Marquette University
e-Publications@Marquette

Master's Theses (2009 -) Dissertations, Theses, and Professional Projects

Real-Time Transport of Internet Telephony Service Utilizing Embedded Resource-Constrained Systems

Kyle Persohn
Marquette University

Recommended Citation
Persohn, Kyle, "Real-Time Transport of Internet Telephony Service Utilizing Embedded Resource-Constrained Systems" (2012). *Master's Theses (2009 -)*. Paper 162.
http://epublications.marquette.edu/theses_open/162

REAL-TIME TRANSPORT OF INTERNET TELEPHONY SERVICE UTILIZING EMBEDDED RESOURCE-CONSTRAINED SYSTEMS

by

Kyle Persohn

A Thesis Submitted to the Faculty of the
Graduate School, Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

August 2012

ABSTRACT

REAL-TIME TRANSPORT OF INTERNET TELEPHONY SERVICE UTILIZING EMBEDDED RESOURCE-CONSTRAINED SYSTEMS

Kyle Persohn

Marquette University, 2012

This thesis presents a real-time framework for resource-constrained devices that improves the listening quality of Voice over Internet Protocol calls transported over congested networks. Many VoIP standards and implementations exist, but gaps in the design space encourage further exploration that previous work fails to address. We describe an experimental hardware platform that expands upon a previous design to accommodate technical research and educational needs.

Our framework, based on the Real-Time Transport Protocol, integrates closely with existing software constructs available in the Embedded Xinu operating system. We offer features derived from RTP by means of a kernel device that alleviates an application from directly interacting with the underlying protocol. An example application based on Xinu's RTP implementation demonstrates measurable robustness to packet loss and delay variation (jitter)—adverse conditions affecting networks used for VoIP, such as the Internet.

Results show that Xinu RTP improves PESQ MOS over the previous design limited to UDP transport. Typically, we observe a 17% to 25% increase in MOS for lost packets and near perfect scores for delay variations within the device's sliding window. Moreover, these improvements are possible with minimal computational overhead.

ACKNOWLEDGMENTS

Kyle Persohn

I offer many thanks to several people who made this work possible:

- Dennis Brylow, my thesis adviser and academic idol, for providing this research opportunity and encouraging aspiration to standards of excellence;
- Mike Johnson and Richard Povinelli, my committee members, for making cross-department research possible and promoting student-driven research in their thought provoking courses;
- Mike Ziwisky and Adam Mallen, my fellow graduate colleagues, for numerous intellectually stimulating discussions and fostering the most entertaining work environment;
- Matt Bajzek, Victor Blas, Jason Cowdy, and the entire Systems Laboratory community for their contributions to the Embedded Xinu Project;
- Jim and Carol Persohn, my parents, for supporting my education and extracurriculars that gave me the tools for success; and
- Ashley Milner, my significant other, for motivation and sharing my attention with the demands of academic research.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . **i**
LIST OF TABLES . . . . **iv**
LIST OF FIGURES . . . . **v**
CHAPTER 1 Introduction **1**
1.1 Thesis Statement 1
1.2 Problem Synopsis 1
1.3 Contributions 2
1.4 Organization of Thesis 3
CHAPTER 2 Background **4**
2.1 Internet Telephony 4
2.1.1 Comparison with the PSTN 5
2.1.2 Interoperating with the PSTN 7
2.2 Embedded Systems 8
2.2.1 Peripherals 8
2.2.2 Software 10
2.3 Digital Signal Processing 11
2.3.1 Sampling Theory 12
2.3.2 Voice Data Compression 13
2.3.3 Analog Reconstruction 14
2.4 Computer Networks 15
2.4.1 Layered Architecture 15
2.4.2 Transport Protocols 17
2.4.3 Real-Time Transport Protocol 18
2.4.4 Telephony Protocols 20
CHAPTER 3 Related Work **21**
3.1 Embedded Systems Education 21
3.2 Embedded VoIP 23
3.3 A Prototype Design 25
3.4 Existing RTP Implementations 27
CHAPTER 4 The XinuPhone **30**
4.1 Theory of Operation 30
4.2 Goals and Requirements 32
4.3 Hardware Design 33
4.3.1 Microprocessor Selection 34
4.3.2 Pre-Amplification Stage 35
4.3.3 Anti-Aliasing Filter 37
4.3.4 Output Amplifier 38
4.3.5 Serial Transceiver 39

4.4 Firmware Design 40
4.4.1 Device Configuration 40
4.4.2 Run-time Operations 44
4.4.3 Task Prioritization 47
CHAPTER 5 A Kernel-based Approach to RTP 52
5.1 Design Approach 52
5.1.1 Integration into Embedded Xinu 53
5.1.2 Basic Mode 55
5.1.3 Sequence Mode 56
5.2 Implementation of Enhanced Features 58
5.2.1 Additions to Support Ordered Queuing 59
5.2.2 The Need for Multiple Threads 61
5.2.3 Packet Queue 63
5.2.4 Other Enhancements 65
5.3 Sample Application 67
CHAPTER 6 Experimental Analysis 70
6.1 Correctness Verification 70
6.2 Metrics and Measurements 72
6.2.1 Mean Opinion Scores 73
6.2.2 Perceptual Evaluation of Speech Quality 74
6.2.3 Determining Test Parameters 76
6.3 Experimental Setup 77
6.4 Test Results 80
6.4.1 Control 81
6.4.2 Dropped Packets 82
6.4.3 Tuning the RTP Device 83
6.4.4 Packet Delay Variation 84
6.5 Summary of Findings 86
CHAPTER 7 Conclusion 87
7.1 Contributions 87
7.2 Future Work 89
7.3 Summary 90
BIBLIOGRAPHY 92
APPENDIX A Audio File Transport Test Application 97

LIST OF TABLES

6.1 RTP Test Suite 71
6.2 Mean Opinion Score Interpretations 73
6.3 Audio File Data Set 76
6.4 Control Experiment 81
6.5 Emulated Packet Loss 82
6.6 RTP Packet Timeout Adjustment 83
6.7 Emulated Delay Variability 85

LIST OF FIGURES

2.1 Zero-Order Hold Signal Reconstruction Model 14
2.2 Communication System Abstraction Layers 16
4.1 XinuPhone System Block Diagram 31
4.2 A Simple Op-Amp-based Pre-Amplification Stage 36
4.3 Sallen-Key Active Low-Pass Anti-Aliasing Filter 37
4.4 Output Amplifier for Driving Speakers/Headphones 39
4.5 Charge-Pump Serial Level Converter 40
4.6 Timer Register and Clock Scalar Configuration 42
4.7 Code for Initializing DAC Peripheral 43
4.8 ADC Interrupt Service Routine 45
4.9 Repeated Segment of `main` Routine 47
4.10 Serial Data Flow Between DSC and Router 48
4.11 Verifying Software Run-Time Behavior with a Logic Analyzer 50
5.1 Thread Relationship for Handling Incoming Packets 59
6.1 Network Topology for PESQ Testing 77

CHAPTER 1

Introduction

1.1 Thesis Statement

Simple primitives available to a lightweight embedded operating system can be used to construct a real-time framework, which improves the quality of voice communications transmitted over an unreliable packet-based network.

1.2 Problem Synopsis

Internet telephony networks are rapidly replacing legacy systems connected to the Public Switched Telephone Network (PSTN). Voice over Internet Protocol (VoIP) technology utilizes packet-based networks, the Internet for example, which have several advantages over the circuit-switched PSTN. Nevertheless, traditional packet-based networks tend to be unreliable mechanisms for transporting deadline sensitive data such as voice communications. Consequently, it is necessary to have additional protocols that are robust enough to provide reliable data transport while maintaining minimal overhead to ensure timely and efficient processing.

Furthermore, an end-user VoIP communications device is often realized as an

embedded system. Such devices must implement reliable communications with limited processing, memory, and energy resources. While proprietary commercial solutions for VoIP communications exist, the community lacks inexpensive, open-source tools specifically targeted towards embedded platforms. Additionally, the resources currently available lack both the simplicity necessary to serve as a pedagogical tool and the versatility necessary to promote novel research in the field.

1.3 Contributions

The work presented in this thesis is summarized by two distinct contributions:

First, we describe an open-source hardware/software platform, referred to as the *XinuPhone*, which is suitable for interactive VoIP education as well as the development of emerging voice communications enhancements. The XinuPhone improves upon a previous embedded VoIP implementation by utilizing interrupt-driven software routines, adding robust digital signal processing (DSP) hardware, and transitioning from an outdated development kit to readily available discrete components. This new implementation demonstrates it is possible to offer an open and flexible platform at a cost that is highly competitive with restrictive, proprietary commercial offerings [1].

Second, we implement a lightweight version of the Real-Time Transport

Protocol (RTP) using basic software constructs provided by the Embedded Xinu operating system (OS). Xinu RTP uniquely combines application layer functionality into a reusable kernel device. Furthermore, we demonstrate that Xinu RTP improves the quality of voice data transmitted under capricious network conditions.

1.4 Organization of Thesis

The remainder of this thesis is organized into the following chapters:

- Chapter 2 reviews background related to embedded systems, Internet telephony, digital signal processing, and computer networks.
- Chapter 3 summarizes previous work related to VoIP systems, educational initiatives, network protocols as well as the problems left to be addressed.
- Chapter 4 describes the XinuPhone Internet telephony platform including hardware designs, software routines, and improvements to the reference design.
- Chapter 5 highlights the advantages of implementing RTP as a device in the Xinu OS along with the sacrifices that accompany an embedded realization.
- Chapter 6 analyzes the performance of the XinuPhone and Xinu RTP using industry inspired evaluation techniques.
- Chapter 7 presents noteworthy conclusions and suggestions for extensions to this work.

CHAPTER 2

Background

The broad topic of Internet telephony is built upon key concepts from a variety of engineering and computer science disciplines. Embedded systems are critical to the delivery of VoIP communications because end-users rely on specialty resource-constrained devices to interface speech signals with VoIP networks. Although users can also use softphone applications on personal computers, embedded VoIP solutions emulate a user experience most similar to traditional telephony. Next, the digital signal processing field is important since the analog speech signals are digitized before transmission and then reconstructed on the receiving end. Lastly, the backbone of VoIP relies on the same computer network framework as the Internet; however, telephony requires additional protocols and specifications beyond those that support the operation of the World Wide Web.

2.1 Internet Telephony

Internet telephony refers to the general concept of voice and message data that is transported over the Internet. Although often used synonymously, VoIP is in fact a subset of Internet telephony. While VoIP focuses on the vocal aspects of

communication, Internet Telephony may also refer to Internet delivery of other services such as fax and the Short Message Service (SMS), which is popular on mobile phone handsets [2]. This work primarily focuses on the voice aspects of Internet telephony; nevertheless, many of the advantages also apply to other messaging mediums.

2.1.1 Comparison with the PSTN

Switching technology is a distinguishing difference between Internet telephony networks and the PSTN. VoIP services use packet-based transport of telephony data to efficiently multiplex many conversations within the same communication channel. Each packet contains a small data payload in addition to meta information used to distinguish different communications in a shared medium. Conversely, the PSTN establishes virtual and physical dedicated point-to-point circuits between each party. While circuit-switching requires less overhead, it does not scale well and does not naturally integrate with the framework of the Internet [3].

The advantages of packet-based services mostly relate back to financial cost. Service providers can handle more customers with fewer physical resources so the service is more profitable. Furthermore, equipment costs for specialty phone circuits are high relative to network appliances that are mass-produced for the infrastructure empowering the Internet. By unifying data and voice networks, both

backbone service providers and institutions save money on equipment and maintenance costs. Since VoIP is so tightly couped with the Internet, it is easy to offer "unified communications" by integrating VoIP with other online services such as email, customer relations databases, visual voicemail, and conferencing applications. Finally, businesses with in-house VoIP deployments can leverage open-source and commercial software suites to provide extra services like caller ID, call forwarding and menu systems without monthly recurring charges from the telecommunications companies [4].

Of course, there are also downsides to integrating voice and data services into the same core technology. The PSTN provides natural redundancy for communications when Internet failures occur. Moreover, VoIP services are sensitive to local power failures, whereas the PSTN provides its own direct current source from the central offices. Consequently, it is common practice to use an uninterruptible power supply (UPS) with any customer-premises equipment (CPE). The dynamic nature of VoIP also presents additional challenges for emergency services. The Internet addresses used to route VoIP calls do not necessarily correlate to geographical addresses; therefore, location-based 911 services require that customers maintain their own records of emergency information. In the United States, Enhanced 911 (E911) mandated by the Wireless Communications and Public Safety Act of 1999 handles some of these concerns [5].

2.1.2 Interoperating with the PSTN

The transition from the PSTN to Internet telephony is a large-scale ongoing process that, so far, spans over a decade. Naturally, interoperability between both networks is necessary to facilitate communications between modernized and legacy installations. Businesses deploy IP-enabled private branch exchange (PBX) equipment and PSTN gateways to interface internal VoIP networks with external POTS (plain old telephone service) lines and vice-versa. Likewise, residential customers can purchase an analog telephone adapter (ATA) to use traditional phone handsets with a growing number of Internet telephony service providers (ITSP).

Devices that bridge analog and digital telephony networks rely on a hybrid device to demultiplex the communication channel used by POTS lines. A typical subscriber line consists of a single pair of wires used to carry audio both transmitted and received by the local party. A 2-to-4 wire hybrid interface translates the multiplexed audio into separate transmit and receive channels. The telephone hybrid was not pioneered by VoIP; in fact, hybrids have long been in use by analog networks to interface two-wire customer lines with four-wire backhaul lines used by the central office (CO) [6]. In VoIP systems, DSP hardware emulates the role of the CO, but the hybrid performs the same function. Hybrids use balanced transformers or operational amplifiers to isolate each audio channel; however, impedance mismatches couple the two audio paths creating undesired echo effects.

Consequently, digital hybrids often employ additional DSP line echo cancellation (LEC) routines to mitigate degradation in quality caused by the hybrid circuitry [7], [8].

2.2 Embedded Systems

Embedded systems, for lack of a precise definition, are usually computers designed to perform specific tasks [9]. While the amount of computing power available to embedded systems is rapidly increasing they are often considered *resource-constrained* systems relative to general-purpose desktop and server computers. In addition to processor speeds, resource-constrained systems may also be limited in memory and energy, as in the case of battery operated devices. Embedded systems provide computing power for many everyday devices such as entertainment systems, cooking appliances, network equipment, and navigation systems. Additionally embedded devices are often crucial subcomponents of larger systems, for example, vehicle safety, aircraft guidance and control, factory automation, and medical imaging [10].

2.2.1 Peripherals

An embedded device interacts with its environment using various peripherals. Communication peripherals such as serial peripheral interface (SPI),

inter-integrated circuit (I^2C), RS-232, and universal serial bus (USB) allow different embedded processors to share data with other closely located devices. Likewise, network peripherals such as Ethernet, controller area network (CAN), local interconnect network (LIN), and wireless radios allow cooperating embedded systems to exchange information on a broader scale. Through analog to digital (A/D) and digital to analog (D/A) converters, embedded devices interact with analog signals and sensors. Similarly, general purpose input/output (GPIO) ports enable simple digital interaction with push buttons, relays, stepper-motors, and light emitting diodes (LEDs).

The user interface on an embedded system, if any, is usually limited in comparison to the periphery found on a general purpose computer. Simple buttons, switches, and keypads are common basic inputs. LCD character displays and indicator lights may provide feedback to the operator. In lieu of elaborate user interfaces, embedded systems leverage their communication and network connectivity to receive configurations and commands from remote systems. Due to the primitive nature of embedded system interfaces and interactions with external events, debugging and troubleshooting are more complex tasks. Most devices offer a Joint Action Test Group (JTAG), In-Circuit Serial Programming (ICSP), or similar interface that allows another system to externally control and query an embedded system during the development process.

2.2.2 Software

At the heart of any embedded system lies one or more microcontroller units (MCU) or digital signal processors executing software code to control the hardware peripherals. Application code running directly on the platform without the use of an operating system is referred to as a *bare metal* application. Developers can still leverage existing software libraries to interact with hardware peripherals and reduce the amount of code needed to be written from scratch. Even without a full blown OS, bare metal systems can manage the processor's response to various events using interrupts. The code executing on the XinuPhone serial audio interface described in this thesis is an example of a bare metal application.

More complex embedded systems employ operating systems to provide applications with a basic layer of support on top of the hardware. An embedded OS typically provides common mechanisms for memory management, task scheduling, inter-process communication, storage, device drivers, and networking [11]. Popular embedded operating systems include μC/OS-II, Embedded Linux, TinyOS, VxWorks, and Windows CE. The XinuPhone uses the Embedded Xinu operating system to host the VoIP application code running on a Linksys WRT54GL router.

Embedded Xinu is a modernized RISC implementation of the classic Xinu design pioneered by Dr. Douglas Comer in the 1980s [12]. Preemptive priority

scheduling in Embedded Xinu arbitrates between various tasks running on the system. Embedded Xinu also provides inter-process communication via message passing, semaphores, and a basic TCP/IP network stack. Even though Embedded Xinu lacks the comprehensiveness of Embedded Linux, its basic software primitives are sufficient for building an Internet telephony system.

2.3 Digital Signal Processing

The field of digital signal processing (DSP) concerns the representation of signals as quantized discrete-time series. Many signals occur naturally in a continuous analog form; therefore, DSP often involves converting to a digital representation, manipulating the digital samples, and finally reconstructing an analog waveform. With advanced computing power readily and cheaply available, the possibilities for intermediate digital processing are far-reaching. Usually, these operations take the form of filtering or coding. Filters manipulate the characteristics of a signal; coding prepares signals for transmission by compressing the representation into a more efficient form or adding additional error-checking information. Many embedded systems contain specialized digital signal processor chips with architectures highly optimized for matrix multiplication and other operations common to DSP [13].

2.3.1 Sampling Theory

Computers manipulate information represented by discrete values. Analog signals must first be converted to a digital representation in order to transmit continuous-time signals in packets. A hardware peripheral called an analog to digital converter (ADC) is responsible or this process. On the time axis, periodic measurements called samples provide a complete representation of the analog waveform. The Shannon-Nyquist Sampling Theorem states that a signal must be sampled at a minimum of twice its highest frequency in order to accurately reconstruct it from the discrete samples [14]. For example, telephony systems typically sample speech signals at 8 kHz because human speech ranges from about 300 Hz to 3.4 kHz. The highest frequency, 3.4 kHz, is less than half the sampling rate, 4 kHz; therefore, the sampling theorem is satisfied [8], [15]. The low-pass filter needed to reconstruct the analog signal cannot have an ideal vertical cut-off, hence the extra headroom beyond the theoretically necessary 6.8 kHz sample rate.

Each sample represents an amplitude value, which, for DSP purposes, must assume a discrete representation as well. Through a process called quantization, the ADC assigns a discrete amplitude level to each sample. The bit-depth of the ADC dictates the number of distinct levels. For instance, the XinuPhone uses a 12-bit ADC, which distinguishes between 2^{12} or 4096 quantization levels. Together, the bit-depth and the sampling rate determine the bandwidth necessary for transmitting

the digitized signal. A system using the parameters described above requires

$$8,000\,\frac{\text{samples}}{\text{sec}} \times 12\,\frac{\text{bits}}{\text{sample}} = 96,000\,\text{bps}. \tag{2.1}$$

In order to reduce the bandwidth requirements, telephony systems use various compression schemes to decrease the number of bits representing each sampling during transmission.

2.3.2 Voice Data Compression

Internet telephony systems compress and expand (compand) voice data samples using a CODEC (coder-decoder). The CODEC exploits certain signal properties allowing for a more compact representation using less bits per sample. In telephony, the International Telecommunication Union (ITU) G.711 specification is the *de facto* standard supported by nearly all VoIP systems. ITU G.711 maps signed 16-bit values to an unsigned 8-bit representation. This scheme works under the assumption that compressing less frequently occurring amplitudes will not be significantly noticeable to the listener. By applying this CODEC to speech the samples in (2.1), we achieve a 33% reduction in transmission bandwidth. 64 kbps serves as a nominal baseline for other CODECs to improve upon. The XinuPhone uses the G.711 μ-law variant due to its widespread use in the United States. Additional library support is available for these other popular VoIP CODECs:

G.711 A-law (used in Europe), G.726 Adaptive Differential Pulse Code Modulation (ADPCM), and Speex [16], [17] [18].

2.3.3 Analog Reconstruction

The final step in the DSP chain involves reconstructing an analog waveform from the digital sample sequence. In the XinuPhone, a hardware digital to analog converter (DAC) performs this function using delta-sigma modulation. First the DAC upsamples the incoming signal to relax the specifications on the analog reconstruction filter. In reference to the zero-order hold model, this effectively increases the granularity of the piecewise step function approximating the analog amplitudes. In other words, the run of the stair-step pattern (Figure 2.1) is much smaller so a less complex filter may be used to smooth the jagged edges into a continuous analog signal [15]. Next, the delta-sigma modulator produces a bit sequence representing the desired analog voltage [19]. Finally, the reconstruction filter averages the bit stream generating the desired analog waveform.

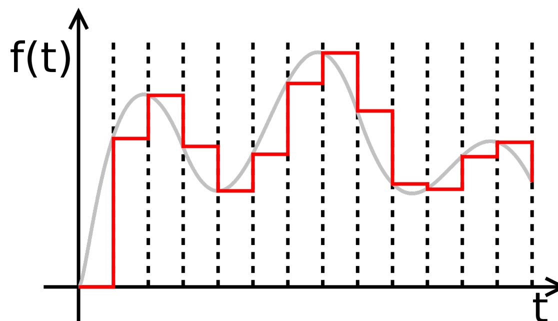

Figure 2.1: Zero-Order Hold Signal Reconstruction Model

2.4 Computer Networks

Internet Protocol networks serve as the transport mechanism for VoIP communications. These packet-based networks can efficiently multiplex numerous users in the same channel and offer many additional features through a layered system of headers and payloads. The work in this thesis is primarily focused on the upper layers that concern how the data is transported and what application features increase robustness to adverse network conditions.

2.4.1 Layered Architecture

Dividing a communication system into layers of abstraction allows designers to build modular protocols targeted to specific goals. Each layer provides support to those above it given certain assumptions so the higher levels do not need to reimplement features lower in the stack. The Open Systems Interconnection (OSI) model and the TCP/IP model are two common layered structures describing communication networks. Depending on the literature, the layers of the TCP/IP model vary [20], [21]. In this work, we refer to the TCP/IP model with five layers that correspond to the lower portion of the OSI model as shown in Figure 2.2 [22]. Thus, common numerical references such as layer 2 meaning the data-link/network interface layer hold true. However, the discussion to follow does not require the specificity of the OSI model, so we simply refer to the upper three OSI layers as the application layer.

	OSI Model	**TCP/IP Model**
7	Application	Application
6	Presentation	
5	Session	
4	Transport	Transport
3	Network	Internet
2	Data Link	Network Interface
1	Physical	Hardware

Figure 2.2: Communication System Abstraction Layers

The lower layers of each model concern the hardware and addressing schemes used to deliver network packets. The physical layer describes the hardware connection to the transmission medium including bus contention resolution and modulation. The distribution of traffic on a local network, also *the link*, is handled by layer 2 frames with addressing information restricted to the immediate network. Finally, the information in layer 3 connects nodes on one or more wide-area networks [20], [23]. In this thesis, we assume the lowest three layers are sufficient as-is for Internet telephony use and resolve any problems that may arise higher up in the stack.

The upper layers deal with data transport and application-specific features. Multiplexing, reliability, and flow control are examples of tasks handled by the transport layer. Application layer protocols provide support for specific services such as domain name addressing, dynamic routing, web, and email. In the latter portion of this thesis, we analyze issues that arise in the transport layer. In response

to these problems, we propose a modular resolution implemented below the application layer such that multiple user programs can share it.

2.4.2 Transport Protocols

Most high-level network traffic is carried by one of two transport protocols: Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). TCP is a connection-oriented protocol that uses ports to distinguish between multiple connections. Some advantages of TCP include error-free transmission, proper ordering, resending of lost packets, and flow control. However, these features come at the cost of the overhead to manage a sliding window system of acknowledgements for each packet sent [23]. As a result, TCP is unsuitable for applications where on-time delivery is critical, such as voice communications.

On the other hand, UDP is a lightweight best-effort protocol that sacrifices the robustness of TCP for increased performance in time-sensitive applications. Like TCP, UDP provides multiplexing functionality through the use of ports. UDP does not, however, protect against packets that are delayed, reordered, or lost during transmission [20]. Consequently, UDP is popular with streaming media applications where dropping lost, late, or corrupted data is better than waiting for retransmitted packets. These issues still impact the quality of VoIP communications, so additional support above transport layer is necessary to ensure reliable transmission.

2.4.3 Real-Time Transport Protocol

The Real-Time Transport Protocl (RTP) adds support for streaming communication such as audio, video, and sensor data [24]. Although technically an application layer protocol, RTP offers transport style features as its name suggests. RTP does not natively provide any multiplexing capability; thus, it relies on an underlying transport protocol for this function. Many streaming applications run RTP on top of UDP as a more efficient alternative to TCP.

An RTP header supplements additional information that the underlying transport protocol may lack. For instance, sequence numbers allow the application to sort out-of-order packets and timestamps enable silence and packet-loss detection. Additionally, a payload type identifies the data encoding and special identifiers make it possible to synchronize content from specific sources. Unlike other protocols, RTP does not strictly mandate what the application must do with this information. It is up to the receiver to utilize and interpret the RTP header in the context of a particular application. In the spirit of flexibility, RTP also provides header extension mechanisms to explore other out-of-band data that might assist streaming applications [24].

Companion documents in the form of profiles and payload specifications accompany the main RTP RFC. For example, RFC 3551 describes a profile for

audio, containing the payload type to codec mappings relevant to VoIP [25]. A profile may also define the use of the marker control bit or add functionality of RTP through its header extensions. Payload formats dictate the layout of a particular encoding of data that follows the RTP header. Byte-ordering, channel arrangement, and recommended sample/frame sizes are common attributes associated with each format specification. The XinuPhone operates under the *RTP Profile for Audio Video Conferences with Minimal Control* using, by default, the payload format for Pulse Code Modulation (PCM) G.711 μ-law encoded audio sampled at 8 kHz in 20 ms blocks.

In addition to the protocol for RTP data, RFC 3550 also specifies the Real-Time Transport Control Protocol (RTCP). This optional companion protocol offers statistics and minimal channel control for quality of service (QoS) monitoring and basic in-band identification. QoS statistics are useful in large streaming sessions with many participants on links of varying reliability. Using RTCP feedback such as jitter, round-trip delay, and lost octet counts the sender can monitor changing network conditions and take corrective action, perhaps switching the CODEC. RTCP also supports primitive messages, allowing participants to send basic identifications and contact information. The XinuPhone, which makes simple point-to-point calls, omits this conferencing targeted protocol in favor of enhancements to the data stream.

RTP paired with UDP can provide reliable transport of streaming audio data suitable for Internet telephony. However, RTP itself is not a comprehensive solution for VoIP calls. RTP does not provide advanced call signaling features, format negotiation, or native security. These features beyond the scope of RTP are handled by other telephony-related protocols and profile extensions.

2.4.4 Telephony Protocols

Internet telephony relies on a bevy of other protocols in addition to the data transport. Most importantly, calls require a signaling mechanism such as the Session Initiation Protocol (SIP) or ITU H.323 [26], [27]. Signaling protocols handle call setup/teardown, CODEC agreement, and Uniform Resource Identifiers (URI) similar to POTS phone numbers. Often signaling protocols leverage a formal rule-set such as the Session Description Protocol (SDP) to define the session parameters [28]. Lastly, SRTP and ZRTP address security concerns by defining a secure profile extension to RTP and an associated method for exchanging cryptographic keys [29], [30].

CHAPTER 3

Related Work

The contributions described in this thesis draw from a diverse collection of previous work. As an education platform, the XinuPhone receives influence from pedagogical initiatives in the embedded systems area. Furthermore, research-oriented features of the XinuPhone draw inspiration from existing academic and commercial embedded telephony systems. Also, we discuss alternatives to our implementation of RTP within Embedded Xinu. The following sections highlight positive aspects and shortcomings of work related to our approach.

3.1 Embedded Systems Education

Embedded systems dominate the computer market. With respect to processor sales, MCU and DSP chips outsell their general-purpose counterparts by several orders of magnitude. Conversely, the focus of most undergraduate computer engineering and computer science programs is on desktop and server programming [31]. Benson et al. propose exposing students to embedded computing concepts earlier in their education by offering summer programs at the high school level and weaving additional curriculum into undergraduate introductory courses. Early exposure to

embedded computing encourages students to engage in systems courses for upper-division electives ultimately preparing them to fill a void in industry for systems engineers [32]. The course content surrounding the XinuPhone as described in [1] assumes a modular structure, which aims to address the claims in [32].

A second set of observations comes from work at Carnegie Mellon, where the opportunities for embedded systems education are plentiful. Koopman et al. find that experience in traditional computing concepts do not necessarily translate into the embedded domain. Students' preconceived ideas that tend to waste memory, CPU time, and network bandwidth must be unlearned before embedded software designs become successful [33]. The XinuPhone content modules emphasize hands-on assignments with resource-constrained hardware throughout various facets of the undergraduate computing experience. Koopman's group also finds that realistic implementations exposing students to the quirks and limitations of embedded systems promote increased knowledge retention. Accordingly, the XinuPhone motivates students by interactively designing an Internet telephony system, a realistic application that explains technology encountered in their everyday lives.

Project Nexos is a joint effort by Marquette University (MU) and the University of Buffalo (UB) promoting the use of experimental laboratory assignments in undergraduate embedded systems courses. Nexos emphasizes

interactive learning with actual hardware, as opposed to simulations or software emulators. Simple, yet functional feature subsets take precedence over comprehensive production-grade software architectures. Consequently, the material is fathomable in single-semester courses and still provides working hands-on demonstrations. Curriculum associated with Project Nexos highlights time-oriented and interrupt-driven motifs frequently encountered in embedded systems, which are often overlooked in traditional courses [34]. The XinuPhone extends the Nexos framework with an external serial peripheral exemplifying reactive programming and strict awareness of time. As design decisions arise throughout this thesis, it is the ideologies of Project Nexos that guide our implementation choices and justify the various trade-offs encountered.

3.2 Embedded VoIP

Work specifically relating to an inexpensive embedded Internet telephony platforms is unfortunately limited. However, work on embedded multi-media systems still provide useful guidance relating to streaming content on resource-constrained devices. For instance, Zhang et al. describe an RTP implementation in Embedded Linux that interprets the RTP header in a similar fashion to our Xinu version [35]. Due to the nonspecific nature of the RTP RFC, we look to other published works, such as Zhang's, for ideas governing the reception of RTP packets. In a similar

light, Cuijuan et al. realize a VoIP gateway using an embedded digital signal processor [36]. Based on the hardware specifications described in their paper, we selected a platform with comparable resources based on overlapping needs, such as G.711 CODEC operations. While these works each make significant contributions to the field, they both lack publicly accessible hardware and software designs. The descriptions of their implementations, restricted to publishing page limits, are insufficient for reproducing their efforts or leveraging existing progress for future work. In response to these frustrations, we have modeled our approach after popular work by the Arduino Team [37]. As such, the work described in this thesis—both hardware designs and software code—is available to the public [38] under open licenses promoting reuse and future expansion.

The most closely related work is that of Rowetel's $10 ATA Project [39]. Rowetel designs open hardware and software supporting telephony in developing countries. Like the XinuPhone, the $10 ATA Project leverages the ubiquitous Linksys WRT54G series for its embedded networking functionality. Both projects employ a handful of discrete components as a serial sound interface for the router. The two designs differ mostly in software: the $10 ATA is based on Asterisk [4] whereas the XinuPhone communicates with other devices using a simplified, custom application. The Asterisk suite provides a robust set of features suitable for everyday communications. On the other hand, the Embedded Xinu VoIP

application demonstrates simple call functionality in a concise format more akin to the goals of an experimental platform for undergraduate education and research experimentation. Given the overlap in hardware design goals, both projects stand to benefit from one another's contributions.

3.3 A Prototype Design

The work described in this thesis is largely based upon a design prototype by Lund as described in his master's thesis [40]. Lund demonstrates that it is possible to construct a functional, basic embedded telephony system by pairing off-the-shelf components with the lightweight Xinu operating system. Furthermore, he establishes a performance baseline against which our improved system may be compared. The XinuPhone leverages Lund's existing framework which includes source code contributed to Embedded Xinu, the design concept of enhancing a commercial router with hardware capable of audio processing, and several tools for testing and troubleshooting.

While Lund's implementation is a substantial stride forward in the non-proprietary embedded VoIP design space, there are a number of areas amenable to improvement. For one, the AppleTalk microphone Lund's design relies on has been discontinued making it increasingly difficult for others to reproduce his platform. An improved design should include a basic pre-amplifier to enable

compatibility with readily available dynamic or electret microphone headsets. Additionally, the DRAGON12 development platform has several pitfalls of its own. For instance, the interrupt logic flaw Lund encountered prohibits proper asynchronous serial communication between the 68HC12 and the Linksys router. The DRAGON12 also contains many unnecessary components that effectively double the production cost. Reproducing the DRAGON12's functionality with discrete components requires specialized skills and tools for surface mount devices that are likely beyond that of the platform's target audience. While the cost analysis looks great on paper, several logistical factors prohibit a scalable realization of Lund's design. An improved design can address all of these issues with a better substitute for the 68HC12 microcontroller that is not constrained to a development board.

In his future work section, Lund lists a collection of suggested ways to extend the current state of his embedded VoIP platform. This thesis expands upon Lund's suggestion to improve the transport robustness of UDP packets by implementing an additional protocol. Specifically, we address this issue using the well-established Real-Time Transport Protocol. Furthermore, we contribute insight into the impact of design decisions on overall voice call quality.

3.4 Existing RTP Implementations

With the current RTP request for comments (RFC) in publication for nearly a decade, several existing implementations are already well established. In this section, we review the contributions of oRTP, LIVE555 Streaming Media, and GNU ccRTP as well as the shortcomings that inspired a fresh look at RTP within Embedded Xinu. These existing RTP libraries provide a robust set of features that work well for the applications that depend on them; however, their complexity makes them unsuitable for policy testing and educational use.

The oRTP library is most notably known for its use in the Linphone, a VoIP software client (softphone) available for many desktop and mobile platforms [41]. Hewlett Packard also uses oRTP in their OpenCall Media Platform. Similarly, LIVE555 Networks produces a set of streaming media libraries for real-time transport of many audio and video content types. The ever popular VLC and MPlayer media players both rely on LIVE555 for RTP support [42]. Lastly, GNU ccRTP is a fully functional RTP stack published by the GNU Telephony project that features many popular extensions for securing voice calls over the Internet. The Twinkle Softphone and SFL Phone clients rely heavily upon the ccRTP library [43].

The RTP specification described in RFC 3550 leaves many of the implementation design choices up to the application programmer. While the RFC

takes care to clearly define packet header structures, performance metric calculations, and payload formats, there is little mandate for how any of this information should be interpreted by the receiver. We draw upon the afore-mentioned implementations for inspiration on how to leverage this information to improve application performance. Packet reordering, delay variation compensation, and scheduling are examples of such inspirations. As common denominators to the status quo, these are areas of interest to test how design choices may impact system performance.

Although it is derived from the same specification, Xinu RTP is quite different from previous approaches. A significant distinguishing factor is how Xinu RTP manifests as a kernel device driver as opposed to an application library (discussed further in Chapter 5). Our implementation sacrifices many of the production features found in other RTP stacks for a significantly simpler and more manageable code base. This is necessary to best accommodate test hooks for performance evaluation and to keep the code comprehensible for our pedagogical mission.

Complications due to the nature of free and open-source licensing models provide further motivation for implementing Xinu RTP from the ground up. The GNU Telephony suite uses the GNU General Public License (GPL), which mandates that the source code of subsequent works be openly available under the

same license. Similarly, both LIVE555 Streaming Media and oRTP are licensed under the GNU Lesser General Public License (LGPL). The LGPL is more flexible than the GPL in the sense that closed-source works can link the LGPL software without distributing proprietary code. Nonetheless, *derivative* works that leverage LGPL code must still release source under LGPL or GPL [44]. Assuming the concepts portrayed in this thesis are more valuable than the code itself, the LGPL and GPL are equally limiting. Embedded Xinu, and subsequently Xinu RTP, is available under the Modified BSD License; therefore, proprietary code is able to freely utilize these conceptual contributions without potential legal ramifications.

In summary, it is necessary to take a fresh look at RTP in order to meet the educational and research goals of this platform while still making these contributes as widely accessible as possible.

CHAPTER 4

The XinuPhone

Testing the interactions between real-time hardware and software components requires an experimental platform that provides a sufficient amount of control over the underlying components. In this chapter, we describe the XinuPhone–a tool for evaluating the impact of design choices on the performance of an Internet telephony system. We start with an overview of the high-level design and the specific requirements influencing various facets of the implementation. Then, for each of the hardware and software designs, we describe major components responsible for the desired operation.

4.1 Theory of Operation

Recall that the goal of our device is to translate between analog speech signals and digital packets that are transported by computer networks. The current hardware platforms supported by Embedded Xinu excel at networking functionality, but lack peripherals for interacting with analog signals. Consequently, our design requires additional assistance from an external device designed for digital signal processing.

Figure 4.1 illustrates how these two devices work in unison to mimic the behavior of an embedded IP telephone.

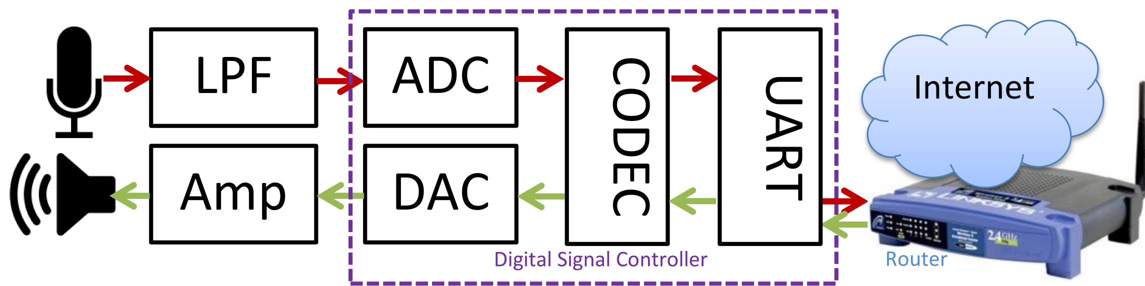

Figure 4.1: XinuPhone System Block Diagram

Transmitting audio begins by sampling speech signals captured by a microphone after a low-pass filter ensures that sample aliasing does not occur [14]. In Figure 4.1, the Digital Signal Controller (DSC) is the external device responsible for interfacing the embedded networking appliance with the analog world. An RS-232 serial interface connects these two components using a pair of Universal Asynchronous Receiver Transmitters (UARTs). The DSC also runs the software CODEC responsible for compressing the digital samples into a format that does not exceed the bandwidth provided by the serial link. As the samples transfer to the router, Embedded Xinu exchanges packets of audio data with other nodes via a computer network, such as the Internet.

Receiving audio works nearly the same way, but in reverse. Embedded Xinu reconstructs incoming network packets into a serial data stream. Next, the CODEC on the DSC expands incoming compressed samples. The DSC's digital to analog

converter (DAC) reconstructs an analog waveform from the uncompressed samples. Finally, an amplifier drives an output speaker generating an audible reproduction of the original speech signal.

4.2 Goals and Requirements

The original XinuPhone was designed with the following requirements in mind:

- Consists of low-cost, readily available components,
- Relies only on open-source software, and
- Achieves comparable performance to desktop applications despite using resource-constrained hardware.

As this platform evolved to include pedagogical goals we introduced ideologies from Project Nexos into the design specification:

- Designs that are simple enough to be understood by upper-division undergraduates, yet complex enough to apply theory that manifests in "real-world" applications;
- Adaptable equipment that promotes student experimentation and re-use throughout multiple courses;
- Mature components with long product life-cycles; and

- Support for freely available, Linux-compatible toolchains.

These specifications maximize the utility of the XinuPhone by helping ensure an institution's investment continues to see relevant and sufficient returns over time in a fact-paced, constantly evolving technological society.

Our revised design addresses many of the logistical shortcomings of Lund's implementation by adding further constraints:

- Compatibility with standard off-the-shelf microphones,
- Interrupt-driven asynchronous serial communications,
- Robustness against sample aliasing and analog reconstruction artifacts,
- All components available in through-hole packages, and
- Independence from a development kit with extraneous parts.

The following discussion highlights aspects of the XinuPhone's design that ensure satisfaction of the goals and requirements outlined above.

4.3 Hardware Design

The hardware selections for this platform feature low cost, easy assembly, and great versatility. The external audio interface simply consists of voltage regulators,

operational amplifiers, a serial level converter, and a digital signal controller; as well as various resistors, capacitors, and interconnects. Together, these components make up five distinct sub-systems: the embedded microprocessor, pre-amplification stage, anti-aliasing filter, output amplifier, and a serial transceiver. All elements are available in through-hole packages eliminating the need for the resources, skills, and patience demanded by surface mounted devices.

4.3.1 Microprocessor Selection

Shortcomings of the Freescale 68HC12 microcontroller used in the prototype design motivate the need for a different processor taking responsibility for the DSP related tasks. With similar needs in terms of instruction speed and peripheral support, we selected a similar class of embedded processor based on the successes reported in [36]. Specifically, the Microchip dsPIC33FJ64GP802 powers the current revision of the XinuPhone. Microchip's freely available, cross-platform Integrated Development Environment (IDE), MPLAB X, provides all of the necessary toolchain support for coding, compiling, and real-time debugging. MPLAB X is based on the popular NetBeans IDE, making it an ideal choice for educational usage.

Several features of the dsPIC make it an appropriate choice for this application. For example, its Direct Memory Access (DMA) engine allows for data transfer between RAM and many of the peripherals without having to interrupt the

main CPU. The XinuPhone leverages this feature to keep data flowing to the DAC while the CPU is busy executing the software CODEC. Peripheral Pin Select (PPS) is another feature on Microchip line of DSCs that benefits our design. PPS dynamically remaps input/output (I/O) pins to serve different peripheral needs based on a user-defined software configuration. This allows the same hardware to handle different tasks and easily adapt to changing needs–both key benefits supporting our versatility and longevity goals.

Readily available implementations of existing useful software algorithms are even further justifications for selecting the Microchip dsPIC. For example, Microchip publishes royalty-free distributions of audio CODECs including G.711 μ-law [16] currently used in the XinuPhone; as well as other popular choices: G.726A [17] and Speex [18]. Furthermore, the dsPIC DSP libraries support generic operations such as matrix multiplication and digital filtering. There are also built-in routines for advanced audio processing like noise suppression and echo cancellation [45]. These existing software building blocks make the dsPIC an ideal choice for an experimental telephony platform.

4.3.2 Pre-Amplification Stage

A pre-amplification (pre-amp) stage is necessary in order to support generic off-the-shelf microphone headsets. The pre-amp boosts single digit micro-scale

fluctuations in voltage up to line-level amplitudes for filtering and sampling. Lund's prototype does not contain a pre-amp stage; thus, it requires direct line-level input from a specialized microphone. The XinuPhone's pre-amp design (Figuere 4.2) features a powered input that works with any standard electret or dynamic microphone.

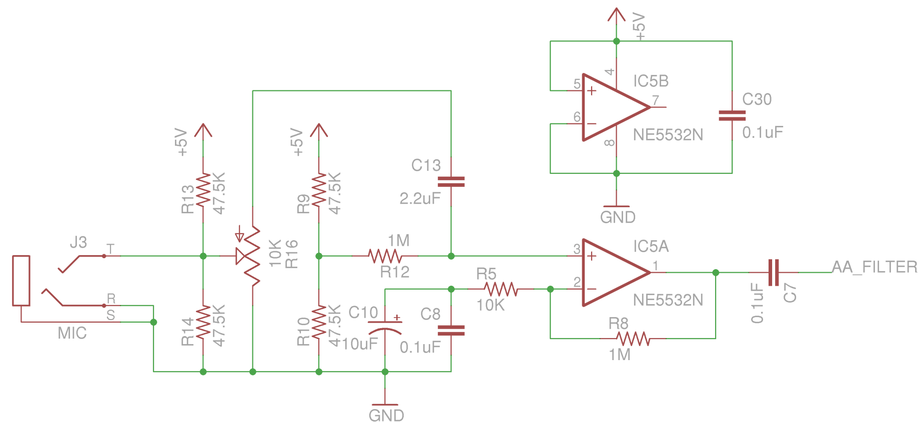

Figure 4.2: A Simple Op-Amp-based Pre-Amplification Stage

The pre-amp illustrated in Figure 4.2 employs a low-noise operational amplifier in a non-inverting configuration to amplify the input signal [46]. The remaining unutilized op-amp from the dual package is configured as a comparator–a minimum (zero) component layout without any floating inputs. A potentiometer on the non-inverting input gives the user control over the incoming signal level to prevent clipping and distortion at the output. Moreover, capacitors used in the signal path have N0G or X7R temperature coefficients in an effort to maintain decent audio quality [47]. Lastly, the capacitor on the output AC couples the signal

destined for the next stage effectively isolating the DC bias required for a single-supply op-amp.

4.3.3 Anti-Aliasing Filter

The addition of an anti-aliasing filter to the input signal is another improvement over the previous design. Before a continuous-time signal can be properly sampled, the high frequency components must be blocked; otherwise, the Shannon-Nyquist theorem may not be satisfied [14]. The XinuPhone includes a filter designed to pass the human vocal range and reject frequencies subject to aliasing at the industry standard sampling rate, $f_s = 8$ kHz. This design (Figure 4.3) applies the Sallen-Key topology [48].

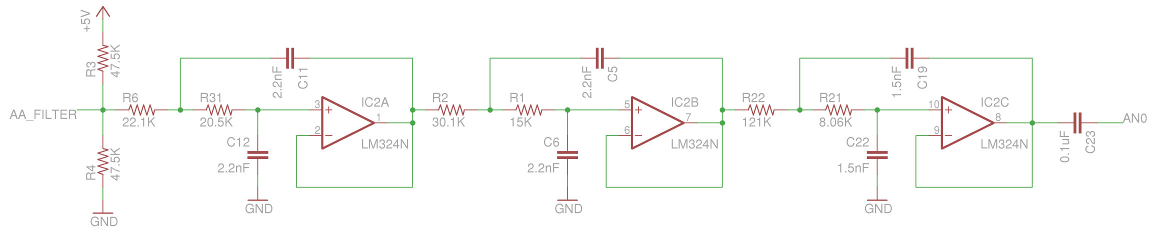

Figure 4.3: Sallen-Key Active Low-Pass Anti-Aliasing Filter

A filter cutoff frequency of $f_c = 3.4$ kHz harnesses the upper spectrum of human speech, yet provides ample room for roll-off before $\frac{f_s}{2}$. Equations 4.1 through 4.3 illustrate the computation of resistor values for the first stage of the 6th order Butterworth active low-pass anti-aliasing filter with capacitors chosen as C = 2.2 nF.

$$Q = \frac{1}{2\cos\left(\frac{\pi}{12}\right)} \tag{4.1}$$

$$R_1 = \frac{2Q}{2\pi f_c C} = 22\ k\Omega \tag{4.2}$$

$$R_2 = \frac{1}{2\pi f_c 2QC} = 20.5\ k\Omega \tag{4.3}$$

These calculations repeat for the remaining poles; cascading the stages produces a composite filter. Note that the high Q stages are last in sequence in order to reduce the risk of high gains saturating the filter hardware [49].

4.3.4 Output Amplifier

The dsPIC has a delta-sigma DAC on board for reconstructing the analog output wave. This peripheral includes filters for reducing images produced by the interpolation process as well as smoothing the modulated bit sequence into an analog signal [19]. Combined, these hardware features provide a superior reconstruction as compared to the basic first-order resistor-capacitor (RC) filter in Lund's design.

Output from the dsPIC is available as positive and negative halves of of the

composite signal on two separate pins. Consequently, the XinuPhone uses an external summing amplifier for combining the output signals (Figure 4.4). A second amplifier (LM386) protects the dsPIC from current overdraw when driving headphones or an external speaker. Output volume level can be adjusted by limiting the input to the LM386 with a potentiometer.

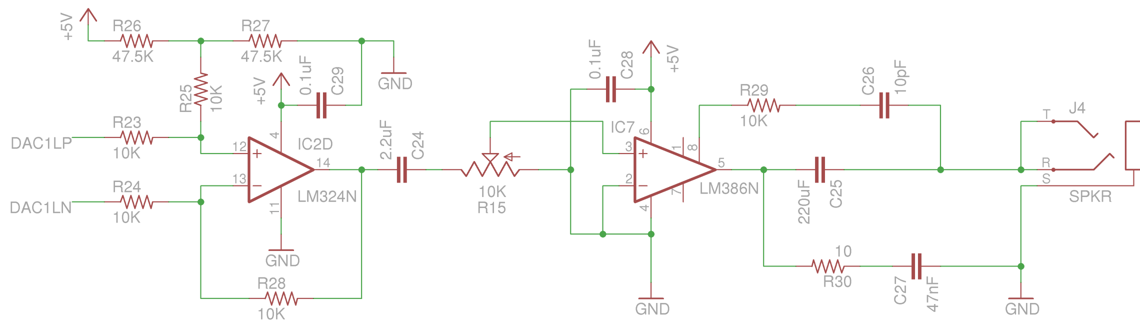

Figure 4.4: Output Amplifier for Driving Speakers/Headphones

4.3.5 Serial Transceiver

Integrated microelectronics typically interface with other chips using digital logic signals represented by positive voltages in the 3 V to 5 V range. On the other hand, RS-232 compliant serial devices handle up to ±25 V short-circuit to ground voltages [9]. The XinuPhone features an on-board level converter for interfacing the more sensitive dsPIC UART with other RS-232 devices (Figure 4.5). Therefore, the XinuPhone audio processor can safely communicate with a modified networking appliance or even a regular PC (for testing and troubleshooting).

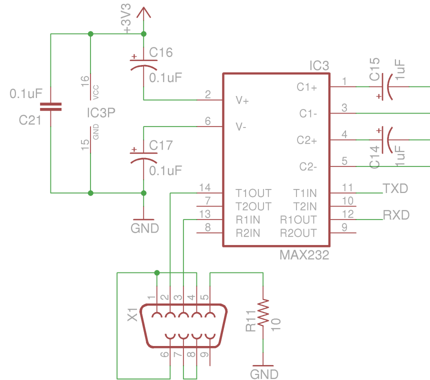

Figure 4.5: Charge-Pump Serial Level Converter

4.4 Firmware Design

The Microchip dsPIC hosts a highly-customized firmware application for processing speech and communicating with another serial device. We divide the code describing this software into two main categories: configuration code that runs once at power-on and run-time operations that repeat indefinitely. In this section, we also discuss some the challenges that arise when scheduling multiple tasks with strict timing deadlines that may lead to degraded performance if missed.

4.4.1 Device Configuration

Embedded processors usually boast a bevy of peripheral devices each with numerous configurable options. The system designer must ensure any default values are appropriately overridden with pertinent settings. The XinuPhone is no

exception, requiring careful consideration of how the timer, ADC, DAC, UART, and DMA peripherals are customized during startup.

In any signal processing application, accurate timing is critical for proper conversion between analog and digital representations. The XinuPhone timing subsystem delivers multiple clock signals to various features of the DSC, which are all derived from a single 16.384 MHz external high-speed crystal oscillator. The dsPIC peripherals have the following requirements:

- 8,000 Hz timer interrupt for ADC
- 2.048 MHz clock to DAC (256x oversampling)
- 115,200 bps UART baud rate

Furthermore, the dsPIC imposes additional constraints on intermediate frequencies in the clocking subsystem:

- High speed primary oscillators operate above 10 MHz.
- Phase-Locked Loop (PLL) input must be 0.8-8.0 MHz.
- PLL output before post-scalar must be 100-200 MHz.
- The device operating frequency must be 12.5-80 MHz.

Several configuration registers are responsible for controlling the clock scalars that

manipulate the dsPIC's timing characteristics. Figure 4.6 shows a configuration that meets the peripheral demands of the XinuPhone as well as the hardware constraints of the DSC.

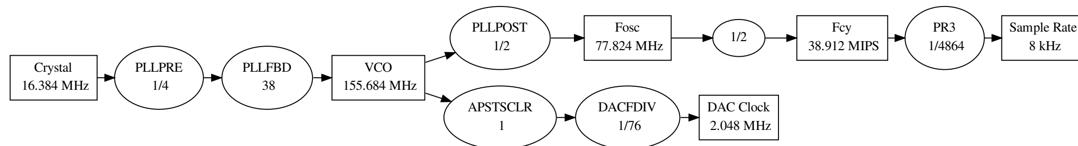

Figure 4.6: Timer Register and Clock Scalar Configuration

The XinuPhone relies on DMA for shuffling data from DPSRAM (dual-ported SRAM) to the DAC. During startup we associate DMA channel 0 with the DAC by mapping appropriate buffer locations and interrupt triggers. The DMA initialization also configures channel 0 for one-shot mode with automatic buffer swapping disabled. At first, this configuration choice may seem counterintuitive for an application that operates continuously with two buffers. However, it is necessary to manually swap buffers and then re-initiate a transfer so the firmware can enforce that the CODEC finishes processing a block of data before manipulating the buffer addressing and sending data to the DAC. In testing, we found the built-in continuous "ping-pong" mode (automatic double buffer swapping) allows a race condition to occur.

Power-on initialization also configures the DSP peripherals. Many of these options, such as the bit-depth and voltage references, are self-explanatory. Some of

```c
void initDAC(void)
{
  DAC1DFLT = 0x8000;              // set default value to the midpoint
  DAC1STATbits.LOEN = 1;          // enable left channel output
  DAC1STATbits.LITYPE = 1;        // interrupt when left ch FIFO is empty
  DAC1CONbits.DACFDIV = 75;       // divide ACLK (Fvco) by 76 (DACFDIV + 1)
  DAC1CONbits.FORM = 1;           // data format is signed integer
  IFS4bits.DAC1LIF = 0;           // clear interrupt flag
  //IEC4bits.DAC1LIE = 0;         // enable interrupt
  DAC1CONbits.DACEN = 1;          // enable DAC
} // end initDAC
```

Figure 4.7: Code for Initializing DAC Peripheral

the other non-obvious steps are for assigning the internal sampling trigger to Timer3 (8 kHz sample rate) and format of the data representation to

$$ssss \quad sddd \quad dddd \quad dddd \tag{4.4}$$

where an '*s*' represents a sign bit and '*d*' a data bit. Figure 4.7 lists a code example for the DAC initialization sequence. The default value register (`DAC1DFLT`) controls what voltage the DAC should generate in the event of an underrun. Line 9 is intentionally commented out—during normal operation DMA intercepts the interrupt so a separate trigger for the CPU is undesired. Nonetheless, a CPU interrupt can be useful for troubleshooting and benchmarking.

Lastly, the UART must be setup with a serial data format that agrees with the other endpoint (usually an embedded router appliance). The XinuPhone operates at 115,200 baud with 8 data bits, no parity, and 1 stop bit (8N1). The choice for baud rate is primarily dictated by hardware limitations elsewhere in the

lab infrastructure. For example, the ADM202 RS-232 line drivers and serial port annexes are limited to this maximum bandwidth. The baud rate is set by loading the baud rate generator register (`U1BRG`) with a value given by

$$\texttt{UxBRG} = \frac{F_{cy}}{4 * \texttt{BAUD}} - 1 \tag{4.5}$$

where F_{cy} is the frequency of the instruction clock as shown in Figure 4.6. Optionally, the UART can run in loopback mode where the transmitter is internally connected back to the receiver. Again, this is useful for benchmarking, but should be disabled for regular interoperation with a router.

4.4.2 Run-time Operations

Run-time operations execute repeatedly in an infinite `while` loop whenever the DSC is powered on. In addition to code found in `main`, the firmware implements run-time behaviors in the form of Interrupt Service Routines (ISRs). The XinuPhone firmware is organized by how data moves between the various hardware peripherals. Consequently, a single ISR may combine many of the following run-time tasks: sampling, CODEC compression, CODEC expansion, serial transmit, serial receive, and reconstruction. Working together, these software elements transform an otherwise idle piece of hardware into a functional audio processor.

```
1  void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void)
2  {
3    short adcsample, utxbyte;
4
5    adcsample = ADC1BUF0 << 4;                    // convert from 12 to 16 bit
6    ulaw_compress(1, &adcsample, &utxbyte);      // G.711 compression
7
8    while(1 == U1STAbits.UTXBF)
9      ; // wait for open spot in UART TX FIFO
10   U1TXREG = utxbyte;                            // write to UART
11
12   IFS0bits.AD1IF = 0;                           // clear interrupt flag
13 }
```

Figure 4.8: ADC Interrupt Service Routine

Locally sampled data enters the system at a precise rate in relatively small units (single samples). Therefore, it makes most sense to individually process each sample and send it on its way as quickly as possible in order to minimize overall delay. The ADC interrupt (Figure 4.8) is responsible for compressing samples with the software CODEC and depositing them in the UART transmit buffer each time a conversion completes. Lines 8-9 block the routine from proceeding if there is no room in the UART's transmit buffer. Theoretically, this condition should never occur—data enters at a fixed 8 kbps and is capable of exiting at 115.2 kbps. Should the system experience temporary timing issues, the 4 word UART FIFO also protects against data loss.

Processing data received from the router is slightly more complicated. This data arrives as packets (grouped samples) so naturally it makes most sense to manipulate these chunks block-wise. The UART receive interrupt collects a packet of data into an incoming buffer while inspecting each sample for control sequences.

When the system falls behind it is possible for the UART RX FIFO to contain multiple samples, so for each interrupt trigger the ISR loops until the FIFO is empty. This design choice allows the system to recover from missed soft deadlines when certain events occur off schedule. After an entire packet arrives, the ISR sets a global flag indicating data is ready for processing.

The `main` routine contains the infinite loop that runs whenever the CPU is not busy servicing an interrupt. This code segment (Figure 4.9) continuously polls the flag set the by UART indicating that a complete incoming packet has arrived. Four buffers interact with the CODEC expansion process: two for incoming (compressed) samples and two for outgoing (expanded) samples. This double buffering technique allows the CODEC to process a block of samples while peripherals are reading from or writing to the opposite buffer. Microchip refers to these size two circular buffer pools as *ping-pong* buffers. Global toggle flags for each peripheral (UART & DAC) keep track of which buffer is read or write active. The output buffers that the CODEC writes to are allocated in a special region of RAM that coexists on a separate bus for DMA transfers. When the CODEC finishes expanding a packet, the `main` routine manually initiates a DMA transfer to the DAC. At this point, not all samples have been reconstructed; however, the CPU can continue to run the CODEC on the next batch of data using the opposite buffer.

```
1  while (1) // run indefinitely
2  {
3    if (expandFlag)                            // wait for full packet
4    {
5      if(!dacToggle)                           // set destintion buffer
6        dacptr = dacBufA;
7      else
8        dacptr = dacBufB;
9
10     ulaw_expand(BUF_SIZE, urxptr, dacptr);   // decompress samples
11
12     while (DMA0CONbits.CHEN)
13       ; // wait for previous transfer to complete
14
15     if (!dacToggle)                          // reconfigure DMA buffer
16       DMA0STA = __builtin_dmaoffset(dacBufA);
17     else
18       DMA0STA = __builtin_dmaoffset(dacBufB);
19
20     DMA0CONbits.CHEN = 1;                    // enable DMA0
21     DMA0REQbits.FORCE = 1;                   // init DMA transfer
22     dacToggle ^= 1;                          // swap ping-pong buffers
23     expandFlag = 0;                          // clear expansion flag
24   }
25 }
```

Figure 4.9: Repeated Segment of `main` Routine

4.4.3 Task Prioritization

As previously discussed, the XinuPhone continuously executes several tasks during normal operation. These roles can be summarized as

- Starting ADC conversion at regular intervals (f_s),
- Compressing and transmitting samples,
- Buffering data received by the serial port,
- Expanding incoming samples, and
- Updating the DAC.

In a real-time system, such as an IP telephone, various tasks must appear to run simultaneously, albeit they actually execute individually in series. Correctly establishing priorities such that these tasks correctly yield to one another in accordance with timing deadlines is an important design consideration for this type of application. In an audio device, missed deadlines often manifest as artifacts and distortion, which may ultimately degrade the user's perception of performance.

In the previous section, we alluded to characteristics of the data flow between the router and the DSC that determine when and how many samples the CODEC should process at once. Task priority assignment must be consistent with the data flow and corresponding resource consumption. Figure 4.10 displays a snapshot of data passing between the DSC and the router during typical operations.

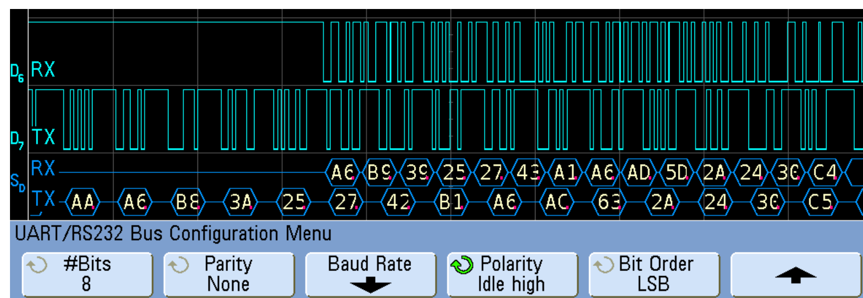

Figure 4.10: Serial Data Flow Between DSC and Router

Data leaves the DSC (TX) as single spaced out samples so compression must be prioritized to occur on schedule in the small gaps between serial bytes. Packets of samples arriving from the router (RX) are already in compact blocks so the idle

time presents a more elastic window for performing block-wise expansion as a lower priority task.

Unlike the router, which hosts Embedded Xinu, the DSC has no operating system. The firmware application runs on bare-metal so there are no built-in software constructs for scheduling or switching context. Scheduling is dependent on priorities assigned to the hardware peripheral interrupt vectors and tasks that can be offloaded from the CPU. Most importantly, the ADC must start conversions and the DAC must update on a precise schedule. Tasks closely related to hardware interacting with analog signals tend to have more strict deadlines with significant influence on performance. The XinuPhone DSC is configured in such a way that these operations do not rely on the CPU; therefore, they cannot be interrupted. Internal timer triggers and DMA make this possible.

Of the tasks that do rely on interrupts, data flowing out of the DSC receives highest priority. The packet-sized receive buffer allows more elasticity; consequently, it yields to transmission. Finally, CODEC expansion assumes least priority. It is not critical when the CPU performs this operation so long as it finishes an entire block of data sometime before the next block arrives completely.

Complex timing interactions between external events, hardware, and software present additional challenges when debugging and verifying correct operation of embedded systems. Print statement, single-step, and breakpoint

troubleshooting tactics often do not adequately describe the system state. Luckily, oscilloscopes and logic analyzers excel at capturing time sensitive data at normal execution speed. Throughout the development process, it is useful to acquire state changes of general purpose input/output (GPIO) pins to track software behavior. Figure 4.11 illustrates an example where the ADC and UART routines are correctly taking priority over the CODEC expansion process with ample overhead.

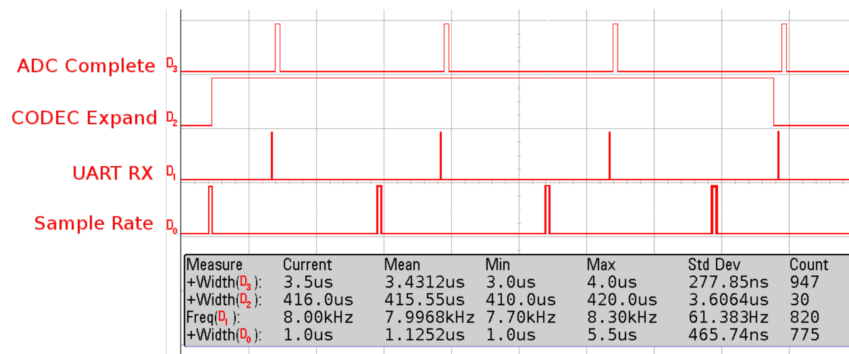

Figure 4.11: Verifying Software Run-Time Behavior with a Logic Analyzer

Each signal represents an ISR or software subroutine: UART RX (D_0), f_s Timer (D_1), G.711 μ-law CODEC expansion (D_2), and ADC conversion complete (D_3). Statistics collected from the GPIO pins approximate the worst, average, and best case execution times of each of the subroutines. This data is also useful for resource budgeting if a particular task is frequently missing deadlines.

Recall from Figure 4.1 that we have two opposing information flows competing for CPU resources: sampling audio to send out of the system and reconstructing audio received from the remote host. The user expects this all to

occur transparently, but at a more granular level these processes take turns. Figure 4.11 shows each of the higher priority tasks taking precedence over the tasks with more lenient deadlines. The lowest priority task, CODEC expansion, takes an average 415.55 μs (including any preemptive tasks). In the illustrated scenario where samples are processed in size 10 blocks, a deadline occurs every

$$10 * T_s = 10 * \frac{1}{f_s} = 10 * \frac{1}{8000} = 1.25\,\text{ms}. \tag{4.6}$$

Since the CPU is only loaded approximately 33%, there is sufficient overhead for the system to reliably make its deadlines. Furthermore, this extra capacity indicates that the same system could handle a more computationally intense CODEC for additional bandwidth savings or the overall clock speed could be reduced to save energy. On the other hand, if we were to observe any task with a duty cycle close to 100%, that is an indication the priority scheme or algorithm complexity needs reconsideration because the system is likely missing deadlines.

CHAPTER 5

A Kernel-based Approach to RTP

The inability to cope with non-ideal network conditions is a major drawback to the original embedded VoIP system. To its credit, this was beyond the design scope; nevertheless, added support for interoperation on production networks is a natural extension to the existing feature set. In this chapter, we propose a framework based on the Real-Time Transport Protocol (RTP) that allows an embedded VoIP application to maintain acceptable levels of quality while operating in disruptive network conditions. As a kernel device, this implementation distinguishes itself from its application library counterparts by leveraging existing support already provided by the Embedded Xinu operating system. The following sections describe the overall design, features above and beyond the RTP specification, and an example application that uses Xinu RTP for testing VoIP listening quality.

5.1 Design Approach

RTP is different from some other protocols in that the RFC lacks specific directives on how to interpret the mandated header information. This section tours Xinu RTP's main components that fill gaps left by the specification. Additionally, we

describe how it provides multiple operation modes to support either the bare-minimum RFC requirements or extensions that we believe to be universally applicable to simple real-time software suited for Embedded Xinu.

5.1.1 Integration into Embedded Xinu

RTP-based applications typically implement the protocol directly or rely on a pre-existing library (see Chapter 3). The state information describing a real-time stream tends to integrate with the application design so the creators of RTP promote protocol libraries [50]. Within Embedded Xinu, we believe a kernel-based approach is more logical. For a particular payload type, we can make certain assumptions about the size and ordering of the data demanded by the application and then offer that functionality by means of a kernel device. Furthermore, this allows reuse of the existing Xinu device API, interprocess communication framework, and synchronization primitives for enforcing real-time behavior.

Separating Xinu RTP from the application puts it in a unique position between levels in the traditional layered network model. Xinu RTP sits above layer 4 as it relies on UDP for some transport features. However, as a modular abstraction away from any specific application, Xinu RTP resides below layer 5 as a framework reusable by user-space programs. The main disadvantage to this approach is how the Xinu device API restricts an application's access to information

inferred from the RTP metadata. We argue that this encapsulation trade-off is reasonable because advanced multi-party applications obtain this information from out-of-band means (such as SIP) and simple point to point conversations are possible by omitting access (as demonstrated by our sample application).

Applications use Xinu RTP just like other high level protocols provided by the network stack. First, a user-space application must call `rtpAlloc` to obtain an RTP device from a pre-allocated device pool. Then, it may interact with the device using Xinu's standardized API:

- `rtpOpen` initializes the device's control block, allocates underlying transport devices, and spawns additional threads for processing incoming packets. Also, the open function relays address/port information to UDP and configures the RTP device for the specified operation mode and RTP profile.
- `rtpClose` tears down an existing RTP session by killing helper threads, closing underlying devices, resetting the control block, and returning the device back to the available pool.
- `rtpRead` copies up to the specified number of bytes of payload data into the application's buffer.
- `rtpWrite` fragments outgoing data according to an RTP profile. The helper

function `rtpSend` prepends the RTP header and relays the packet to Xinu's network stack.

- `rtpControl` allows the application to set the profile specification and toggle basic flow control on/off.

Xinu RTP uses the same API to pass data to and from the underlying UDP device as if it were the application itself so it is easy to integrate into existing applications.

5.1.2 Basic Mode

Xinu RTP implements the core features required by the RFC at the boundary between the RTP device and the underlying network transport. Xinu RTP's *basic mode* satisfies these constraints without further modifications to the packet stream. This mode serves as the baseline for any performance gain achieved by enhancements that go beyond the RFC's requirements. Essentially, this mode manipulates RTP packet headers, maintains bookkeeping in the device control block, and enforces the minimum constraints mandated by an RTP profile. For outgoing packets, `rtpSend` handles these responsibilities, likewise, `rtpRecv` for incoming packets.

The system call `rtpSend` combines payload data from the application with an RTP header constructed from control information. An RTP header consists of

the protcol version, a payload type code, sequence number, timestamp, and a unique identifier for the source (SSRC). Sequencing and timestamp values are maintained within the device, freeing the application from this responsibility. Once combined, this information becomes the new payload for a UDP packet.

A separate thread, `rtpRecv`, processes packets inbound from UDP. The receive thread validates the header against a series of checks and ensures only packets of the configured payload type enter the incoming packet queue. Therefore, the application may safely assume packets read from this device belong to the stream without further analysis. Xinu RTP does not handle packets with payload padding or header extensions. This design elects to sacrifice interoperability with other devices supporting these features in favor of minimizing the number of dynamic memory allocations. In basic mode, `rtpRecv` queues packets in order of arrival and immediately makes them available to the application calling `rtpRead`. Most of the RTP header is effectively discarded and serves no purpose to minimally comply with the RFC.

5.1.3 Sequence Mode

One might expect that a protocol designed around sequence numbers and timestamps would enforce some kind of numeric organization and notion of elapsed time. The RTP specification mandates no such behavior; nonetheless, Xinu RTP

leverages the existing RTP packet structure to offer enhanced features. In addition to the header processing and profile compliance offered by basic mode, Xinu RTP's *sequence mode* provides additional guarantees:

- Duplicate packets do not add redundant data to the receiver's byte stream.
- The receiver application reads bytes from the RTP device in the same order the sender application wrote them.
- `rtpRead` returns padding bytes in place of any payload that does not arrive by a specified deadline. As a corollary, the receiver application's stream is time-aligned with the sender's although it may be missing data.
- Payload data that eventually arrives past its deadline does not corrupt the output stream.
- In the event the receiver gets significantly out of synchronization with the sender, the system is able to gracefully recover.

None of the enhanced features affect the behavior of the sender; the receiver handles all of the intelligent packet processing.

Compensating for variations in network latency is one of the primary motivations for using sequence mode. When the transit time is inconsistent, the destination host likely receives packets in an order different from which they were

sent. The classic solution to this problem uses a "de-jitter" buffer on the receiver to queue packets in order. As a packet arrives, simply insert it into the queue at the proper location to maintain numeric order according to the sequence number. If packets have infinite time to arrive, this solution is trivial. However, in a real-time system the additional constraints imposed by getting sequential data to the application on schedule add further complications.

Obviously, the system cannot pause to keep time from passing in order to wait for delayed packets. However, it can introduce additional fixed delay up front so there is elasticity to handle latency variations. Xinu RTP's sequence mode accommodates this flexibility by partially filling the de-jitter buffer before releasing any samples to the application. This buffer serves two purposes: a staging area to re-order packets and holding area to queue packets before the application is ready for them. The latter is necessary because the application demands packets periodically; however, due to packet delay variation, they tend to arrive aperiodically.

5.2 Implementation of Enhanced Features

Xinu RTP implements the sequence mode guarantees using an intermediate thread that arbitrates communications between the RTP receive thread and the application. In this section, we explain how this scheme operates and motivate the

need for it by examining a failed approach. Lastly, we annotate how Xinu RTP addresses other issues encountered that the RTP specification does not account for.

5.2.1 Additions to Support Ordered Queuing

In basic mode, the receive thread communicates directly with the application thread. Sequence mode uses an additional thread between these two parties to monitor the state of the packet queue. For simplicity, we refer to each thread by the respective C function it executes: `rtpRecv`—a bridge between the network stack and RTP, `rtpWarden`—the packet queue monitor, and `rtpRead`—called by the application to request data (Figure 5.1).

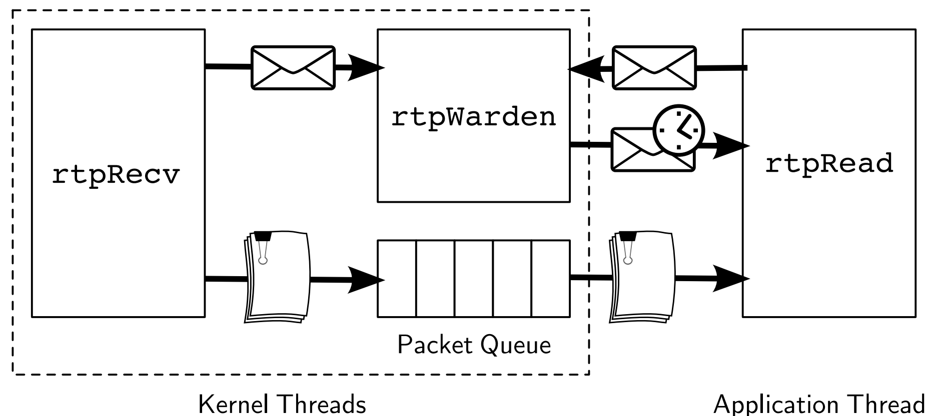

Figure 5.1: Thread Relationship for Handling Incoming Packets

Decisions concerning packet reordering rely on state information about the next *expected sequence* number. In other words, the packet that the system would

like to deliver to the application as soon as it becomes available. The RFC recommends randomizing the initial sequence number for security, so the receiver initializes its expectation to the sequence number identifying the first valid packet belonging to a particular session. The expected sequence number increases monotonically by one each time the application successfully retrieves a packet's payload or gives up on the data due to a missed deadline. The RTP control block stores this state information, making it accessible to all three threads.

The receive thread makes blocking calls to `udpRead`; it waits on a semaphore until a packet is available. Once signaled, `rtpRecv` performs the same validation checks as in basic mode. Additionally, it checks the incoming packet's sequence number against slots currently allocated for expected packets. Any packet preceding the current expected sequence is late and therefore discarded. Next, packets enter the queue in a specific slot corresponding to sequence numbers instead of by order of arrival. Finally, `rtpRecv` sends a message to alert the `rtpWarden` that new data is available.

The `rtpWarden` monitors for a match between the expected sequence number and the packet in the first queue position. Initially, the `rtpWarden` blocks until it receives an inter-thread communication message indicating a match is possible. Therefore, this thread does not proceed until a new packet arrives or the expected sequence number changes. In either case, the new expectation *could* be satisfied,

but not necessarily, so the `rtpWarden` must check if the first queue position contains valid data. If it does encounter the desired packet, it sends a message to `rtpRead` (the application), which is waiting to dequeue the data. Otherwise the `rtpWarden` resumes waiting for another message.

An application calls `rtpRead` when it wishes to obtain a block of bytes from the RTP device. Since the `rtpWarden` sends a message when the correct packet is ready, `rtpRead` simply waits for a message to arrive. Hopefully, the packet arrives on schedule, `rtpRead` dequeues the packet and copies its payload to the application buffer. Alternatively, a message does not arrive within the timeout period. In this case, `rtpRead` substitutes padding data (zeros by default) in place of the missing packet. Either way, `rtpRead` sends a message back to the `rtpWarden` indicating the expected sequence has changed.

5.2.2 The Need for Multiple Threads

At first, it was not obvious a moderator thread between the network stack and the application would be necessary. Indeed, it is possible for the receive thread to communicate directly with the application as in basic mode. This implies the receive thread must also monitor for the next desired packet's arrival because each call to `rtpRead` must return data (or padding). Consolidating roles into the receive thread turned out to be a poor design choice because it cannot make blocking calls

to `udpRead` and still react to external changes to the expected sequence number. We attempted to resolve that issue with a bidirectional messaging system between the outer two threads where either could timeout (and unblock). The resulting system was highly unpredictable. It did not seem possible to predict the state or outcome given a packet series, a timeout threshold, and arrival times. The other combinations of having each thread block or poll exhibit deadlock or fail to satisfy the proposed guarantees.

The introduction of a third thread for monitoring the queue allows for an elegant, asynchronous solution using Embedded Xinu's built-in inter-thread communication functions. First consider a normal scenario:

1. `udpRead` signals `rtpRecv` to queue the packet,
2. `rtpRecv` messages the `rtpWarden` that a new packet is in the queue,
3. the `rtpWarden` messages `rtpRead` (the application) to copy the available data, and finally
4. `rtpRead` advances the expectation and sends a message back to the `rtpWarden` to continue looking for the next packet.

Data propagates through the system consistently; each event has a deterministic reaction. When data does not arrive in sequence, each half of the system continues to operate independently: `rtpRecv` queues packets as long as there is room in the

buffer and `rtpRead` copies data to the application as soon as it becomes available. Consequently, Xinu RTP can handle transient spikes or lulls in traffic, yet still return to equilibrium when normal conditions resume.

5.2.3 Packet Queue

The queue for incoming packets is the core data structure enabling much of RTP's desired functionality. Xinu RTP implements a circular buffer using index offsets relative to the expected sequence number. The buffer head always represents the next expected sequence so the other threads have direct access to it for quick retrieval. The placement of a packet with sequence S_{packet} is based on a *virtual* sequence number, $S_{virtual}$, given by

$$S_{virtual} = ((S_{packet} - S_{expected}) + 65536) \bmod 65536. \tag{5.1}$$

We introduce the virtual sequence number representation to solve the issue of comparing sequences that wrap around the max value. The RTP header allocates two bytes to the sequence number so naturally they repeat after 65535. Comparing "before" and "after" a certain value does not necessary correspond to "greater than" and "less than" operations in the mathematical sense because of the circular values. We avoid complicated boundary conditions by translating all of the sequence

numbers to a frame of reference relative to the expected sequence, which effectively unwraps the sequence to a linear region. Using this virtual representation, the comparisons are straightforward.

A fixed-size statically allocated array is a sensible choice for a buffer that is managed by a resource-constrained system. As a result, the range of sequence numbers the system can accept any given time is limited by this buffer size. Effectively this creates a sliding window of packets past the expected sequence number. This is similar to the mechanism TCP uses for packet acknowledgments (ACKs) [23]. As the application retrieves data from the buffer, the expectation increments and the head of the queue advances one position, sliding the window of acceptable packets forward.

Before queuing a packet, `rtpRecv` checks whether the packet's sequence is within the sliding window. Packets outside the window are either too early or already past deadline. With a system of circular sequence numbers, this distinction is somewhat arbitrary. Xinu RTP operates under the assumptions that packets are more likely to arrive late than early and early packets arrive relatively close to the current sequence. We define *on-time* packets as those that fit within the sliding window. In terms of the virtual sequence and the buffer size N_{max}, these packets fall within 0 and $N_{max} - 1$. Xinu RTP considers packets *early* if the virtual sequence is greater than or equal to N_{max} but less than $2N_{max}$. Any other packet is considered

late and therefore gets dropped as the system has already moved on without it. Note that without virtual sequence numbers these comparisons would not fall into a consistent non-circular range there by greatly complicating the validation stage.

Once determined in range, `rtpRecv` inserts a packet with sequence number S_{packet} at buffer index I_{packet} determined by

$$I_{packet} = (I_{head} + S_{virtual}) \bmod N_{max}. \tag{5.2}$$

This method leaves gaps for packets that have not arrived yet so recording occurs without the overhead of moving data around. Advancing to the next sequence is as simple as incrementing the index of the buffer head, I_{head}, modulo the buffer size. Then, the process repeats when a new packet arrives.

5.2.4 Other Enhancements

Xinu RTP features more enhancements that, again while not mandated by the RFC, offer significant utility to real-time applications. First, it offers duplicate packet rejection. If the transport network replicates a packet in transit, the receiver may experience time mis-alignment and an artifact when redundant bytes enter the stream. The queuing system in `rtpRecv` detects duplicate packets when a buffer slot

required for insertion is already in use. Dropping the duplicate packet prevents it from corrupting the stream.

Next, two fault detection mechanisms prevent Xinu RTP from experiencing unrecoverable failure modes. An *early* packet, as defined above, indicates the system may have fallen significantly behind. Instead of just dropping the packet, Xinu RTP interprets this information as a signal to cut its losses and catch-up with the sender. In this scenario, the expectation immediately advances to the early packet's sequence number in an effort to realign itself with the current timeline.

Another way Xinu RTP recovers from faults is by keeping track of successive timeouts. If `rtpRead` consistently fails to find expected data in the buffer, that is a good indication the receiver is out of synchronization with the sender. When `rtpRead` exceeds a threshold of consecutive read failures, a reset condition occurs. Then, the next packet arrival sets the expected sequence value and the buffer refills so the device can continue on as normal. Although this scenario may result in an undesirable artifacts from the application's perspective, it is generally better than the likely alternative that the connection is lost entirely.

Lastly, Xinu RTP provides a primitive flow control mechanism. Normally, RTP does not handle flow control directly—the application generally assumes that responsibility. We chose to integrate basic flow control into the driver for supporting streams of pre-recorded data. Xinu RTP accepts a byte stream from an application

and regulates the period between sent packets according to the selected profile. This feature is disabled by default, but can easily be enabled via the API `control` function.

5.3 Sample Application

To demonstrate the use of Xinu RTP, we offer a sample application (Appendix A) for sending audio files between two instances of Embedded Xinu. From the Xinu shell, invoke the application with `--help` for a detailed list of options. In Chapter 6, we use this application to transport VoIP data between routers to evaluate Xinu RTP. But first, we discuss how to customize the device to suite the needs of a particular application, in this case Internet telephony.

First, create a profile that matches the application's payload type. The ITU predefines several of these for audio/video use in RFC 3551 [25]. The dataset used in our experiment contains 16-bit linear PCM WAV (little-endian) format audio. Add the profile entry to the table in `rtpControl.c`. The sample entry contains

- `L16LE`, a name designating the profile;
- `RTP_PT_P16LE_8K`, a dynamic code for this undefined format;
- 20 milliseconds of audio per packet;
- 320 bytes per packet; and an

- 8000 hertz sample rate.

An appropriate profile specification allows the RTP device to correctly fragment the stream into packets and properly code the packet headers.

Next, pick an appropriate buffer size for the application. A larger buffer is more elastic in terms of delay variation, but requires a longer initial fixed latency to allow compensation for fluctuations later on. During a phone call, human listeners are very sensitive to delay variation (jitter) but are relatively tolerant of fixed latencies. In fact, even the most particular listeners accept one-way latencies in the range of 150 ms to 200 ms before perceiving a conversation as half-duplex, or "walkie-talkie" like [51]. We exploit this phenomenon when configuring RTP to maximize the device's ability to reorder packets without disrupting the conversation.

A simple delay budget helps apply this knowledge to the RTP device. Consider three main components of total latency:

- Processing Delay—overhead from the CODEC and network stack,
- Propagation Delay—time to travel through the transmission medium, and
- Queuing Delay—additional waiting time from buffering.

To ensure best performance, the sum of all three components must not exceed the budget. At this scale, processing delay is negligible—the other two components

dominate. For regional calls across the Internet, let us allow 50 ms for one-way transit. Therefore with a VoIP payload of 20 ms per packet, we can still queue an average of 5 packets without disrupting the conversation. Xinu RTP generates the fixed delay by filling the queue half-way, so a buffer size 10 best satisfies the constraints. This parameter may be tuned as desired in the Embedded Xinu `rtp.h` header file. Although Xinu RTP is relatively application agnostic, adjusting a few parameters helps boost performance for a specific problem domain.

This chapter presented a lightweight real-time framework suitable for resource-constrained systems as an integral part of the Embedded Xinu operating system. Next we evaluate this framework using the example application that applies RTP to Internet telephony.

CHAPTER 6

Experimental Analysis

The objective of this thesis is to demonstrate a measurable improvement in voice quality using the proposed real-time framework relative to a transport mechanism used in the reference embedded VoIP system [40]. Before analyzing the application, we verify the functional correctness of Xinu RTP with a series of test cases. Then, we benchmark the performance of the two systems by transporting a standard collection of audio files through adverse network conditions and comparing the received data with the original. The test parameters, data set, and algorithm for analyzing results are all derived from industry standards published by the International Telecommunications Union.

6.1 Correctness Verification

First, we demonstrate the proper operation of Xinu RTP's underlying components before analyzing its performance in the context of a VoIP system. For this task, we use a suite of test cases, each focused on a specific feature or scenario (Table 6.1). The sender deterministically constructs out-of-order sequences that stress boundary

Table 6.1: RTP Test Suite

Test	Name	Sent	Expected
1	Normal	1,2,3,4,5	1,2,3,4,5
2	Reorder	1,4,3,5,2	1,2,3,4,5
3	Timeout	1,3,4,5,6	1,T,3,4,5,6
4	Drop Late	1,3,4,5,6,P,2	1,T,3,4,5,6
5	Buffer Fill	1,3,4,5,6,P,8,9,10,11,7	1,T,3,4,5,6,7,8,9,10,11
6	Overflow	1,2,3,4,5,. . . ,15	Drop Packets (No Flow Ctrl)
7	Duplicates	1,3,3,3,2	1,2,3
8	Late Wrap	1,2,0,65534,4,3	1,2,3,4
9	Early Reset	1,2,5,8,7,9	1,2,8,9
10	Early Wrap	65533,65534,0,4,3,5	65533,65534,4,5

P = Pause, T = Timeout

conditions in the receiver's ability to reconstruct the original data stream. All tests assume a packet queue of size 5 and pauses equal to an RTP timeout of 5000 ms.

The first test considers the system in its normal operation mode where packets arrive in order and on schedule. In addition to handling anomalies, we wish to ensure Xinu RTP does not harm perfectly good packet streams. The next three tests show the system can handle lost and out-of-order packets by reordering the input buffer, giving up on late packets, and dropping those which arrive post-deadline. Test 5 exhausts the queuing functions by filling the circular input buffer, flushing all packets, and then repeating the process once again. This shows the modular arithmetic for wrapping the queue is correct. Next, we test overrun by sending input to device faster than it can process data. The system gracefully drops packets it is not able to accept without error and resumes normal operation when the burst is over. Test 7 demonstrates Xinu RTP can recognize and reject duplicate

sequence numbers. Next, we test late arrivals on the boundary where sequence numbers wrap beyond their maximum value. This check also audits the bounds of the sliding window for dropped packets. Finally, we test the scenario where an early packet triggers a reset condition for the device to catch-up from falling significantly behind. The last two tests analyze this condition for regular and wrapped sequences.

This payload-agnostic collection of tests allows us to verify correct operation of the Xinu RTP device independently of any specific application. As new features are implemented this suite serves as a basis for regression testing to ensure problems are not introduced by additional components [52].

6.2 Metrics and Measurements

Our testing procedure leverages metrics and evaluation tools internationally recognized by industry leaders and regulatory agencies. Specifically, we directly use the ITU's Perceptual Evaluation of Speech Quality and Mean Opinion Score standards. Additionally, we manipulate testing parameters based on influential components identified in computational models designed to estimate performance quality. In this section, we review the relevant aspects of these publications.

Table 6.2: Mean Opinion Score Interpretations

Score	**Quality (MOS)**	**Listening Effort ($\mathbf{MOS}_{LE}$)**
5	Excellent	Complete relaxation possible; no effort required
4	Good	Attention necessary; no appreciable effort required
3	Fair	Moderate effort required
2	Poor	Considerable effort required
1	Bad	No meaning understood with any feasible effort

6.2.1 Mean Opinion Scores

A Mean Opinion Score (MOS) is a subjective measurement of voice quality perceived by a human listener, as per the ITU-T P.800 recommendation [53]. Results are calculated from an arithmetic average of users rating the call quality in a strictly controlled environment. Scores range from 1 (worst) to 5 (best), where values above 4 are generally considered *toll quality* by service providers. Mean Opinion Scores are costly and cumbersome to acquire. Consequently, several companion recommendations describe alternate ways to estimate a MOS value based on various objective measurements.

The ITU designates several modifiers to clarify the origin of a MOS figure. One may become easily confused by the myriad of choices derived from combinations of Listening Quality, Conversational Quality, and Talking Quality tests; Subjective, Objective, and Estimated models; and Wide or Narrow bandwidths. Unless otherwise noted, references to MOS in this document refer to narrowband (3.1 kHz) MOS-LQO (Mean Opinion Score-Listening Quality Objective) [54].

6.2.2 Perceptual Evaluation of Speech Quality

The Perceptual Evaluation of Speech Quality (PESQ) assessment is an objective measurement based on an analysis of audio recordings, as described in the ITU-T P.862 recommendation [55]. PESQ works by comparing a known *reference signal* with the result after this signal traverses the system under test, the *degraded signal*. We use the reference implementation of the PESQ algorithm provided in P.862 Annex A for this comparison.

PESQ is known to accurately test a variety of factors such as speech input levels, channel errors, environmental noise, variable delays (listening only), and time warping. Conversely, PESQ is not well-suited for loudness loss, delays in conversational tests, fixed delays, and talker echo. PESQ is appropriate for this experiment because we wish to manipulate delay and packet loss for a one-way transmission channel (non-conversational). In other words, we analyze the transmission of audio from point A to point B in an isolated channel without considering the interactions of audio returning from point B back to point A. Note that PESQ is not explicitly validated for evaluating packet loss with pulse-modulated audio. As the recommendation explains, PESQ tends to be more sensitive than humans to front-end temporal clipping (we may hear as missing words). Therefore, we interpret results from these tests with care by restricting

comparisons relative to other PESQ scores or treating MOS generalizations as worst-case.

The raw scores PESQ generates do not directly correspond to MOS. In fact, PESQ-MOS values range from -0.5 to 4.5 instead of 1 to 5. The function

$$y = 0.999 + \frac{4.999 - 0.999}{1 + e^{-1.4945x+4.6607}} \tag{6.1}$$

maps a raw PESQ-MOS value x to the corresponding MOS-LQO value y [56]. Note that as a result of this model the maximum MOS-LQO score for a perfect reconstruction is approximately 4.549 instead of 5. Using this translation, we can make comparisons between objective PESQ results and other subjective scores, such as the benchmark for toll quality audio.

Speech data used in this experiment for PESQ evaluation is available with the official algorithm release. These samples are the same audio segments used to validate third-party compliance with the ITU-T recommendation. Test audio contains pairs of sentences separated by silence spoken by male and female talkers. As per ITU-T recommendation, speech bursts occur in 1 to 3 second durations and represent 40% to 80% of the clip relative to silence intervals. Table 6.3 assigns each of the specific files with a test number referred to through the results section.

Table 6.3: Audio File Data Set

Test #	Sample Rate (Hz)	Filename
0	8000	`u_af1s01.wav`
1	8000	`u_af1s02.wav`
2	8000	`u_af1s03.wav`
3	8000	`u_am1s01.wav`
4	8000	`u_am1s02.wav`
5	8000	`u_am1s03.wav`

6.2.3 Determining Test Parameters

Instead of using objective or subjective assessments to determine a MOS directly, one can also estimate performance from characteristics of the underlying network. The E-model is one such computational model based on a rating factor R, intended for end-to-end transmission planning. The rating factor considers signal-to-noise ratio, fixed and variable delays, and packet loss [57]. MOS values can be derived from the rating factor; thus, these parameters are known to have an influence on user perception.

Similarly, the ITU proposes a network model specifically for multimedia transmission over IP networks [58]. Although this model is not specific to telephony it identifies three common factors affecting performance: overall delay, packet delay variation, and packet loss. Neither of these models are used directly in this experiment; however, we use them as a guide for which network conditions we can emulate to influence voice quality. By subjecting the proposed system to factors

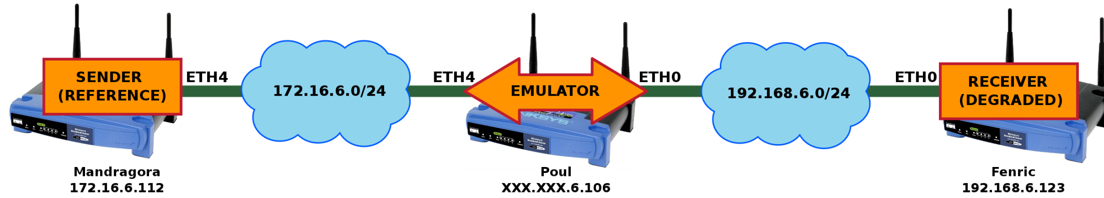

Figure 6.1: Network Topology for PESQ Testing

known to be detrimental to user satisfaction, we achieve confidence that it is indeed robust if it achieves higher scores than the status quo.

6.3 Experimental Setup

The testing environment consists of three routers connected via two Ethernet networks (Figure 6.1). The sender, `Mandragora`, contains the reference audio files locally in memory as part of its kernel image. `Mandragora` transmits a reference audio file to `Fenric`, the receiver. `Mandragora` cannot communicate with `Fenric` directly because they do not reside on the same network subnet. However, `Poul` is able to route packets between `Mandragora` and `Fenric` because it contains interfaces on both networks. Furthermore, `Poul` runs the Embedded Xinu Network Emulator, NetEmu, which allows stochastic manipulation of network conditions [59]. As `Poul` routes traffic from `Mandragora` to `Fenric`, NetEmu, drops and delays packets according to statistical test parameters. Once `Fenric` receives a complete set of degraded audio packets, it relays them to a workstation via Trivial File Transfer

Protocol (TFTP). Finally, the PESQ algorithm compares the reference file to the degraded audio and assigns a MOS-LQO.

In addition to physically connecting the routers' Ethernet interfaces as shown in Figure 6.1, Embedded Xinu requires some software configuration to model the illustrated scenario. First, bring up Ethernet interface ETH4 on `Mandragora`.

```
xsh@mandragora$ netup ETH4 172.16.6.112 255.255.255.0 0.0.0.0
ETH4 is 172.16.6.112 with netmask 255.255.255.0 (gateway: 0.0.0.0)
```

The default gateway is irrelevant because next we add a static route entry to inform `Mandragora` it can access `Fenric's` network via `Poul`.

```
xsh@mandragora$ route add 192.168.6.0 172.16.6.106 255.255.255.0 ETH4
xsh@mandragora$ route
Destination     Gateway         Mask            Interface
172.16.6.0      *               255.255.255.0   ETH4
default         0.0.0.0         0.0.0.0         ETH4
192.168.6.0     172.16.6.106    255.255.255.0   ETH4
```

On `Poul` bring up two Ethernet interfaces so the routing daemon can forward packets between two networks.

```
xsh@poul$ netup
ETH0 is 192.168.6.106 with netmask 255.255.255.0 (gateway: 192.168.6.50)
xsh@poul$ netup ETH4 172.16.6.106 255.255.255.0 0.0.0.0
ETH4 is 172.16.6.106 with netmask 255.255.255.0 (gateway: 0.0.0.0)
```

`Fenric's` default network configuration is usable directly; however, it is useful to add a static route back to `Mandragora's` network. This is not necessary for the PESQ experiment; nevertheless, it is handy for testing the network topology with ICMP Echo Request/Reply packets.

```
xsh@fenric$ netup
ETH0 is 192.168.6.123 with netmask 255.255.255.0 (gateway: 192.168.6.50)
xsh@fenric$ route add 172.16.6.0 192.168.6.106 255.255.255.0 ETH0
xsh@fenric$ route
Destination     Gateway         Mask            Interface
192.168.6.0     *               255.255.255.0   ETH0
default         192.168.6.50    0.0.0.0         ETH0
172.16.6.0      192.168.6.106   255.255.255.0   ETH0
```

If everything is configured correctly, `Mandragora` and `Fenric` should be able to `ping` each other with the built-in Xinu shell command.

The endpoint routers communicate the test audio files using a custom test application implemented as a shell command in Embedded Xinu (complete source code listed in Appendix A). On the source router, invoke the command as the sender (`-s`) with a test number corresponding to a reference file (Table 6.3). On the destination router, invoke the command as the receiver (`-r`) using the same test number. Common knowledge about the test number determines how many bytes the receiver should expect and what filename to use for the resulting file. By default, the receiver operates in a pass-through mode, where the RTP device does not influence the packet sequence. This mode mimics the behavior of the previous

generation embedded VoIP implementation. Adding the `--seq` switch to the `test` command enables the full-featured RTP device. The subsequent experiments examine the complete dataset in each mode: the default sets a baseline, then the enhanced mode demonstrates how much of an improvement Xinu RTP offers.

6.4 Test Results

This section presents the results from a collection of tests specific to VoIP performance. Each experiment represents a different type of traffic characteristic on an IP network. We observe how the legacy and proposed systems react to dropped packets and delay variation. In a VoIP application, these conditions translate to lost information, rearrangement of the audio waveform, and playback jitter. For simplicity, we refer to the legacy system as UDP, in reference to the highest layer protocol it relies on for data integrity. Likewise, the proposed system assumes the RTP label for the same reason.

The network emulator is non-deterministic. It applies the same statistical traffic shaping properties to each packet stream; however, specific segments of audio may experience varying conditions from trial to trial. For this reason, we average the MOS from multiple audio files to get a composite score for each traffic condition. It is not uncommon for an individual trials to show degradation even

when the general trend suggestions improvement because silent and active regions are not affected identically.

6.4.1 Control

Under ideal network conditions, the two operational modes should perform the same. This first experiment verifies the testing environment is not introducing errors from conditions other than those being examined.

Table 6.4: Control Experiment

Test #	**PESQ-MOS**		**MOS-LQO**	
	UDP	*RTP*	*UDP*	*RTP*
0	4.500	4.500	4.549	4.549
1	4.500	4.500	4.549	4.549
2	4.500	4.500	4.549	4.549
3	4.500	4.500	4.549	4.549
4	4.500	4.500	4.549	4.549
5	4.500	4.500	4.549	4.549
Average	4.500	4.500	4.549	4.549

Table 6.4 shows the two protocols behave identically with no emulated traffic conditions active. Both operational modes achieve top scores confirming that outside factors are not influencing the results. Recall that the maximum MOS-LQO score is 4.549 despite the traditional subjective scale extending up to 5.000.

6.4.2 Dropped Packets

Sometimes packets get lost enroute to their destination. The network emulator mimics this behavior by intentionally discarding a certain percentage of the packets it forwards. Table 6.5 shows scores corresponding to a range of packet loss rates from 1% to 10%.

Table 6.5: Emulated Packet Loss

Test #	**1%**		**2%**		**3%**		**5%**		**10%**	
	UDP	*RTP*	*UDP*	*RTP*	*UDP*	*RTP*	*UDP*	*RTP*	*UDP*	*RTP*
0	4.261	4.520	3.439	3.990	2.888	3.561	3.851	2.544	1.984	2.789
1	3.262	3.899	4.211	4.334	3.157	3.954	2.056	2.724	2.096	2.779
2	3.170	4.096	3.490	3.561	3.895	3.779	2.937	2.981	2.390	2.576
3	4.272	4.392	4.365	3.835	3.393	3.068	3.056	3.013	3.118	2.772
4	2.855	4.165	2.547	3.762	2.125	3.985	3.271	3.012	2.362	2.578
5	3.866	4.515	4.414	4.230	3.504	3.914	2.020	3.640	2.110	2.904
Average	3.614	4.265	3.344	3.952	3.160	3.710	2.865	2.986	2.343	2.939

These results indicate that RTP typically provides a 17% to 25% increase in MOS over UDP. When RTP detects a lost packet, it inserts a period of silence in place of that information. Therefore, the overall output from RTP is better time-aligned when considering subsections of the audio signal. The PESQ algorithm rates signals with highly correlated utterances favorably so this observation is consistent with the metric's design.

RTP does a reasonable job maintaining scores near toll quality for minimal amounts of packet loss. However, at loss rates above a few percent both protocols experience a significant decrease in listening quality. This might suggest Xinu RTP

could further benefit from a packet loss concealment technique using a comfort noise or repeating the last packet in place of inserting silence [51]. The results must be interpreted with caution as PESQ is extra sensitive to clipping effects due to packet loss. These scores tend to be lower than human subjects might rank. Still, RTP shows noticeable improvement relative to UDP.

6.4.3 Tuning the RTP Device

Configuring Xinu RTP for a particular application is a balancing act between competing parameters. We have already discussed the trade-off between the ability to handle delay variation and exceeding a fixed delay budget. A related topic is how long to wait for a particular packet before giving up and considering it lost. This timeout depends on the packet queue size; if the timeout is too large the buffer may overflow. Alternatively packets are more likely to be considered late if the timeout is too small. This experiment considers various timeout settings for a packet queue size 10 and a 2% packet loss rate (Table 6.6).

Table 6.6: RTP Packet Timeout Adjustment

Test #	**40 ms**	**60 ms**	**80 ms**	**100ms**	**120 ms**
0	2.948	3.684	3.990	4.041	3.917
1	3.954	4.365	4.334	4.005	3.753
2	3.693	4.184	3.561	3.809	4.080
3	3.669	3.816	3.835	3.617	3.383
4	3.779	4.458	3.762	3.529	3.905
5	4.035	3.814	4.230	3.572	3.357
Average	3.680	4.054	3.952	3.762	3.683

According to the results, a 60 ms timeout attains the highest MOS. However, an 80 ms timeout is just slightly lower yet allows for additional delay variation. We can also consider this parameter in terms of packets queued for a particular RTP profile. Since each packet contains 20 ms of audio, the listed timeouts correspond to allowing 2 to 6 packets accumulate in the buffer. An 80 ms timeout is a good fit for this application because it allows 4 additional packets beyond the 5 originally queued during call initiation. This still leaves one slot of headroom before filling the buffer completely. We use an 80 ms timeout in these experiments unless otherwise noted.

6.4.4 Packet Delay Variation

Packets traversing networks from source to destination may take different paths. Also traffic saturation can vary greatly throughout the duration of a phone call. Consequently, the total time it takes for individual packets to arrive at its destination deviates to a significant degree. The network emulator can imitate this phenomenon by intentionally increasing the transit time for a certain percentage of packets. In this experiment, we observe the effect of packet delay variation on VoIP traffic. The parameters in Table 6.7 are representative of an actual network carrying VoIP calls amidst other data. The field study responsible for these figures comes from Psytechnics, creators of the technology behind the PESQ algorithm [60].

Table 6.7: Emulated Delay Variability

Test #	**25 ms 20%**		**50 ms 5%**		**75 ms 15%**		**Overall**	
	UDP	*RTP*	*UDP*	*RTP*	*UDP*	*RTP*	*UDP*	*RTP*
0	1.801	4.549	2.612	4.549	1.545	2.964	1.986	4.020
1	1.891	4.549	2.267	4.549	1.636	2.560	1.931	3.886
2	1.940	4.452	3.566	4.549	1.737	2.380	2.414	3.794
3	2.205	4.549	2.145	4.549	2.124	1.985	2.158	3.694
4	2.209	4.549	2.921	4.549	1.714	2.600	2.281	3.899
5	2.156	4.549	2.590	4.549	2.130	2.551	2.292	3.883
Average	2.034	4.533	2.684	4.549	1.814	2.507	2.177	3.836

RTP is especially good at correcting for packet delay variation within its calibrated window. As Table 6.7 shows, RTP completely reassembled all but one of the delayed packet streams for up to 50 ms delays. On the other hand, UDP suffered significant quality loss for all trials. As the delay approaches the RTP timeout setting, the device is less effective at reconstructing the original stream, but still shows improvement over UDP. RTP is significantly better at handling delayed packets compared to dropped packets because it can re-align the audio stream with the original data, not just a placeholder.

In terms of the toll quality reference target, RTP maintains increased scores coming up just shy of a 4.0 MOS overall. UDP achieves scores implying calls would require considerable effort to interpret and may not be understandable at all. We could further improve the performance of RTP by increasing the timeout and the packet queue size; however, these changes are limited by the budget set by the maximum delay a listener cannot distinguish.

6.5 Summary of Findings

Independent of any application, Xinu RTP correctly handles situations where packets arrive out of order or do not arrive on schedule. Our test suite demonstrates successful operation in both regular and more obscure scenarios. Furthermore, Xinu RTP provides measurable improvement in voice communication quality as compared to the legacy UDP-based system. RTP adds some robustness against dropped packets, but is much more effective at reordering streams subject to packet delay variation. For best performance, RTP must consider its timeout setting and packet queue size in terms of the fixed delay acceptable to human perception.

CHAPTER 7

Conclusion

This thesis describes hardware and software enhancements to an embedded VoIP platform focused on real-time operation. The remaining sections conclude this work with a summary of the main contributions and suggestions for future expansion.

7.1 Contributions

The work this thesis describes spans three main categories: dependencies, implementations, and original designs. Dependent works, while intertwined throughout the XinuPhone, are contributions from third parties for whom we wish to attribute proper credit (also referenced throughout this document). These components primarily exist 'as-is' with little adaption from their original form. For example, the Embedded Xinu operating system running on the router platform falls into this category. Specifically, the device API and Lund's VoIP application code are major components leveraged in this work. Likewise, the dsPIC firmware makes use of Microchip's macros and platform support libraries as building blocks for the XinuPhone application, such as the ITU G.711 CODEC.

Next, we consider original implementations based on the high-level designs of others. Contributions in this category leverage concepts from existing solutions, but with non-trivial adaptation to suit this application. The hardware designs for the filters, amplifiers, and serial transceiver are good examples of this kind of contribution. The original concepts are well-defined in textbooks and application notes; nonetheless, the application specific implementation requires component material selection, appropriate nominal values, and adjustments for single-rail power supply. The core of the RTP device is also largely an implementation of the requirements described in the RFC. The required fields and packet structure directly follow the specification; however, the implementation details in Embedded Xinu, including a number of simplifications, remain unique.

The last category encompasses contributions unique to this thesis. Although inspired by ideas from various sources, these segments of work represent novel contributions to the research community. For instance, the lightweight design and implementations of software constructs that allow RTP to run efficiently on resource-constrained hardware. Namely, the packet queuing mechanism and thread model for asynchronous processing are not presented in other literature. Similarly, the firmware for the XinuPhone's audio processor is entirely homegrown. In summary, this work combines many existing contributions with novel advancements to deliver the proposed real-time telephony system.

7.2 Future Work

The XinuPhone looks to be a promising pedagogical tool for hands-on systems design experience. Our Internet telephony platform combines software architecture, signal processing, computer networking, and embedded systems curriculum into an application relevant to students. An empirical evaluation of the XinuPhone's success in the classroom is a logical next step. Ideally, the use of this platform should demonstrate a significant improvement in student interest, a better understanding of hardware/software co-design, and an increased awareness of challenges arising from resource limitations. Evaluating educational curriculum can be especially challenging because a baseline or control group may be difficult to establish. Nonetheless, we believe a learning outcome analysis would be a worthwhile endeavour.

In terms of technical research, an open and flexible platform reveals opportunities for advancement in the Internet telephony field. One of the obvious drawbacks to the current system is the static configuration parameters that must be selected at compile time with a specific application in mind. While this design is ideal for systems with limited resources, embedded devices are rapidly seeing increases in speed, memory, and energy efficiency that all enable more possibilities for dynamic designs. Others have already proposed means for obtaining feedback about the performance of the transmission channel [61], [62]. Perhaps an adaptive

system could apply this knowledge about the channel characteristics to make more informed decisions about how many packets to queue and how long to wait for late arrivals. Dynamic adaptation would remove the application programmer's need to customize the RTP device, therefore, eliminating the need to compile custom kernels for different applications.

At present, this platform still lacks an elegant mechanism to associate with peer devices and initiate phone calls. The user must configure the address and port information for the channel manually rather than through a directory scheme. Although SIP and H.232 provide means for call signaling, full implementations of these protocols are likely overkill for an experimental lab environment. Instead, we envision a lightweight discovery protocol that would allow the XinuPhone to rendez-vous with other systems on the same network. This extension would better fit in with Embedded Xinu's paradigm and make for an excellent networking lab assignment.

7.3 Summary

Voice over Internet Protocol networks are here to stay. Over the next several years we will continue to see Internet telephony replace legacy systems to leverage many advantages of converged data/voice networks. Despite the field being relatively mature, the lack of research and standardization concerning protocol behaviors

leaves unanswered questions about how to best implement these systems. This thesis proposes an experimental telephony platform for exploring the design space left open by these questions.

The XinuPhone's new hardware design is more robust, scalable, and logistically easier to implement. These features make the platform ideal for university research and classroom environments alike. Software improvements show demonstrable performance increase over the previous generation when the transport network causes packet loss and delay variation. Moreover, we can provide these enhancements for a reasonable monetary cost and with minimal computing overhead.

BIBLIOGRAPHY

[1] K. Persohn and D. Brylow, "Interactive Real-Time Embedded Systems Education Infused with Applied Internet Telephony," in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, July 2011, pp. 199 – 204.

[2] D. Minoli and E. Minoli, *Delivering Voice over IP Networks*, 2nd ed. Wiley Publishing, 2002.

[3] B. Douskalis, *Putting VoIP to Work*. Prentice Hall, 2002.

[4] L. Madson, J. V. Meggelen, and R. Bryant, *Asterisk: The Definitive Guide*, 3rd ed. O'Reilly Media, 2011.

[5] U. S. Congress, "Wireless Communications and Public Safety Act of 1999," October 1999.

[6] P. D. van der Puije, *Telecommunication Circuit Design*. John Wiley and Sons, 2002.

[7] B. Wildrow and S. D. Stearns, *Adaptive Signal Processing*. Prentice Hall, 1985.

[8] S. Haykin, *Adaptive Filter Theory*, 4th ed. Prentice Hall, 2002.

[9] T. Noergaard, *Embedded Systems Architecture*. Newnes, 2005.

[10] J. Catsoulis, *Designing Embedded Hardware*, 2nd ed. O'Reilly Media, 2005.

[11] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts*, 8th ed. John Wiley & Sons, 2009.

[12] D. E. Comer, *Operating System Design: The XINU Approach*. Prentice Hall, 1984.

[13] Microchip, "dsPIC DSC DSP Library," Microchip Technology Inc., Datasheet DSO1033B-28, 2006.

[14] J. H. McClellan, R. W. Schafer, and M. A. Yoder, *DSP First: A Multimedia Approach*. Prentice Hall, 1988.

[15] A. Ambardar, *Digital Signal Processing: A Modern Introduction*. Thomson Learning, 2007.

[16] "Pulse Code Modulation (PCM) of Voice Frequencies," International Telecommunications Union, Recommendation G.711, November 1988.

[17] "Adaptive Differential Pulse Code Modulation (ADPCM)," International Telecommunications Union, Recommendation G.726, December 1990.

[18] J.-M. Valin, "The Speex Codec Manual," Valin/Xiph.org Foundation, Tech. Rep. 1.2 Beta 2, May 2007.

[19] R. Schreier and G. C. Temes, *Understanding Delta-Sigma Data Converters.* IEEE Press, 2005.

[20] D. E. Comer, *Computer Networks and Internets*, 5th ed. Prentice Hall, 2008.

[21] M. A. Dye, *Network Fundamentals, CCNA Exploration Companion Guide*, 2nd ed. Cisco Press, 2008.

[22] B. A. Forouzan, *Data Communications and Networking*, 3rd ed. McGraw-Hill, 2003.

[23] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*, 4th ed. Morgan Kaufmann, 2007.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," Internet Engineering Task Force, RFC 3550, July 2003.

[25] ——, "RTP Profile for Audio and Video Conferences with Minimal Control," The Internet Society, RFC 3551, July 2003.

[26] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," Internet Engineering Task Force, RFC 3261, June 2002.

[27] "Packet-based multimedia communications systems," International Telecommunications Union, Recommendation H.232, December 2009.

[28] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," The Internet Society, RFC 4566, July 2006.

[29] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)," The Internet Society, RFC 3711, March 2004.

[30] P. Zimmermann, A. Johnston, and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP," Internet Engineering Task Force, RFC 6189, April 2011.

[31] J. Ganssle, "Embedded Y2K," *Embedded Systems Programming*, vol. 3, no. 17, pp. 97 – 99, 1999.

[32] B. Benson, A. Arfaee, C. Kim, R. Kastner, and R. K. Gupta, "Integrating embedded computing systems into high school and early undergraduate education," *Education, IEEE Transactions on*, vol. PP, no. 99, pp. 1–6, September 2010.

[33] P. Koopman, H. Choset, R. Gandhi, B. Krogh, D. Marculescu, P. Narasimhan, J. M. Paul, R. Rajkumar, D. Siewiorek, A. Smailagic, P. Steenkiste, D. E. Thomas, and C. Wang, "Undergraduate embedded system education at

Carnegie Mellon," *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 3, pp. 500–528, August 2005.

[34] D. Brylow and B. Ramamurthy, "Nexos: A next generation embedded systems laboratory," *SIGBED Review*, vol. 6, no. 1, January 2009.

[35] R. Zhang, J. Liu, Y. Gao, and J. Qiao, "A realized embedded streaming media system," in *Wireless, Mobile and Multimedia Networks, 2006 IET International Conference on*, November 2006, pp. 1–4.

[36] G. Cuijuan, M. Changyun, W. Zhigang, and L. Jie, "Design and implementation of simplified mgcp stack based on dsp," in *Future Networks, 2010. ICFN '10. Second International Conference on*, January 2010, pp. 381 –384.

[37] "Arduino," http://www.aruino.cc, 2011.

[38] Marquette University Systems Lab, "Embedded Xinu Wiki," http://xinu.mscs.mu.edu/, 2011.

[39] D. Rowe, "Rowetel: A phone call should be a human right, not a privlege," http://www.rowetel.com/, 2011.

[40] Z. D. Lund, "A VoIP Implementation on an Embedded Platform," Master's thesis, Marquette University, 2010.

[41] S. Morlat, "oRTP, a Real-time Transport Protocol (RTP,RFC3550) library," http://www.linphone.org/eng/documentation/dev/ortp.html, 2012.

[42] LIVE555 Networks, Inc., "LIVE555 Streaming Media," http://www.live555.com/liveMedia/, 2012.

[43] GNU Telephony, "GNU ccRTP," http://www.gnu.org/software/ccrtp/, 2006.

[44] Free Software Foundation, Inc., "Various licenses and comments about them," http://www.gnu.org/licenses/license-list.html, 2012.

[45] Microchip, "dsPIC33FJ64GP802: High-Performance, 16-bit Digitial Signal Controllers," Microchip Technology Inc., Datasheet DS70292F, 2011.

[46] National, "Low Power Quad Operational Amplifiers," National Semiconductor Corporation, Datasheet DS009299, 2004.

[47] D. Self, *Small Signal Audio Design.* Focal Press, 2010.

[48] R. Schaumann and M. E. V. Valkenburg, *Design of Analog Filters.* Oxford University Press, 2001.

[49] B. Lathi, *Linear Systems and Signals*, 2nd ed. Oxford University Press, 2005.

[50] H. Schulzrinne, "Some Frequently Asked Questions about RTP," http://www.cs.columbia.edu/~hgs/rtp/, January 2008.

[51] T. Braun, M. Diaz, J. E. Gabeiras, and T. Staub, *End-to-End Quality of Service Over Heterogeneous Networks.* Springer-Verlag, 2008.

[52] G. J. Myers, *The Art of Software Testing.* John Wiley & Sons, 1979.

[53] “Methods for subjective determination of transmission quality,” International Telecommunications Union, Recommendation P.800, August 1996.

[54] “Mean Opinion Score (MOS) terminology,” International Telecommunications Union, Recommendation P.800.1, July 2006.

[55] “Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs,” International Telecommunications Union, Recommendation P.862, February 2001.

[56] “Mapping function for transforming P.862 raw result scores to MOS-LQO,” International Telecommunications Union, Recommendation P.862.1, November 2003.

[57] “The E-model: a computational model for use in transmission planning,” International Telecommunications Union, Recommendation G.107, December 2011.

[58] “Network model for evaluating multimedia tranmission performance over Internet Protocol,” International Telecommunications Union, Recommendation G.1050, March 2011.

[59] D. Brylow and K. Thurow, “Hands-on networking labs with embedded routers,” in *SIGCSE 2011: Proceedings of the 42nd SIGCSE technical symposium on Computer science education*, 2011.

[60] A. Rix, “End-to-end speech quality assessment of networks using PESQ (P.862),” Psytechnics Limited, Workshop ITU-T SG12, October 2011.

[61] T. Friedman, R. Canceres, and A. Clark, “RTP Control Protocol Extended Reports (RTCP XR),” The Internet Society, RFC 3611, November 2003.

[62] L. Carvalho, E. Mota, R. Aguiar, A. F. Lima, J. N. de Souza, and A. Barreto, “An E-model implementation for speech quality evaluation in VoIP systems,” in *10th IEEE Symposium on Computers and Communication (ISCC)*, June 2005, pp. 933 – 938.

[63] P. Single, “Optimum hybrid design,” National Semiconductor, Application Note 397, July 1985.

[64] A. R. Hambley, *Electronics*, 2nd ed. Prentice Hall, 2000.

[65] R. C. Hsu and W.-C. Liu, “Project based learning as a pedagogical tool for embedded system education,” in *Information Technology: Research and Education, 2005. ITRE 2005. 3rd International Conference on*, June 2005, pp. 362–366.

[66] W. Stapleton, “Microcomputer fundamentals for embedded systems education,” in *Frontiers in Education Conference, 36th Annual*, October 2006, pp. 6–10.

[67] M. Mitchell, "Using PWM Timer_B as a DAC," Texas Instruments, Application Report SLAA116, December 2000.

[68] *MCF532x/7x Embedded VoIP Solution*, Freescale Semiconductor, 2007.

[69] *SPA3102 Phone Adapter with Router*, Cisco Systems, Inc., 2008.

[70] B. Carter, "A Single-Supply Op-Amp Circuit Collection," Texas Instruments, Application Report SLOA058, November 2000.

APPENDIX A

Audio File Transport Test Application

```
1  /**
2   * @file      xsh_test.c
3   * @provides xsh_test.
4   *
5   * $Id: xsh_test.c 2610 2012-03-24 21:04:35Z kpersohn $
6   */
7  /* Embedded Xinu, Copyright (C) 2009.  All rights reserved. */
8
9  #include <stddef.h>
10 #include <stdlib.h>
11 #include <stdio.h>
12 #include <device.h>
13 #include <network.h>
14 #include <ether.h>
15 #include <rtp.h>
16 #include <tftp.h>
17 #include <shell.h>
18
19 extern int _binary_data_u_af1s01_wav_start;
20 extern int _binary_data_u_af1s02_wav_start;
21 extern int _binary_data_u_af1s03_wav_start;
22 extern int _binary_data_u_am1s01_wav_start;
23 extern int _binary_data_u_am1s02_wav_start;
24 extern int _binary_data_u_am1s03_wav_start;
25
26 struct rtpTestFile rtpTestTab[] =
27 {
28     {0, "u_af1s01.wav", 128044, &_binary_data_u_af1s01_wav_start},
29     {1, "u_af1s02.wav", 128044, &_binary_data_u_af1s02_wav_start},
30     {2, "u_af1s03.wav", 128044, &_binary_data_u_af1s03_wav_start},
31     {3, "u_am1s01.wav", 128044, &_binary_data_u_am1s01_wav_start},
32     {4, "u_am1s02.wav", 128044, &_binary_data_u_am1s02_wav_start},
33     {5, "u_am1s03.wav", 128044, &_binary_data_u_am1s03_wav_start}
34 };
35
36 static void listRtpTests(void);
37 static thread testrecv(ushort, void *, uint, tid_typ);
38
39 /**
40  * Shell command (test) provides a mechanism for testing Xinu features.
   The
41  * action and output varies depending on the feature currently being tested.
42  * This is not meant to serve as a permanent shell command for a particular
43  * action.
44  * @param nargs number of arguments
```

```
45  * @param args   array of arguments
46  * @return non-zero value on error
47  */
48 shellcmd xsh_test(int nargs, char *args[])
49 {
50     ushort dev = 0;
51     ushort test = 0;
52     struct netaddr rhost;
53     struct netaddr *localhost = NULL;
54     struct netaddr *remotehost = NULL;
55     struct netif *interface = NULL;
56     uchar *data = NULL;
57     uchar rtpmode = 0;
58     uchar mode = 0;
59     uint rxbytes = 0;
60     int count = 0;
61     int i = 0;
62     message msg = 0;
63     tid_typ tid;
64 
65     rtpmode = RTP_MODE_BASIC;
66 
67     if (nargs < 2 || (nargs == 2 && strncmp(args[1], "--help", 6) == 0))
68     {
69         printf("\nUsage:\n");
70         printf("\t%s -l\n", args[0]);
71         printf("\t%s [--seq] -r <test>\n", args[0]);
72         printf("\t%s -s <test> <dst ip>\n", args[0]);
73         printf("Description:\n");
74         printf("\tSends and receives audio files using RTP.\n");
75         printf("Parameters:\n");
76         printf("\t-l\t\tList available test files\n");
77         printf("\t-r\t\tReceive data from any host\n");
78         printf("\t-s\t\tSend data to specified remote host\n");
79         printf("\t--seq\tOptionally re-order incoming packet sequence\n");
80         printf("\t<dst ip>\tIP address of remote host\n");
81         printf("\t<test>\t\tIndex of test file (see -l)\n");
82         return SHELL_OK;
83     }
84     else
85     {
86         for (i = 1; i < nargs; i++)
87         {
88             if (strncmp(args[i], "-l", 2) == 0)
89             {
90                 listRtpTests();
91                 return SHELL_OK;
92             }
93             else if (strncmp(args[i], "-r", 2) == 0)
94             {
95                 if (i + 1 < nargs)
96                 {
97                     mode = RTP_TEST_MODE_RECV;
98                     test = atoi(args[i + 1]);
```

```
 99                     i += 1;
100                 }
101                 else
102                 {
103                     fprintf(stderr, "Missing parameter.\n");
104                     return SHELL_ERROR;
105                 }
106             }
107             else if (strncmp(args[i], "-s", 2) == 0)
108             {
109                 if (i + 1 < nargs)
110                 {
111                     mode = RTP_TEST_MODE_SEND;
112                     test = atoi(args[i + 1]);
113                     // FIXME: Assuming 192.168.6.123 for thesis testing
114                     //remotehost = &rhost;
115                     //dot2ipv4(args[i + 2], remotehost);
116                     dot2ipv4(RTP_TEST_IP_ADDR, &rhost);
117                     //i += 2;
118                     i += 1;
119                 }
120                 else
121                 {
122                     fprintf(stderr, "Missing parameter.\n");
123                     return SHELL_ERROR;
124                 }
125             }
126             else if (strncmp(args[i], "--seq", 5) == 0)
127             {
128                 rtpmode = RTP_MODE_SEQ;
129             }
130             else
131             {
132                 fprintf(stderr, "Unrecognized parameter. ");
133                 fprintf(stderr, "Try %s --help\n", args[0]);
134                 return SHELL_ERROR;
135             }
136         }
137     }
138
139     /* Validate test case */
140     if (test >= RTP_TEST_TAB_SIZE)
141     {
142         fprintf(stderr, "Invalid test case.\n");
143         return SHELL_ERROR;
144     }
145
146     /* Pick the first active local interface */
147     i = 0;
148     while ((i < NETHER) &&
149            (NULL == (interface = netLookup((ethertab[i].dev)->num))))
150         i++; /* Keep looking */
151
152     /* Lookup IP address of local interface */
```

```
153     if (NULL == interface)
154     {
155         fprintf(stderr, "No network interface found.\n");
156         return SYSERR;
157     }
158     localhost = &(interface->ip);
159 
160     /* Allocate RTP device */
161     if ((ushort)SYSERR == (dev = rtpAlloc()))
162     {
163         fprintf(stderr, "Failed to allocate RTP device.\n");
164         return SHELL_ERROR;
165     }
166 
167     /* Open RTP device */
168     if (SYSERR == open(dev, localhost, &rhost, RTP_TEST_PORT,
169     //if (SYSERR == open(dev, localhost, remotehost, RTP_TEST_PORT,
170                        RTP_TEST_PORT, RTP_PT_L16LE_8K, rtpmode))
171     {
172         fprintf(stderr, "Failed to open RTP device.\n");
173         return SHELL_ERROR;
174     }
175 
176     switch (mode)
177     {
178         case RTP_TEST_MODE_RECV:
179 
180             /* Allocate space for incoming file */
181             if (NULL == (data = (uchar *)malloc(rtpTestTab[test].size)))
182             {
183                 fprintf(stderr, "Failed to allocate receive buffer.\n");
184                 close(dev);
185                 return SHELL_ERROR;
186             }
187 
188             /* Spawn helper thread to receive data */
189             tid = create(testrecv, SHELL_CMDSTK, SHELL_CMDPRIO, "testrecv",
190                          4, dev, data, rtpTestTab[test].size, gettid());
191             if (SYSERR == tid)
192             {
193                 fprintf(stderr, "Failed to create receiver thread.\n");
194                 free(data);
195                 close(dev);
196                 return SHELL_ERROR;
197             }
198 
199             /* Redirect stdin, stdout, stderr to parent (this) thread */
200             thrtab[tid].fdesc[0] = thrtab[gettid()].fdesc[0];
201             thrtab[tid].fdesc[1] = thrtab[gettid()].fdesc[1];
202             thrtab[tid].fdesc[2] = thrtab[gettid()].fdesc[2];
203 
204             /* Launch thread */
205             recvclr();
206             ready(tid, RESCHED_NO);
```

```
207
208                 /* Poll helper thread for progress updates */
209                 while ((msg != TIMEOUT) && (msg != tid))
210                 {
211                     rxbytes = msg;
212                     printf("RX Bytes: %d\r", rxbytes);
213                     msg = recvtime(RTP_TEST_TIMEOUT);
214                 }
215
216                 /* Was the helper thread stood up? */
217                 if (TIMEOUT == msg)
218                     kill(tid);
219
220                 /* TFTP received file to default server */
221                 tftpPut(data, rxbytes, rtpTestTab[test].file, TFTP_SERVER);
222                 printf("Saved %d bytes to TFTP server.\n", rxbytes);
223
224                 free(data);
225                 break;
226             case RTP_TEST_MODE_SEND:
227
228                 /* Enable flow control */
229                 control(dev, RTP_CTRL_FLOW_CTRL, TRUE, NULL);
230
231                 /* Send */
232                 count = write(dev, rtpTestTab[test].addr, rtpTestTab[test].size);
233
234                 if (SYSERR == count)
235                 {
236                     fprintf(stderr, "RTP write failed.\n");
237                     close(dev);
238                     return SHELL_ERROR;
239                 }
240
241                 printf("Sent test file %s successfully.\n",
242                        rtpTestTab[test].file);
243
244                 break;
245             default:
246                 fprintf(stderr, "Unknown mode.\n");
247                 close(dev);
248                 return SHELL_ERROR;
249     }
250
251     close(dev);
252
253     return SHELL_OK;
254 }
255
256 static void listRtpTests(void)
257 {
258     int i = 0;
259
260     printf("\n");
```

```
261     printf("\t+===+======================+============+==============+\n");
262     printf("\t| I |       Filename       |    Size    |   Address    |\n");
263     printf("\t+===+======================+============+==============+\n");
264     for (i = 0; i < RTP_TEST_TAB_SIZE; i++)
265     {
266         printf("\t| %d ", rtpTestTab[i].index);
267         printf("| %16s ", rtpTestTab[i].file);
268         printf("| %8d ", rtpTestTab[i].size);
269         printf("| 0x%08x ", rtpTestTab[i].addr);
270         printf("|\n");
271     }
272     printf("\t+===+======================+============+==============+\n");
273     printf("\n");
274 }
275
276 static thread testrecv(ushort dev, void *buf, uint len, tid_typ parent) {
277     int count = 0;
278     uint rxbytes = 0;
279
280     while (rxbytes < len)
281     {
282         /* Read from RTP device */
283         count = read(dev, buf, len - rxbytes);
284
285         if (SYSERR == count)
286         {
287             fprintf(stderr, "RTP read failed.\n");
288             free(buf);
289             close(dev);
290             return SYSERR;
291         }
292
293         /* Advance pointer */
294         buf += count;
295
296         /* Keep track of bytes read */
297         rxbytes += count;
298
299         /* Report progress back to parent */
300         //kprintf("Sending %d bytes to tid %d.\r\n", rxbytes, parent);
301         send(parent, rxbytes);
302     }
303
304     return OK;
305 }
```