goes back to sending periodic broadcast messages.

Summary

When sending real-time data over UDP from a general-purpose operating system like *Windows* or *Linux*, scheduling policies cause data to be sent in bursts with fairly long delays between bursts. Microcontrollers with limited amount of buffering and processing power cannot handle this burst property. TCP transport avoids the burst issue because the microcontroller can throttle the incoming data by opening only single-packet windows. In principle, opening larger windows should also work, but it fails in practice, at least when sending from *Windows XP*, because single packet loss triggers complex TCP behaviors that sometimes generate long latencies.

The use of single-packet windows links the maximum throughput to the round-trip latency. In direct cable connection and connection through a wired Ethernet switch, the prototype supported full-duplex two-channel 192kSPS rate. But when communicating through WiFi, the round trip latency rose and the throughput dropped, allowing only up to 48kSPS. This would also limit throughput when the IP connection goes through multiple switches and routers, which also increases the latency.

From the application viewpoint, real-time data transport over TCP (or UDP when possible) reduces the dependency of SDR applications on both sound-card device drivers and the audio system's API.

Appendix: Additional Technical Details

The schematic of the CODEC circuit is shown in Figure 5. The microphone-input and line-output portions of the circuit are not wired in my prototype. The circuit was not optimized for high dynamic range and low noise; it was intended only as a platform for testing the Ethernet-based protocol.

The microcontroller is configured to generate the 12.28MHz master clock for the CODEC. The same clock frequency is used for all CODEC sampling rates. The CODEC is configured as a clock master for both the DAC and the ADC, and the microcontroller is configured as a slave in both. The CODEC therefore generates the bit clock and the left-right clock signals. Configuring the microcontroller as an audio clock slave gets around a silicon bug in revision B of the microcontroller. I also configured the audio as left-justified (rather than the more standard I2S format) to get around a related bug.

The microcontroller cannot generate a master clock at exactly 12.28MHz; it uses a fractional clock divider to get close, but there is some discrepancy between the CODEC's
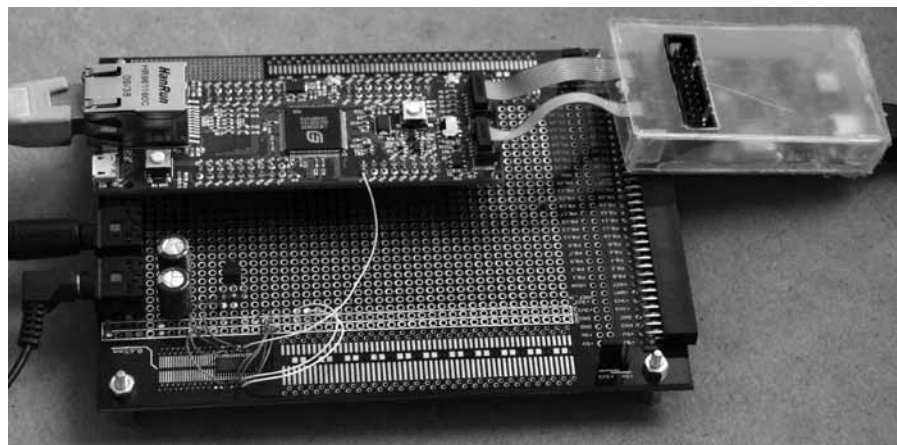
Figure 4 – Photo of the development system. The bottom board contains the CODEC parts. The board on the top left is the TI development system. The board in the plastic box is the debugging interface.

sampling rate and the rate at which the PC sends or consumes audio data. A better but more expensive solution would be to drive the CODEC with a crystal or a canned oscillator at exactly 12.28MHz.

Sivan Toledo is Professor of Computer Science at Tel-Aviv University. He holds BSc and MSc degrees from Tel-Aviv University and a PhD from the Massachusetts Institute of Technology, where he was also Visiting Associate Professor in 2007-2009. He was licensed in 1982. You can contact him at stoledo@tau.ac.il.

Notes

[1] Gerald Youngblood, AC5OG, *Software-defined radio for the masses*. QEX Jul/Aug 2002, Sep/Oct 2002, Nov/Dec 2002, and Jan/Feb 2003

[2] See **groups.yahoo.com/group/softrock40**

[3] The term SDR encompasses a wider range of radio architectures, including radios that do not use a personal computer but a special-purpose computer. In this article, however, I will use the term SDR to refer to radios that use a personal computer to do some of the signal processing

[4] **www.flex-radio.com**

[5] These receivers have a Web interface accessible from **wwwhome.cs.utwente. nl/~ptdeboer/ham/sdr**

[6] See, for example, Jim Ahlstrom, N2ADR, An all-digital SSB exciter for HF, QEX May/Jun 2008

[7] USB Implementers Forum. *USB Device Class Definition for Audio Devices*. Release 1.0, 1998, and Release 2.0, 2006

[8] See Jan Axelson, *USB Complete*, 4th edition, Lakeview Research, 2009, or John Hyde, *USB By Example*, 2nd edition, Intel Press, 2001

[9] UDP appears to be used by all or most Ethernet-based SDR platforms, such as the USRP2 (**www.ettus.com**), Jim Ahlstrom's radios, and Pieter-Tjerk de Boer's radios. The PC applications for these radios run on Linux, not Windows, which helps, as we'll see later in the article

[10] For good coverage of TCP, see W. Stevens, *TCP/IP Illustrated*, 3 volumes, Addison-Wesley, 1994

Figure 5 – Schematic of the development board and CODEC interface.
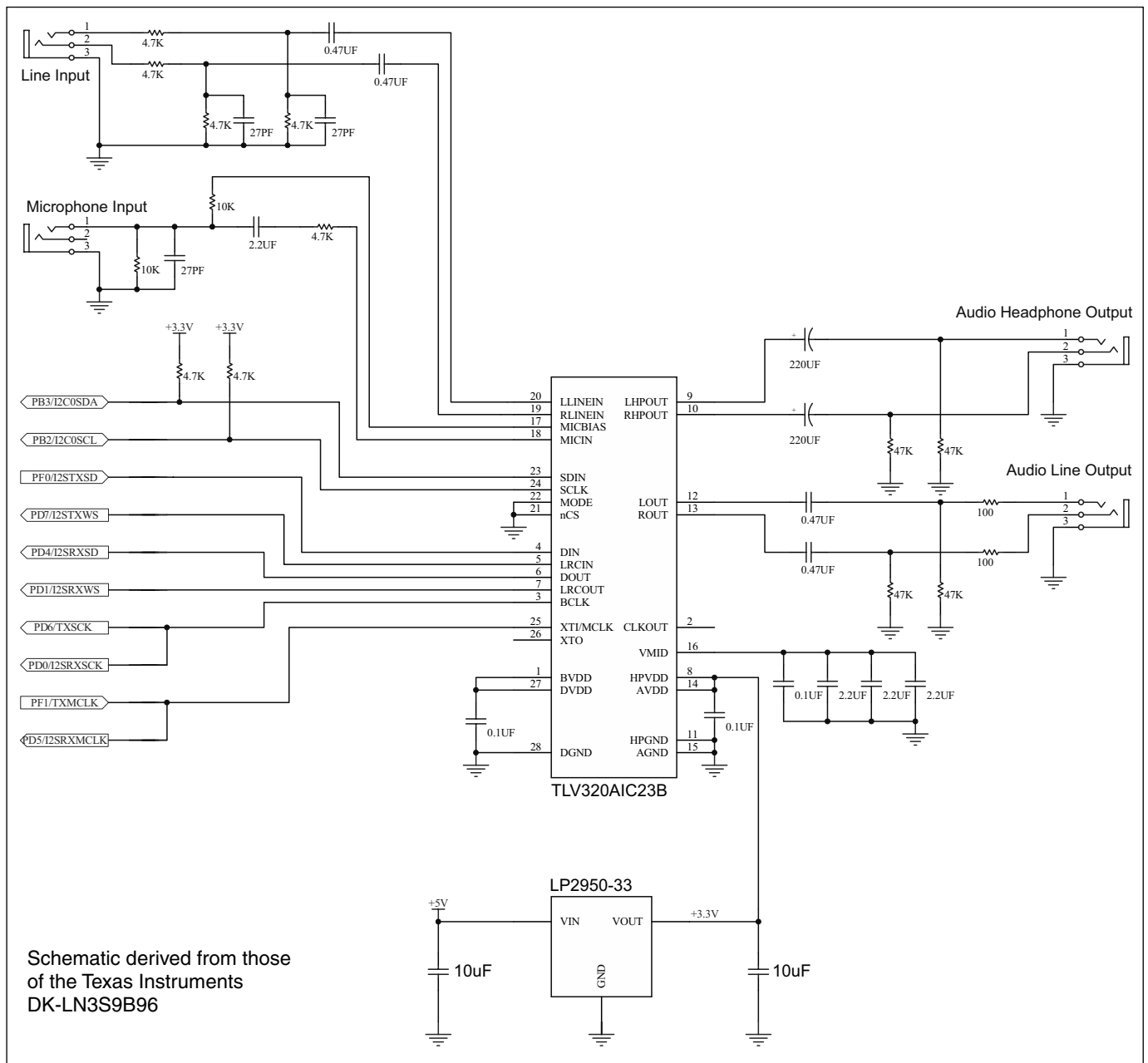
Schematic derived from those of the Texas Instruments DK-LN3S9B96