

# Software Radios for Wireless Networking

Vanu G. Bose\*

*Laboratory for Computer Science  
Massachusetts Institute of Technology*

Alok B. Shah†

*Motorola, Inc.*

Michael Ismert‡

*Laboratory for Computer Science  
Massachusetts Institute of Technology*

## Abstract

*This paper describes a novel architecture for building software wireless network interfaces. These interfaces, implemented in user-level software, run on off-the-shelf PCs and replace all of the link and many of the physical layer functions typically implemented in dedicated hardware on a network interface card (NIC). They provide all of the processing needed to transform between wideband IF signals and network packets. Moving this functionality into user-level software has several advantages. Among other things, it makes it easy to implement protocols that adapt to different applications and environmental conditions.*

*Our approach is compatible with the existing OSI protocol stack, but supports a finer granularity of layering. This finer granularity makes it possible for our NIC to dynamically change functions, such as modulation technique, that are fixed in other NICs. It also offers interfaces that facilitate interoperation with a variety of other systems, e.g., codecs.*

*We also present a brief description of our architecture which allows these software NICs to be built, as well as a sample NIC that runs on a PentiumPro, designed to interoperate with a commercial 2.4 GHz ISM band frequency hopping spread spectrum radio.*

## 1 Introduction

The recent rapid growth in wireless network technology has greatly expanded the capabilities of mobile computing devices. However, the multitude of wireless network standards hinders seamless interoperability by requiring different physical devices to interoperate with different networks. Not only do wireless LANs operate in different RF bands, but even those using the same band employ different coding, modulation, and network protocols. The implementation of network interface cards (NICs) in dedicated hardware limits the flexibility of these devices. Our approach to solving this problem is to implement as much of the processing as possible in software, allowing the NIC functionality to be dynamically modified. Our software system provides all

of the processing needed to transform between wideband intermediate-frequency (IF) signals and network packets. We have enabled the implementation of systems with the following characteristics:

- **Dynamic Flexibility:** Devices that can dynamically modify aspects of their processing such as modulation or multiple access technique.
- **Portability:** A system that can run on a variety of platforms and track the rapidly advancing technology curve.
- **Compatibility:** Applications that can interoperate with other hardware and software systems.
- **Software Reuse:** Simplify the process of developing new applications by constructing a modular environment.

Our approach does not rely on special purpose DSP processors, but rather on general purpose processors such as the Intel Pentium or DEC alpha. The rapid advances in processor clock speeds have made cycles available on the scale required to perform these real-time tasks. Despite the fact that today's general purpose processors and operating systems were not explicitly designed to handle the constraints imposed by signal processing applications, we have succeeded in designing a system for implementing computationally intensive real-time signal processing applications on such platforms.

The next section describes the software architecture, and the partitioning of the network and radio functions. Section 3 outlines our general architecture for implementing software network interfaces. An example network interface with performance results is presented in section 4.

## 2 Software Architecture

Today, wireless networks are statically specified by their built-in link and physical layer functions. In the future, we envision systems that can dynamically modify their functionality to interact with different systems and/or adapt to changing conditions. For example, a cellular base station could tailor its channel allocation and modulation scheme based on traffic and environmental conditions, and then

---

\*vanu@lcs.mit.edu

†shahalok@plhp002.comm.mot.com

‡izzy@lcs.mit.edu

indicate to each mobile unit what kind of radio to compile. However, this requires a well defined way of describing a communication system. The software radio architecture presented in this paper consists of several well-defined processing layers, which can be used to completely specify a wireless communications system.

The layering presented in section 2.1 is a refinement of the OSI layering model [Tan88], which subdivides the existing *Link* and *Physical* layers. The signal processing involved in these layers can be naturally subdivided into a finer grain model, but has traditionally been lumped into one layer because of its implementation in dedicated hardware. For our purposes, however, this is too coarse. To interoperate with different networks, it may only be necessary to change small parts of the existing layers. For example, two different systems may employ the same modulation and coding but use different multiple access protocols, or a given system may only need to change the type of coding to dynamically adapt to changing channel conditions. To facilitate this flexibility, we would like to create new network interfaces by simply combining existing functional modules, rather than by writing a new piece of software for each NIC that encompasses all of the functions in the link and/or physical layers.

While layering provides an excellent framework for modularity, it can lead to an inefficient implementation [CT90]. Since many software radio applications involve intensive data manipulation functions, the overhead of a layered implementation can be quite significant. In the short term, our engineering approach has been to group layers together where necessary, but to insure that the grouping still uses our interfaces at its edges. The penalty here is that our ability to re-use software modules is at a coarser granularity when these grouped layers are implemented. Our approach to dealing with the tradeoff between modularity and efficiency is discussed in section 2.2.

## 2.1 Processing Layers

The definition of new layers is not something to be done lightly. Too many layers would result in a cumbersome programming model, so the layering must balance this cost against the flexibility to be gained. The layers were defined according to the design principles of the OSI model [Tan88].

Figure 1 illustrates the new sublayers, and their relation to the OSI model. The function of the link layer is to take the raw transmission facility and transform it into a line that appears to be free of transmission errors to the network layer. The traditional link layer (e.g. the X.25 protocol) is preserved as the first sublayer entitled *Link Framing*, but the sublayer of channel coding is added. A channel code is designed specifically for one of three purposes: error detection, error correction or error prevention [LM94]. These are clearly link layer functions, as they are used to facilitate the

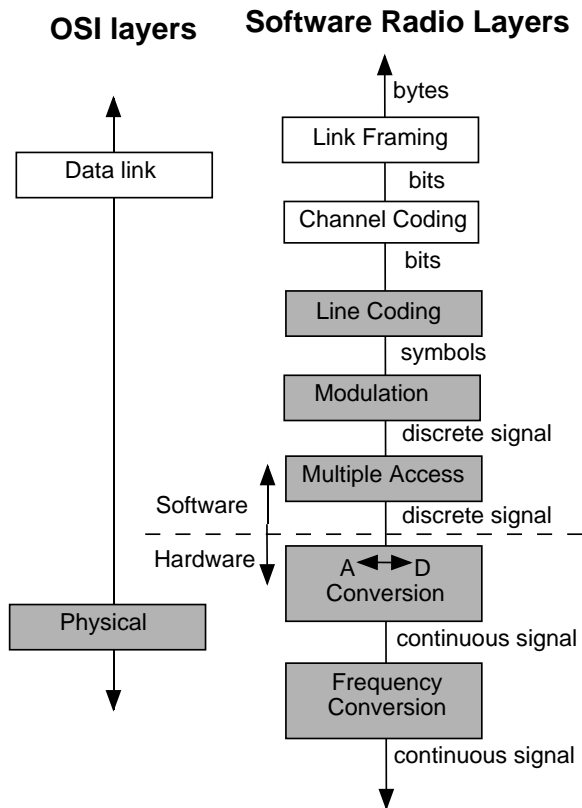


Figure 1: The Software Radio Layering Model shifts many physical layer functions into software.

appearance of transmission medium that is free from errors.

The physical layer is concerned with transmitting bits over a communication channel. Spectrum control over the physical layer can be achieved through the use of line coding, the first sublayer of the physical layer. Unlike channel codes, lines codes are not concerned with errors, but rather controlling the statistics of the data symbols, such as the removal of baseline drift or undesirable correlations in the symbol stream. The desired parameters are determined by physical characteristics of the transmission medium.

The modulation sublayer is concerned with the transformation between symbols and signals. This not only includes traditional modulation functions such as QAM, but also channel equalization functions. It is tempting to define equalization as its own layer, but this would not be appropriate for two reasons. First, it would violate the layering principle that layer  $n$  on one machine *carries on a conversation* with layer  $n$  on another machine. In general, equalization is concerned with correcting for effects imposed by the channel, not by a corresponding function on the transmitter, so there is no comparable layer on the transmission side with which to communicate. The second reason is that equalizers are an integral part of the transformation between signals and symbols, and therefore should be part of the same func-

tional layer as modulation techniques.

Finally, we have the multiple-access sublayer, which includes techniques such as TDMA and FDMA. The choice of multiple access technique is usually independent of the modulation technique, and therefore should occupy its own layer.

In general a given system may contain only a subset of the layers. However, one could think of such a system as containing all of the layers, with default functions in some of the layers that do not manipulate the data in any way. This model provides a cleaner way of thinking about a system design.

One of the goals of layering is to minimize the information flow across the layer boundaries. This is enforced by the implementation of clean interfaces. There are four different data types that exist at the interfaces of the software portion of the layered structure described above: bytes, bits, symbols and discrete signals. The data type required for each interface is indicated on the right-hand side of figure 1. The interface data structure also contains parameters relevant to the data type, such as sampling frequency and bits per sample in the case of discrete signals.

## 2.2 Integrated Layer Processing

A layered architecture provides functional modularity, but often at the cost of efficiency in the implementation. One approach is to use the layered architecture as a design tool, but to separate this model from the engineering of any particular application, where integrating layers can provide significant performance gains [CT90],[AP93]. However, we do wish to maintain some of the modularity of the layered model in the implementation, so that a given wireless NIC can be dynamically modified by changing only a small amount of code. The ability to dynamically incorporate different protocols is enabled by a layered implementation.

On the other hand, the processing involved is quite intensive, and it is often necessary to combine layers to achieved the desired latency or throughput characteristics. In particular, the layers involved with the processing of discrete signals involve many load and store operations because their data sets are typically quite large. For example, the waveform associated with a single bit in the above example requires a buffer of size  $BitPeriod * SamplingFrequency \approx 16$  samples, and each sample requires two bytes of storage. A typical IP packet containing an ICMP packet generated by the “ping” application is 64 bytes. By the time each of these bits are framed and then modulated up to the IF frequency, the waveform requires over 20K bytes of storage. In order to balance the tradeoff between flexibility and performance we follow a few guidelines for implementing integrated layer processing:

- Combine layers only if necessary to meet performance

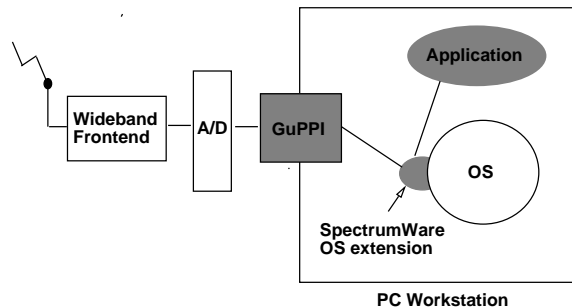


Figure 2: Block diagram of our software radio infrastructure. The hardware is used only to convert the desired RF band down to the IF frequency, digitize it and stream the samples into host memory. All subsequent processing of this digital wideband IF signal is performed in software.

requirements, and then start by combining the layers closest to the IF signal, as these will provide the largest performance gains.

- Any combination of layers must still use the defined interfaces at the edges
- Do not combine across OSI layer boundaries

We wish to leave as many layers separate as possible, so it makes sense to start by integrating the layers that require the most processing time. Using valid interfaces still allows for modularity, albeit at a coarser level, and by separating the OSI layers we leave open the ability to interoperate with other software or hardware systems that implement these layers.

## 3 Architecture

Our system architecture moves the analog/digital boundary as close to the antenna as possible and moves the software/hardware boundary right up to the wideband A/D converter. This increases flexibility by bringing more functions under software control. Since current A/D technology and available processors will not support the direct sampling of wide RF bands, our approach, as illustrated in figure 2, is to use hardware only to convert the desired RF band to the IF frequency, then directly sample the wideband IF waveform and transfer these samples into host memory. All subsequent processing is performed in user-level software.

The introduction of even this minimal amount of hardware could significantly reduce our flexibility by locking us in to a single RF band. However, multi-band frontends are becoming available which allow the software to select the center frequency and width of an RF band in the range spanning 2 MHz to 2 GHz, and sample the specified band

at a resolution of 12 bits<sup>1</sup>. Such frontends will allow for the construction of true multi-band, multi-mode software radios. The system described in this paper used a frontend that operates in the 2.5 GHz ISM band<sup>2</sup>, which allowed the implementation of all functions in software, except for RF band selection.

### 3.1 From Analog IF to Software

There are two stages in the conversion of the IF signal to a software accessible form: digitizing the signal and transporting the digital samples into the host computer's main memory. The IF signal is digitized using a 12-bit converter from Analog Devices (AD9042) that is capable of digitizing signals at a maximum rate of 40 MSPS. This provides a maximum of a 20 MHz IF bandwidth. Currently, we use either direct or bandpass sampling [Wep95] of the IF band, the next generation will take the entire RF band down to a complex baseband signal.

Wideband radio applications require that data be transferred into host memory at a very high rate. For example, to transfer a stream of 16-bit samples of a 10 MHz wide IF band (i.e. a minimum 20 MHz sampling rate) to the application, a 320 Mbits/sec data rate is required. Conventional workstations have two I/O bottle necks which had to be overcome. First, the available I/O ports on today's workstations cannot handle the required data rates. Second, the path through the operating system between a device driver and the application is inefficient [Ous90]. To overcome these limitations, we developed a PCI-based I/O system which consists of two parts. The hardware component, the GuPPI (for General Purpose PCI I/O), physically connects the analog frontend to the workstation's I/O bus. The software component, several operating system additions, provide the means for the application to efficiently access the sample streams.

The GuPPI provides the ability to burst data between the analog frontend and main memory at near the maximum I/O bus rate. In order to accommodate the jitter in the availability of resources, we use memory to temporally decouple [TB96] the sample stream. FIFOs on the GuPPI, connected to the A/D and D/A converters, decouple the timing between the fixed rate domain of the analog frontend and the variable rate I/O bus without losing any samples. This effectively absorbs any jitter caused by the bursty access to the PCI bus. These functions are performed without significant intervention from the processor; the required processing overhead per sample is less than half a cycle.

The operating system support consists of a device driver for the GuPPI and several small additions to the virtual mem-

ory system, all for the Linux kernel. The total size of the code is just under 1200 lines, with the virtual memory system additions representing just 200 of those. Another important aspect of the additions is that they do not affect the performance or functionality of any part of the system not related to the GuPPI; all other applications run completely unperturbed. The device driver provides for the continuous transfer of data between the GuPPI and main memory while absorbing jitter due to the scheduling of the signal processing applications. Data buffers in the driver, which can be as large as several hundred pages each, allow us to store data when we don't have enough cycles available, and then catch up by processing the buffer faster than real-time when the cycles become available. The virtual memory additions provide low overhead, high-bandwidth transfer of data between the application and the device driver by eliminating the expensive data copying between kernel and user space.

Running on a 200 MHz PentiumPro running Linux with a 33 MHz, 32 bit wide PCI bus, the I/O system has been shown to support continuous sample streams at rates up to 512 Mbits/sec. The peak burst rates are 933 Mbits/sec for input and 790 Mbits/sec for output. Details on the implementation can be found in [Is98].

Once the data is in memory, the software must process the data to produce a network frame. We have chosen to interact with the operating system at the IP layer; the interface to the kernel's network stack is through our *SoftLink* device driver. This driver appears to the kernel as just another network device driver. However, instead of exchanging packets between a hardware device and the IP layer, the *SoftLink* driver exchanges packets between our user-level application and the IP layer. For reception, the processing application converts incoming IF data into an IP packet; this packet is then handed to the *SoftLink* driver, which passes it on to the IP layer in the kernel, just as the device driver for any network card does. For transmission, the *SoftLink* driver accepts IP packets from the network layer and hands them up to the user-level process. Here the packet is processed producing the IF waveform. This waveform is transferred, via the GuPPI, to a 12-bit D/A converter (AD9713) which outputs the analog IF waveform.

## 4 Example Network Interface

This section presents an implementation of a software NIC designed to be compatible with a commercial frequency hopping radio operating in the 2.4 GHz ISM band, employing FSK modulation and supporting a data rate of up to 625 kbps [Sha97]<sup>3</sup>. Parameters such as the FSK frequency deviation and the spacing of the hopping channels can be dynamically modified in software; the only constraints im-

<sup>1</sup>For example the Rockwell 95x family of wideband receivers.

<sup>2</sup>Constructed using evaluation boards from RF Micro Devices [RF 97].

<sup>3</sup>The NIC was designed to be compatible with the 2.4 GHz frequency hopping radio from GEC Plessey, model DE6003.

posed by the hardware are the width of the IF band and the RF band to which the signal is converted. The results reported here utilized a 4.8 MHz wide IF band sampled at 10 MSPS with 12 bit resolution and an RF band centered at 2.45 GHz<sup>4</sup>. The transmission system generates continuous phase waveforms at a sustainable data rate of 320 kbps while hopping 1000 times per second; the reception currently runs at a rate of 64 kbps, supporting the same hopping rate. The following is a description of the transmission and reception applications, together with a discussion of the design issues and an evaluation of the performance of the system.

## 4.1 Transmission

The sequence of processing modules for the transmission application is shown in figure 3. The system interfaces with the host at the IP layer, through our *SoftLink* device driver. The first level of processing is the network framing. For this example, the packets were framed by inserting a start code and byte stuffing the data. A length code, indicating the total length of the packet including the stuffed values, was also inserted after the start code. The next module, representing the channel coding, takes the sequence of bits output by the network framing layer and performs byte framing, inserting start, stop and parity bits. Note that this system does not contain a line coding layer, which means that the symbols input to the modulation layer are actually bits.

The conversion of each symbol into a discrete signal is performed by the FSK module, representing the modulation layer. The multiple access technique is frequency hopping, which assigns the waveform to the appropriate IF frequency. In this implementation we combined the modulation and multiple access layers, which resulted a significant computational savings. This allowed us to directly generate the IF sinusoid corresponding to the particular bit and hopping frequency, rather than generating a sinusoid for each bit, and then re-modulating that sinusoid to the appropriate hopping frequency. All of the possible transmission waveforms are known a priori. There are two possible waveforms, corresponding to 1 or 0 for each hop frequency. All of these waveforms can be precomputed and stored at startup, significantly reducing the computation required to produce the transmitted waveform. On a 180 MHz PentiumPro. 2.2  $\mu s$  were required for producing the IF waveform corresponding to a single bit. This corresponds to a maximum possible transmit data rate of  $\approx 450kbps$ .<sup>5</sup>

The generation of continuous phase waveforms is fairly straightforward in software. The precomputed waveforms

<sup>4</sup>Since these results were obtained, our system has been shown to support real-time processing of IF bands sampled at 25 MSPS.

<sup>5</sup>The measurements were obtained using the PentiumPro cycle counter. To insure that branch prediction did not lead to erroneous cycle counts, the serializing "cpuid" instruction was executed prior to each reading of the counter. The overhead of this instruction, plus that of reading the cycle counter was quantified and subtracted out.

are actually oversampled, and only a sub-sampled set, corresponding to the output sampling rate, are copied into the output buffer. The oversampling allows us to index in to the buffer to match the phase, and the pattern is treated as a circular buffer, allowing the generation of waveforms for any bit period. After copying the samples to the output buffer, the phase value is updated and used as the index for the waveform corresponding to the next bit. In a similar manner, we are able to maintain continuous phase between hops, even when the hop occurs in the middle of the bit.

## 4.2 Reception

As is usually the case, reception is considerably more complex than transmission, although the sequence of processing modules is essentially the reverse of the transmission system shown in figure 3. The receiver must detect the presence of a valid transmission and sync to it, as well as perform the inverse function of each of the transmission layers. Again, combining the parameters of the frequency hopping and the FSK demodulation, we constrained the receiver to look for one of the two valid waveforms at a given hop frequency. Separate functions were implemented to track the hopping sequences, lock onto a bit boundary and demodulate the bits. These bits are de-framed, and then the IP packet is extracted. The SoftLink driver then hands the packet off to the host IP layer for processing. The number of cycles required for each reception function is given in table 1.

Function	Time
Frequency Hopping	0.5 $\mu s$ / hop
FSK lock	66.3 $\mu s$ / bit
FSK Demodulation	4.5 $\mu s$ / bit
De-Byte Framing	5.7 $\mu s$ / bit
De-Packet Framing	4.6 $\mu s$ / bit

Table 1: Average time required for each reception function on a 180 MHz PentiumPro.

## 4.3 Real-Time performance

The system we have constructed is not a hard real-time system, since it is implemented on a multi-tasking operating system without explicit real-time support. In a hard real-time system, each task has a deadline for completion, and there is some mechanism to insure that this deadline is met. Rather than insuring real-time performance with the tight synchronous control over the processing that is typical of many DSP and digital hardware designs, we take an approach that is statistical in nature. Although the actual number of cycles available over any given period of time varies due to demands on the system, the buffering and temporal decoupling provided by the I/O subsystem allow

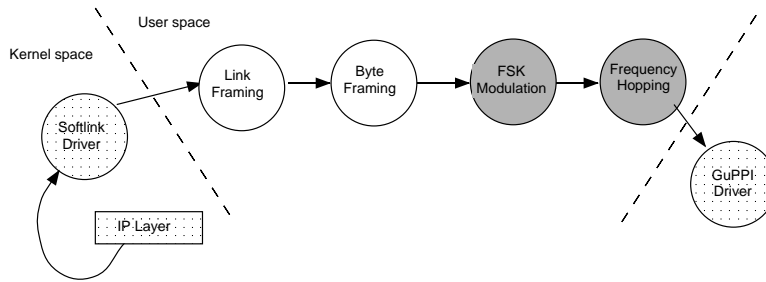


Figure 3: Software components of the transmission application

us the require only that, on average, there are enough cycles available to perform the necessary processing.

In order to quantify the performance of our system, we introduce the notion of *statistical real-time* performance. We characterize the system by defining a probability that the work will be completed within the specified time limit, and specifying the action that is to be taken when the deadline is not met.

The probability is determined by profiling the algorithm under the expected conditions on the work station. In this case the expected load was simply a Linux workstation running an X server, an NFS file system, and the usual network daemons (e.g. sendmail, amd, etc.). In another case, the expected load might involve other signal processing tasks, or significant user activity.

Figure 4 shows the distribution of times required to extract one bit using the FSK demodulation function from the example of section 4. The probability that more than  $10\mu s$  is required is less than 0.003 If we chose to stop the processing after  $10\mu s$  and make an arbitrary decision as to the value of the bit, this would correspond to an increase in the

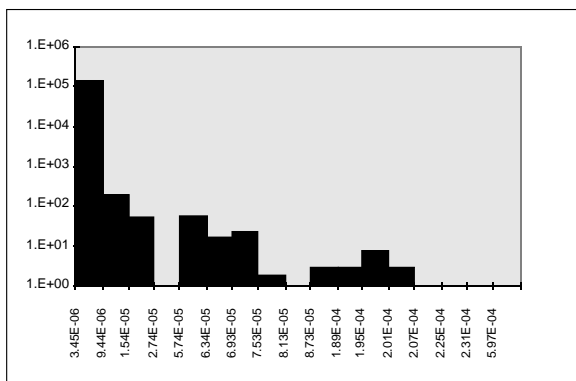


Figure 4: Histogram of the time required to perform the FSK demodulation function on a 180 MHz PentiumPro. Note that the vertical axis is on a logarithmic scale, and the horizontal axis is in units of seconds. The probability that more than  $10\mu s$  are required for processing is less than 0.003.

probability of a bit error by only 0.0015.

The application presented in the previous section had no mechanism to take alternate action when the deadline was not met. This was acceptable for a packet network application since jitter in packet arrival can be tolerated, and the probability of missing the deadline was very low. However, this is not the case for many applications, such as full-duplex voice communication. The action to be taken when the deadline is not met is application dependent. Possible actions could include dropping the data, continuing processing for an additional period of time, saving the data to be processed at a time when there are spare cycles available, or using an estimate of the output values. Dropping the data might be appropriate in applications such as a voice system, where missing small a amount of data may be more tolerable than an increase in latency.

The implementation of a mechanism to enforce statistical real-time constraints is currently under development. The system utilizes the exception handling methods in C++ and microsecond resolution timers for Linux<sup>6</sup>. The system time enforcement will likely be significantly coarser than a microsecond. However, this will still be useful to enforce constraints over an aggregate of processing functions or block operations. However, the results of failures, and the appropriate actions will have to be re-evaluated in this light.

As processor speed has increased, many real-time applications, such as audio processing, have migrated from dedicated digital hardware into software. These applications are able to run in real-time, without explicit real-time support in the operating system, because the number of cycles available is considerably greater than the number of cycles required by the application, and the probability of an event that causes a deadline failure is very small. With computer clock speeds continuing to double every eighteen months, the probability of a given application missing a deadline will continue to decrease. Furthermore, this brings an ever wider range of real-time signal processing applications within the grasp of systems based on general purpose processors running conventional, multi-tasking operating systems.

<sup>6</sup>Microsecond timers for Linux are under development at the University of Kansas, information on this project can be found at: <http://hegel.ittc.ukans.edu/projects/utime/>

## 5 Summary

In this paper, we have described an approach for building software wireless interfaces. We also described the design and implementation of one instantiation of that architecture, a NIC designed to interoperate with an existing commercial 2.4 GHz ISM band frequency-hopping spread-spectrum radio.

This implementation and our experience with it demonstrated some important facts about software NICs in general and our layered architecture in particular:

- It is feasible to build software NICs that achieve good performance,
- It is relatively easy, using our environment, to build software NICs. The code implementing this particular radio totaled only 520 lines of C++,
- Our architecture facilitates constructing devices that interoperate with existing systems, and
- Software NICs can ride the technology curve of commodity PCs. As PC's get faster, our wireless NIC will automatically get faster.

Our experiments did not illustrate one important potential advantage of software NICs, the ability to implement functionality that is not available through NICs that perform their signal processing on dedicated hardware. In the long run, this may be the most important advantage of all.

## References

- [AP93] Mark B. Abbott and Larry L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5):600–610, October 1993.
- [CT90] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM '90*, pages 200–208, Philadelphia, PA, September 1990.
- [Ism98] Michael Ismert. Making Commodity PCs Fit for Signal Processing. In *USENIX*, New Orleans, August 1998. IEEE. Submitted.
- [LM94] Edward A. Lee and David G. Messerschmitt. *Digital Communication*. Kluwer Academic Publishers, 2nd edition, 1994.
- [Ous90] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX*, pages 247–256, June 1990.
- [RF 97] RF Micro Devices, Inc., 7625 Thorndike Road, Greensboro, NC. *1997 Designer's Handbook*, April 1997.
- [Sha97] Alok B. Shah. Software-Based Implementation of a Frequency Hopping Two-Way Radio. Master's thesis, MIT, May 1997.
- [Sta97] William F. Stasior. *An Interactive Approach to the Identification and Extraction of Visual Events*. PhD thesis, MIT, July 1997.
- [Tan88] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ 07632, second edition, 1988.
- [TB96] David L. Tennenhouse and Vanu G. Bose. The SpectrumWare Approach to Wireless Signal Processing. *Wireless Network Journal*, 2(1), 1996.
- [Wep95] Jeffery A. Wepman. Analog-to-Digital Converters and Their Applications in Radio Receivers. *IEEE Communications Magazine*, 33(5):39–45, May 1995.