View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by epublications@Marquette

Marquette University
e-Publications@Marquette

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

A VoIP Implementation on an Embedded Platform

Zachary D. Lund
Marquette University

Recommended Citation
Lund, Zachary D., "A VoIP Implementation on an Embedded Platform" (2010). *Master's Theses (2009 -)*. Paper 35.
http://epublications.marquette.edu/theses_open/35

A VOIP IMPLEMENTATION ON AN EMBEDDED PLATFORM

by

Zachary D. Lund

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

May 2010

ABSTRACT
A VOIP IMPLEMENTATION ON AN EMBEDDED PLATFORM

Zachary D. Lund

Marquette University, 2010

This thesis presents a method of building an open VoIP telephone using off-the-shelf hardware. VoIP has become an important area of study in computer science and engineering, but many of the pieces are expensive and proprietary. We discuss the process of building an open IP telephone, the design decisions and difficulties, a performance evaluation of the different pieces of the system, and methods and suggestions for improving the overall system.

The IP telephone is built on the Freescale 68HC12 microcontroller because of its analog and digital capabilities and the Linksys WRT54GL router for its embedded simplicity and network capabilities. The lightweight Embedded Xinu operating system was selected to build up the VoIP capabilities on top of a functional network stack.

Our results show that building this VoIP device is viable. The completed IP telephone establishes a call between two routers using UDP network transport and μ-law companding. The roundtrip latency is less than two milliseconds, and the phone has a PESQ MOS (voice quality) of 3.13.

ACKNOWLEDGEMENTS

Zachary D. Lund

Many thanks to those who have helped me along the way to a completed thesis:

- Dennis Brylow, my thesis advisor and good friend, for the research opportunity and assistance.
- George Corliss, my thesis director, for the meticulous proofreading and continuous pushing to finish.
- Doug Harris and Mike Johnson, my committee members, for providing useful feedback about networks and signal processing.
- Aaron Gember, Adam Koehler, Adam Mallen, Michael Schultz, and the entire Systems Laboratory for providing a supportive and entertaining lab environment.
- Donna Lund, my mother, for never allowing me to think that schooling was optional.
- Clarissa Kampa, my fiancée, for always supporting me, even when it meant one more semester in different states.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . **i**

TABLE OF CONTENTS . . . . **ii**

LIST OF TABLES . . . . **v**

LIST OF FIGURES . . . . **vi**

CHAPTER 1 Introduction **1**

1.1 Thesis Statement 1

1.2 Overview 1

1.3 Contributions 3

CHAPTER 2 Background **4**

2.1 Embedded Systems 4

2.2 Voice Over IP Telephony 5

2.3 Digital Signal Processing 6

2.3.1 Pulse-code Modulation 6

2.3.2 Audio Filters and Effects 8

2.4 Tools 8

2.5 Summary of Background 9

CHAPTER 3 Related Work **10**

3.1 Voice Data Compression 10

3.2 VoIP Over Wireless and Mobile Networks 11

3.3 Specialized VoIP Protocols 12

3.4 VoIP Security 13

3.5 Embedded VoIP 13

3.6 Summary of Related Works 14

CHAPTER 4 Embedded VoIP Framework **15**

4.1 68HC12 Microcontroller 16

4.1.1 Compilation 17

4.1.2 Code 19

4.1.3 Analog and Digital Conversion 24

4.1.4 Power Cycling 26

4.2 Linksys WRT54GL Router 27

4.2.1 Xinu Operating System 27

4.2.2 Serial Communication 28

4.2.3 Network Transmission 30

4.3 Desktop Tools 30

4.4 Summary of Framework 34

CHAPTER 5 Performance Analysis **36**

5.1 Embedded VoIP Latency 36

5.2 Desktop VoIP Latency 43

5.3 Embedded VoIP Voice Quality 46

5.4 Sources of Error 48

5.4.1 Sampling and Digitization 48

5.4.2 Clock Synchronization 49

5.4.3 Compression 51

5.4.4 Serial 51

5.4.5 Network 52

5.5 Network Robustness 52

5.6 Embedded IP Telephone Cost 55

5.7 Summary of Performance Analysis 56

CHAPTER 6 Discussion and Summary **57**

6.1 Discussion 57

6.1.1 A Previous Attempt 57

6.1.2 From Synchronous to Asynchronous 58

6.2 Future Work 62

6.3 Summary 63

REFERENCES **65**

APPENDIX A 68HC12 Makefile **69**

APPENDIX B Serial Receive Rate **71**

LIST OF TABLES

4.1 68HC12 Timer Interrupt Tick Resolution 22

4.2 Effect of μ-Law Compression on a Sine Wave 33

5.1 Summary of latency measurements from shortest data path to longest data path 44

5.2 PESQ MOS test results summary with sample size of 23 47

5.3 Costs of components to build experimental prototype system 55

5.4 Costs to build a basic embedded VoIP device 56

LIST OF FIGURES

4.1 Embedded VoIP device connection overview 15

4.2 68HC12 hardware overview 17

4.3 68HC12 timer interrupt code flow 20

4.4 ADC microphone circuit 24

4.5 DAC speaker circuit 25

4.6 Microcontroller rebooter circuit 27

4.7 1 kHz Sine Wave Sampled at 8 kHz 33

4.8 1 kHz Sine Wave Sampled at 8 kHz After μ-Law Compression 34

5.1 Embedded VoIP device connection and data path flow overview 37

5.2 Data path flow for ADC and DAC latency test case 38

5.3 Original 1 kHz sine wave (top) and sine wave delayed 92 μs by ADC and DAC (bottom) 38

5.4 Original 1 kHz sine wave (top) and sine wave delayed 131 μs by ADC, DAC, and μ-law companding (bottom) 38

5.5 Original 1 kHz sine wave (top) and sine wave delayed 142 μs by ADC, DAC, μ-law companding, and 8 kHz sampling interrupts (bottom) 38

5.6 Data path flow for latency measurement through the serial loopback device on the microcontroller 39

5.7 Original 1 kHz sine wave (top) and sine wave delayed 389 μs by the serial loopback on the microcontroller (bottom) 39

5.8 Data path flow for latency measurement through the serial ports with loopback on the router 39

5.9 Original 1 kHz sine wave (top) and sine wave delayed 574 μs by the serial ports and router loopback (bottom) 39

5.10 Data path flow for latency measurement through the serial ports with loopback on the router (internal measurement using returned data from the serial port) 41

5.11 Square wave representation of the 563 μs delay due to the serial ports and the router loopback 41

5.12 Data path flow for latency measurement through the Ethernet loopback device 42

5.13 Square wave representation of the 891 μs delay due to the addition of the Ethernet loopback device 42

5.14 Data flow for roundtrip latency measurement using two microcontrollers, two routers, and a network 43

5.15 Square wave representation of the 1.875 ms delay for the round trip latency test case 43

5.16 Desktop VoIP connection and data path flow overview 44

5.17 Data flow for desktop computer ADC and DAC latency test case 45

5.18 Distorted square wave representation of the 420 μs delay for ADC and DAC latency test case 45

5.19 Data flow for roundtrip latency test case using two desktop computers and a network 46

5.20 Distorted square wave representation of the 892 μs delay for the roundtrip latency test case using two desktop computers and a network 46

5.21 Data flow for embedded speech quality testing 47

5.22 Original sine wave (top) and sampled sine wave with no correction (bottom) 49

5.23 Original sine wave (top) and sampled sine wave after μ-law companding and 8 kHz sampling interrupts with no correction (bottom) 49

5.24 Original sine wave (top) and sampled sine wave after μ-law companding and 8 kHz sampling interrupts with smoothing capacitor correction (bottom) 49

CHAPTER 1

Introduction

1.1 Thesis Statement

Inexpensive off-the-shelf components and a lightweight embedded operating system can be used to build an open IP telephone.

1.2 Overview

Voice Over IP (VoIP) is the transfer of voice data over a network or the Internet. Most VoIP solutions require either the use of proprietary software on a high powered desktop computer or the purchase of a special proprietary adapter or handset. None of the technologies required for VoIP communication are particularly specialized or proprietary. Therefore, instead of using an expensive desktop computer or specialized hardware, we will use an inexpensive off-the-shelf Linksys router, and instead of a complex, heavyweight operating system, we will use the lightweight embedded Xinu operating system. First, a microphone records the audio. Then the microcontroller samples the audio so that a computer can manipulate it. Once the computer has the digitalized signal, the computer sends the

signal across a network or the Internet to another computer. The receiving computer converts it back to an analog signal and plays it through a speaker.

This work focuses on working with the lower bound for system requirements. Even on the embedded devices market, many of the devices have hardware that is complete overkill for building an IP telephone. Therefore, we have built our system around inexpensive devices that can handle IP telephony but do not leave overhead for a verbose or sloppy implementation. The implementation also uses as little as possible beyond the hardware available on the Linksys router and the 68HC12 microcontroller.

These design decisions have kept the overall prototype system simple, inexpensive, and easy to replicate. In addition, it has been neatly integrated into the existing Marquette University Systems Lab infrastructure. This includes programmable power cycling and remote console connections, which make it ideal for use in education. Since one of the other principles of this work was openness, all embedded VoIP code is part of the Embedded Xinu repository and has been released as part of Embedded Xinu.

The structure of this thesis is described in the following outline.

- Chapter 1 introduces this thesis and why the work is important and outlines the contributions.
- Chapter 2 describes background material related to embedded systems, digital signal processing, and voice over IP telephony.

- Chapter 3 describes related work in the area of VoIP and its relation to and dissimilarities from this research.
- Chapter 4 describes the embedded VoIP implementation including hardware modifications and the code additions.
- Chapter 5 presents results, discusses the performance of the system, and analyzes sources of error and ways to mitigate those errors.
- Chapter 6 discusses some of the difficulties of the research, summarizes this work, and presents directions for future research.

1.3 Contributions

This thesis delivers an implementation of voice over IP for a resource-constrained embedded system, which we refer to as "embedded VoIP." This implementation includes the hardware modifications to attach a microphone and speaker to a 68HC12 microcontroller. It also includes the code that runs on the 68HC12 microcontroller and the Linksys WRT54GL router.

The embedded VoIP performance analysis shows that the IP telephone establishes a call between two routers. An established call has a latency of less than two milliseconds, and the phone has a PESQ MOS of 3.13. We also perform an analysis of sources of error and provide a discussion of mitigation strategies for those sources of error and the different problems that occurred during the development of the system, why they happened, and how they were fixed.

CHAPTER 2

Background

Embedded voice over IP relies on three main areas including embedded systems, voice over IP telephony, and digital signal processing. Each of these areas is described below in a context relevant to the work described in the rest of this thesis.

2.1 Embedded Systems

Embedded systems typically are resource-constrained computers that provide a dedicated function through the use of specialized hardware and software [1] [2]. These embedded devices are usually part of a system that has additional hardware or that controls other components such as motors or that interacts with the environment through sensors. Examples of embedded systems include cell phones, cruise control systems in cars, and traffic lights.

Many of these embedded systems include a real-time element. Real-time systems have varying degrees of urgency. Soft real-time systems generally degrade under missed deadlines, for example, dropping frames from a video while it is playing, while hard real-time systems must adhere to strict deadlines. Medical devices such as pacemakers are examples of hard real-time systems.

2.2 Voice Over IP Telephony

Voice Over IP (VoIP) refers to a broad array of technologies and protocols that transmit human voice over Internet Protocol (IP) networks including the Internet, some of which use embedded devices [3] [4] [5]. These networks are packet switched, which means that voice data is turned into packets and transmitted and routed across a network to a destination. Prior to the beginnings of VoIP in the mid 1990s, all voice conversations took place over the Public Switched Telephone Network (PSTN). While VoIP is packet switched, the PSTN is circuit switched, which means that each phone call requires a dedicated line to connect two individuals at each end of the phone line. VoIP has the advantage of routing multiple calls over the same circuit.

On a local intranet, quality of service (QoS) can be guaranteed much like on the traditional PSTN. This means that the packets that make up a VoIP conversation can be guaranteed to be delivered because of known levels of network traffic or because of rules that prioritize real-time traffic such as VoIP traffic and de-prioritize less important network traffic. In contract to an intranet, VoIP conversations over the Internet can be nothing more than best effort, which means that VoIP packets are not guaranteed to arrive on time or at all.

There are many different implementations used for IP telephony. Two of the most common are Session Initiated Protocol (SIP) and H.323. SIP serves strictly as a signalling protocol to initiate and establish calls [6]. Since it does not define a

protocol for the actual transport of the voice data, it typically is paired with Real-time Transport Protocol (RTP) and the RTP Control Protocol (RTCP). RTP is a standard for transmitting audio and video data over the Internet. It provides sequence and time-stamp information but does not make any real-time delivery guarantees [7]. H.323 is an ITU recommendation number that specifies protocols for audio and video communication over IP networks including a signalling protocol (like SIP) and a transport protocol (like RTP) [8]. During call setup, both SIP and H.323 negotiate a codec, which handles the encoding and decoding of the voice data.

2.3 Digital Signal Processing

As some of the VoIP background suggests, quite a bit of digital signal processing is involved in voice over IP telephony. Analog voice data must be converted into a digital signal and back again. During the signal's trip across the network, it may be compressed and filtered.

2.3.1 Pulse-code Modulation

Pulse-code modulation is the process of representing a signal as a sequence of coded pulses [9]. Specifically, this allows an analog waveform to be represented as a binary sequence of data. The process of pulse-code modulation includes sampling, quantization, and encoding. Sampling and quantization acquires a discrete value of the waveform at discrete points in time. In our system, sampling and quantizing are

performed by the analog-to-digital converter.

Analog-to-digital conversion is the process of converting a continuous waveform into a discrete-time waveform. This can be accomplished by sampling the voltage level of a waveform at regular intervals in time using an external clock. According to the Nyquist sampling theorem, the sampling frequency must be twice the highest frequency in the signal [10]. Therefore, a system sampling at 8 kilohertz can capture at most a 4 kilohertz signal.

In our system, another layer of quantizing and then encoding are performed together. There are many algorithms for quantizing/encoding a signal, and they go by many names. As discussed in VoIP, the algorithm used is often called a codec (encoder/decoder), and it can also be a compander (compressor/expander). For this research, we selected the μ-law algorithm for encoding as specified in the ITU G.711 recommendation [11]. This algorithm is used in virtually all telecommunications in the United States, while A-law is used in most of Europe. Encoders can be linear, which simply divides the value by some constant, or non-linear, which performs greater compression for some values of the signal and lesser compression for other values. This means that human speech can be encoded in as little as 8 bits because the larger, higher magnitude values which occur less frequently can be more compressed than the lower magnitude values which occur more frequently.

The μ-law algorithm can be performed on an analog signal, but in this system, it is performed after the signal has been turned into discrete values. If the quantizing stage were done in an analog manner, the final step would be to encode

the value as binary pulses that can be sent over some communication medium. In our system, the value is already in a binary format, and the serial port transmits it.

2.3.2 Audio Filters and Effects

Before or after the signal has been digitized, it may be filtered to remove unwanted portions of the signal. For example, a low-pass filter passes the low frequencies and attenuates the high frequencies, while a high-pass filter does the opposite. The cutoff frequency is the frequency at which the filter begins attenuating the signal. A simple first-order low-pass filter can be realized using a resistor and a capacitor.

In addition to audio filters, which remove portions of the audio, audio effects can be used to manipulate the audio in other ways. For example an echo effect records a portion of the audio and stores it in a buffer and then adds it back to the signal at a later time producing a repeated signal called an echo.

2.4 Tools

The network and serial transmission portions of this research were performed on Linksys routers which run a modern, embedded port of the Xinu operating system [12] [13] [14]. Xinu was first developed over 25 years ago by Douglas Comer as a tool to teach operating systems. It has been ported to numerous architectures and platforms. The operating system was selected for its current use in the

Marquette University Systems Lab. It is easy to extend and already provides an IP and UDP implementation necessary for VoIP communication.

2.5 Summary of Background

The selected embedded systems provide the hardware on which the actual VoIP protocols run using sampled digital audio received by methods of digital signal processing. Everything is wrapped together using existing tools in the Marquette University Systems Laboratory.

CHAPTER 3

Related Work

Research in VoIP has been concerned primarily with bandwidth usage and compression, wireless and mobile networks, new transmission protocols, and security. Many papers have been published discussing ways to decrease bandwidth to allow more calls on a single Internet connection and decrease latency to decrease the delay during a conversation. Other papers have discussed making VoIP calls more reliable over wireless and mobile networks. Several papers describe new network protocols that have been developed or extensions to existing protocols. Other research has focused on increasing security to prevent spoofing and eavesdropping during VoIP conversations. This chapter surveys the current fields of VoIP research. While all of these fields are important, little research is taking place in the field of embedded VoIP.

3.1 Voice Data Compression

As more people begin communicating using VoIP technologies, compressing VoIP traffic to allow for more VoIP conversations and better coexistence with existing technologies such as transmission control protocol (TCP) continues to

remain important. Wang et al. focus on decreasing bandwidth usage [15]. They explain that multiple voice streams and both voice traffic and TCP traffic must coexist on the same 802.11b wireless network. They propose a method called multiplex-multicast (M-M) that multiplexes voice packets together at the access point into one larger packet that is then multicast to all of the client machines in the wireless network. This decreases the overhead because of the combined packets, leaving more room for additional conversations and for the TCP traffic.

Gokhale and Lu discuss the amount of time that it takes to set up a VoIP call [16]. They tested the time it takes for two user agents (UA) to connect to each other on both the same machine and different machines connected over a LAN. The connections between different machines were established between 4 and 9 milliseconds faster than those on the same machine. This means for call setup time, the network connection is insignificant relative to the processing power of the computer used. This observation could be relevant because the platform for this research is embedded and could cause a slowdown in setup time for each conversation.

3.2 VoIP Over Wireless and Mobile Networks

As discussed in Wang et al., mobile and wireless networks have been an important focus for compression [15]. As mobile broadband and mobile networks have become more ubiquitous, making VoIP available in the mobile realm is the next logical step. (Mobile VoIP provides a cheaper alternative to traditional cell

service, but also does not provide the same level of quality.) Kim proposes extensions to Resource Reservation Protocol (RSVP) called Split Tunnel Based Mobile Resource Reservation Protocol (ST-MRSVP) to achieve a Quality of Service (QoS) guarantee suitable for mobile VoIP [17]. This protocol decreases packet loss during hand-off between different access points in mobile networks. Belhoul et al. describe a Mobility Extension to RSVP (RSVP-ME) [18]. Their method works by allowing end nodes to signal intermediate routers with new nodes and new QoS information for nodes, thereby making RSVP work on dynamic networks in addition to static networks.

3.3 Specialized VoIP Protocols

ST-MRSVP and RSVP-ME are two examples of specialized extensions to the RSVP protocol. Ali et al. list other extensions to RSVP including MRSVP (Mobile RSVP) and DRSVP (Dynamic RSVP) and explain their disadvantages in mobile ad hoc networks [19]. RSVP offers a high level of assurance when it is used in a fixed network environment. It does not scale well and requires quite a bit of processing power. Most of the extensions have had similar limitations. These protocols also require the modification of every node in the path to include the proposed protocol. Huang et al. propose a method that performs congestion and error control to improve quality [20] and only requires changes to the client and the server. This method uses UDP and RTP, which are both established protocols.

3.4 VoIP Security

While compression and delivery guarantees are important, conversation security is also necessary. Butcher et al. discuss security considerations regarding VoIP systems [21]. The majority of the security problems are similar to those present in regular data networks and in traditional phone systems. Some of these problems include denial of service, eavesdropping, alteration of voice stream, call redirection, and caller ID impersonation. They discussed several techniques to mitigate these security problems. Eavesdropping and alteration of voice stream can be prevented by encrypting the traffic between the systems on the network. Call redirection and caller ID impersonation can be prevented by properly authenticating devices on a network using public-private key pairs and by separating the data portion of the network from the voice portion of the network.

Kim et al. also survey the different security threats present in VoIP communications [22]. They propose secure communication procedures to replace the current insecure communications and call negotiations. They also propose secure proxies to handle calls between diverse providers and networks.

3.5 Embedded VoIP

On the embedded front, Ho et al. developed a low-power embedded VoIP processor using a System-On-Chip (SOC) processor for a standalone IP telephone [23]. The standalone IP telephone with its low-power SOC processor consumes less

power than currently available IP telephones. However, this IP telephone still uses a specially designed chip and not off-the-shelf hardware like the embedded IP telephone presented in this document. Hsu et al. implement much of the server-side functionality of SIP on an embedded 32-bit ARM7 network processor [24]. Despite their use of an embedded platform, this implementation is very focused on the software, and the embedded platform only reduces the space, cost, and power consumption compared to servers or desktop computers. In the future, many of these protocols and services could be implemented on Xinu. However, the focus of the current research is the hardware VoIP phone.

3.6 Summary of Related Works

These areas of focus are all important, but they do not cover another important area, embedded systems with limited resources. Gokhale and Lu used a 2.4 GHz Intel Pentium 4 processor with 1 GB of RAM running Windows XP Professional [16]. These resources are unnecessary to establish a single phone call. An embedded platform, namely a Linksys router running the lightweight educational operating system Xinu, can be used to establish and maintain a VoIP call over a network or the Internet. This combination uses commodity hardware that is cheaper than most options available in 2009. The others focus on extending unused protocols or developing their own.

CHAPTER 4

Embedded VoIP Framework

This chapter discusses the hardware and code additions for the 68HC12 microcontrollers and the code written for the Linksys WRT54GL routers. Figure 4.1 presents an overview of the pieces involved in the Internet protocol (IP) phone and how they interact with each other.

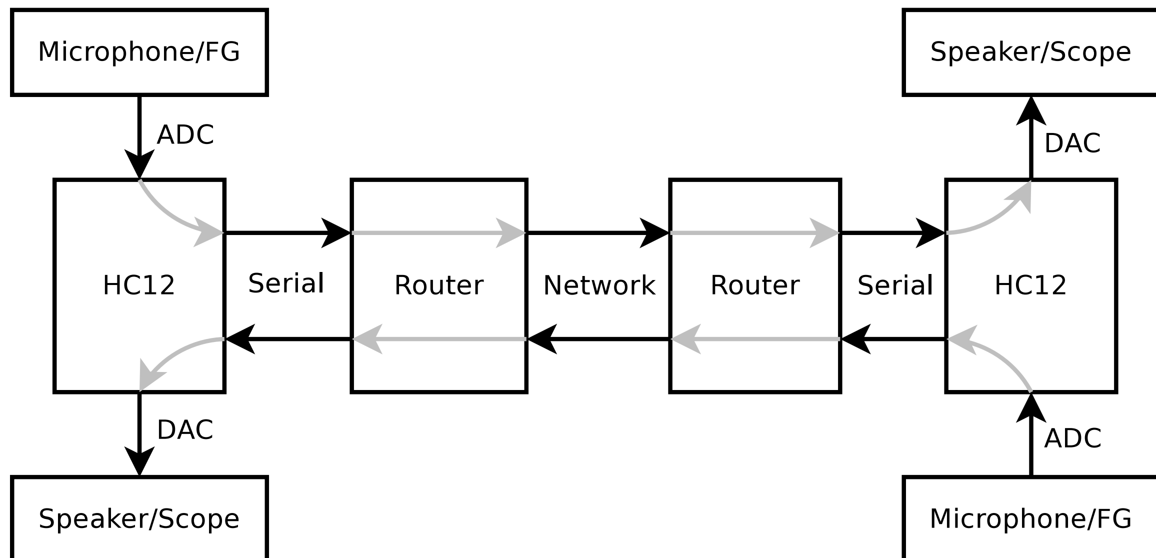

Figure 4.1: Embedded VoIP device connection overview

Figure 4.1 shows two separate and symmetrical processes. The diagram can be divided by a horizontal line through the middle. The top half progresses from left to right, and the bottom half progresses from right to left. The microcontrollers

contain analog to digital converters (ADC), which have microphones (to talk) or function generators (FG) (to test) connected to them. The microcontrollers also have digital to analog converters (DAC), which have speakers (to listen) or oscilloscopes (scopes) (to test) connected to them. The microcontroller contains a serial port to communicate with the router. The router uses its network interface to transmit the data to another router where the process is repeated in reverse. Voice from a microphone is sampled by the ADC on the 68HC12. The data then is transmitted out the serial port to the Linksys router. The router buffers that data and transmits it over a network to another router. That second router transmits the data over its serial port to the second microcontroller. The microcontroller passes the data to the DAC, which reconstructs the voice signal for playback by a speaker.

4.1 68HC12 Microcontroller

The 68HC12 microcontroller was selected because it contains both an ADC and a DAC to sample and reconstruct voice data, a serial port to communicate with the Linksys WRT54GL routers in the Systems Lab at Marquette University, and enough processing power to process and compress an 8 kHz voice signal. As shown in Figure 4.2, the MC9S12DP256 microcontroller on the DRAGON12 development board contains two 8-channel, 10-bit analog-to-digital converters [25] [26] and a LTC1661 chip, which is a dual channel 10-bit serial peripheral interface (SPI) digital-to-analog converter [27] [28]. The ADC and the DAC components are required for voice communication, and since the Linksys WRT54GL routers do not

contain these components, an additional microcontroller is used. The DRAGON12 development board also contains enough breadboard space to build and enough power to drive the additional circuits for a microphone and a speaker. The development board also contains two serial ports on a serial communication interface (SCI). The primary serial port is used as a console to load programs onto and to interact with the microcontroller. The second serial port is used to transmit and receive the sampled voice data.

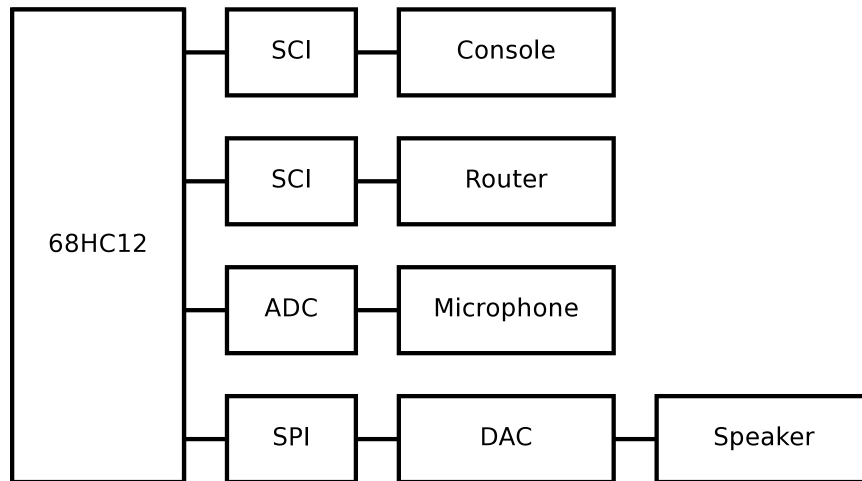

Figure 4.2: 68HC12 hardware overview

4.1.1 Compilation

The GNU Development Chain for the 68HC11 and 68HC12 was used to compile all C and assembly code run on the 68HC12 microcontroller [29]. The microcontroller accepts files in the Motorola S-record (SREC or S19) format [30]. Therefore, downloaded files must be compiled, converted into the S19 format, and

then downloaded onto the board over a serial port. To load both a text segment and a data segment onto the microcontroller, the makefile generates two separate S19 files. Next, the makefile merges the data and text S19 files along with appropriate “LOAD” commands into a single load file that can be sent over the serial port to the microcontroller. The makefile is displayed in Appendix A.

Interrupts on the 68HC12 have posed a problem during both the compilation and the execution phase. Initially, the return from interrupt (RTI) assembly instruction was used in an interrupt service routine (ISR) to ensure that the routine would return properly. The method was simple and compiled without error.

```
void isr(void)
{
    /* Interrupt code */

    asm("rti");
}
```

However, this method of coding interrupts is only sufficient for very simple ISR functions. It does not work if any functions are called from inside of the ISR because it does not build and tear down the stack properly, and the RTI instruction returns too early. To have more complex code inside of an ISR, the makefile was updated so that the ISR function prototypes could be marked with the `interrupt` attribute and could omit the `RTI` instruction, which is then inserted in the proper location by the assembler.

```
void isr(void) __attribute__((interrupt));
```

4.1.2 Code

The majority of the existing example code for the 68HC12 was written in assembly. These examples include good examples for many of the peripherals on the DRAGON12 board but not for the analog-to-digital converter, the digital-to-analog converter, and the serial ports. This meant that despite its wide use, much of the basic functionality of the DRAGON12 board needed for the VoIP phone had to be written in C from the technical manuals for the 68HC12 microcontroller and the other components. In an attempt to make the code as readable, reusable, and extendible as possible, the flags for each register were documented, commented, and used in the code. The code is part of the Marquette University System Laboratory code repository and will be released as part of Xinu.

The code written for the microcontroller includes initialization routines for the ADC, the DAC, the timer, and the second serial port. The ADC initialization routine powers up the ADC, and the DAC initialization routine configures port M to be used as a load input flag for the DAC. Next, it enables SPI and sets master mode. Finally, it sets the load input flag high to disable loading on the DAC. The timer initialization routine enables the timer, sets up timer 7 for output compare, and enables timer interrupts for timer 7. Next, it sets the timer clock to use the bus clock. Finally, it associates an interrupt service routine with the timer 7 interrupt. The SCI initialization routine sets the baud rate to 115.2 kilobaud, sets the character format to 8-bits with one start and one stop, and enables the transmitter and receiver.

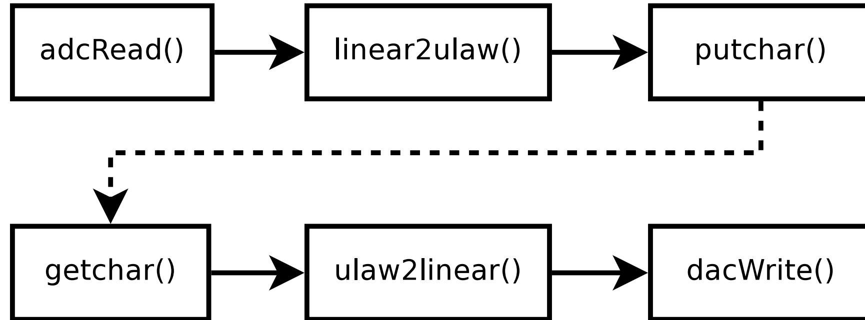

Figure 4.3: 68HC12 timer interrupt code flow

The bulk of the 68HC12 code is in the timer interrupt service routine, which is displayed in Figure 4.3. This sequence of code reads a 10-bit value from the ADC, compresses it to 8-bits using the μ-law algorithm, and sends the byte out the serial port. If a byte is available from the serial port, the routine reads it, decompresses it using the same μ-law algorithm, and writes the 10-bit value to the DAC.

μ-law is a nonuniform quantized compression law [9]. Nonuniform quantization is beneficial for voice because it protects the quieter amplitudes which occurs more frequently and applies more compression to the louder less frequent amplitudes. This makes it possible to encode human voice adequately in as few as 8 bits.

Before the timer interrupt service routine executes the code described above, it increments the timer 7 output compare register and clears the interrupt flag for the timer 7. Because the microcontroller is sampling at 8 kilohertz, the sampling period is 125 microseconds, and because the timer operates at the bus speed which is 25 megahertz [25], the period of the timer is 40 nanoseconds. Therefore, the

theoretical value to increment the timer by is

$$3125 = \frac{125\ \mu s}{40\ ns}.$$

However, the overhead of the running code seems to cause the actual value to be less than the theoretically calculated value. Table 4.1 shows different timer increment offsets and their corresponding delay. For each offset and delay, the timer interrupt tick is calculated. For offsets between 1000 and 9000, the ticks stay constant between 40 and 42 nanoseconds, which is consistent with the calculated period of 40 nanoseconds. If the period were 42 nanoseconds, then the theoretical value to increment the timer would be 2976. In the microcontroller code, the output compare register is incremented by 2937 so that the next timer interrupt will fire 125 microseconds later. This value was arrived at by experimentation, but it correctly produces 8 kilohertz sampling on two different 68HC12 microcontrollers. Serial receive rates for output from the router of the sampled signal on the microcontroller are displayed in Appendix B. The overall average for the receive rates is 8004.39 bytes per second, which means the sampling rate of the microcontroller is also 8004.39 Hertz.

The transmit portion of the interrupt code starts by calling `adcRead()`, which initiates a right-justified unsigned 10-bit single-channel conversion from the ADC. Next, it waits until the conversion completes, and then it returns the value from the conversion. The value returned from `adcRead()` is converted from an

Offset	**Delay (μs)**	**Tick (ns)**
16384	700	42.7246
8192	330	40.2832
4096	168	41.0156
4000	164	41
3500	144	41.1429
3038	125	41.1455
2048	84	41.0156
1024	43	41.9922
512	22.5	43.9453
256	12	46.875
128	7	54.6875
64	4.3	67.1875
32	3	93.75
16	2.4	150

Table 4.1: 68HC12 Timer Interrupt Tick Resolution

unsigned 10-bit value to a signed 16-bit value such that the

$$[\text{16-bit signed value}] = ([\text{10-bit unsigned value}] - 512) * 64.$$

Next, the value is passed to `linear2ulaw()`, which uses the μ-law algorithm to compress the signed 16-bit value to an unsigned 8-bit value. The algorithm compresses low magnitude values less than it compresses high magnitude values. The compressed value then is passed as an argument to `putchar()`, which reads from the SCI status register to clear it and then writes the byte of data to the SCI data register.

After the transmission process is complete, the receive process begins. `getchar()` is not so much a function as a check to make sure that a byte of data is in the buffer. If there is a character, it is removed from the queue and passed to

`ulaw2linear()`, which uses the μ-law algorithm to decompress the unsigned 8-bit value to a signed 16-bit value. The signed 16-bit value then is converted to an unsigned 10-bit value such that the

$$[\text{10-bit unsigned value}] = \frac{[\text{16-bit signed value}]}{64} + 512.$$

Next, the value is passed to `dacWrite()`, which waits until the transmitter is empty and then sends a 4-bit control code to load DAC A and perform a DAC update along with the first 4 bits of the value. The function waits until the transmitter is empty and then sends the last 6 bits of the value. The SPI flags are cleared, and the load bit is set high so the DAC will process and display the value.

The serial communication interfaces on the 68HC12 have no hardware buffering [31]. Interrupts would be an ideal solution to handle incoming and outgoing SCI data. However, interrupts do not behave in the expected manner on the 68HC12 because of a bug in the SCI hardware interrupt structure which causes the interrupt flags to be added together instead of OR'd together [32]. Therefore, if the number of interrupts is even, a zero is returned when a one should have been returned. A software receive buffer must be used to handle the incoming stream of bytes from the Linksys router. Because bytes are never transmitted at a faster rate than 115.2 kilobaud, no software transmit buffer is necessary. The receive buffer greatly reduces the buffer overrun errors that the SCI experienced with no receive buffering. This is further discussed in Section 6.1.2. The buffer is filled by polling in a loop in the main program. Bytes are read out of the buffer during each interrupt.

4.1.3 Analog and Digital Conversion

The 68HC12 microcontroller contains the ADC and DAC components necessary for sampling and reconstructing voice data. For input into the ADC, an Apple PlainTalk microphone was used. This microphone has an extra-long, 4-conductor plug which accepts 5 volts at the tip and a ground reference at the base [33] [34]. The middle connectors are the microphone signal. The PlainTalk microphone was selected because it outputs a line level signal by using an internal preamplifier. Therefore, it requires no additional amplification before it is connected to the ADC. We also hoped that it would be simple to connect another line-level audio source in place of the microphone, but it turns out that it would require additional work to match the impedance or build a voltage shifting operational amplifier circuit.

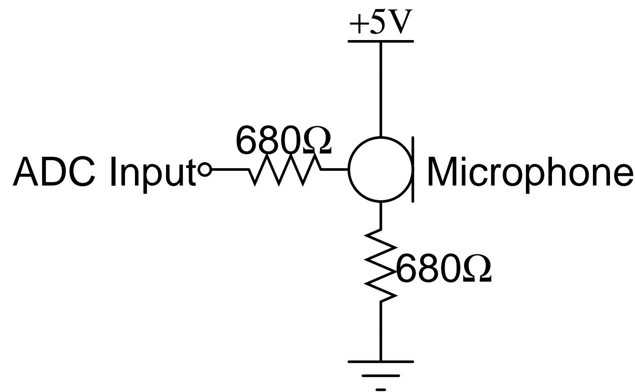

Figure 4.4: ADC microphone circuit

The PlainTalk microphone output voltage typically ranges between -1 and +1 volts peak-to-peak. The ADC low and high reference voltages on the

DRAGON12 board are set initially to 0 volts and 5 volts, respectively. Therefore, a 680 ohm resistor was placed between the ground on the microphone and the ground on the development board, causing the microphone to see a ground signal of 1 volt. The circuit is shown in Figure 4.4. The trace was cut between the 5 volt signal and the high reference voltage for the ADC and was connected instead to a 2 volt signal using a simple voltage divider circuit. The result was a microphone with a center at 1 volt, and an ADC that ranged between 0 and 2 volts.

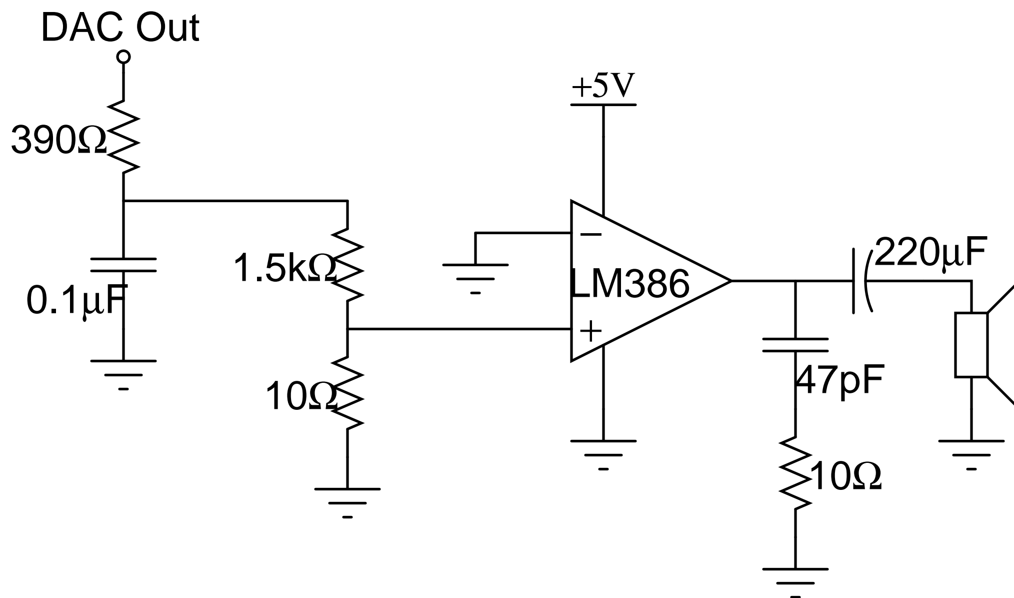

Figure 4.5: DAC speaker circuit

A LM386 amplifier was used to connect a speaker to the DAC [35]. This protects the development board from current overdraw and amplifies the signal before being sent to the speakers. The final circuit connected to the DAC output included a simple resistor-capacitor first-order low-pass filter to smooth the edges of the signal, a voltage divider to step the voltage down before feeding it into the

amplifier, and a capacitor-resistor circuit to attach the speaker. The circuit is shown in Figure 4.5.

The simple low-pass filter has a cutoff frequency of 4 kilohertz. From the Nyquist Sampling Theorem, the sampling frequency must be twice the highest frequency in the signal [10]. Therefore, since the ADC is sampling at 8 kilohertz, the maximum frequency in the signal is 4 kilohertz, and the low-pass filter should eliminate all higher frequencies.

4.1.4 Power Cycling

The remote power control device in the System Laboratory uses a standard computer power supply and mechanical relays to provide 12 volts to the Linksys routers. The Xinu power daemon allows students and researchers to power cycle routers in the pool without needing to plug and unplug them manually. To make the 68HC12 microcontrollers more useful in the lab, they needed to be able to be power cycled using the rebooter. However, they require 9 volts and not 12 volts. Therefore, we built a small board that used the 12 volt line from the rebooter to flip another mechanical relay that connected the 9 volt wall transformer to the microcontroller as shown in Figure 4.6.

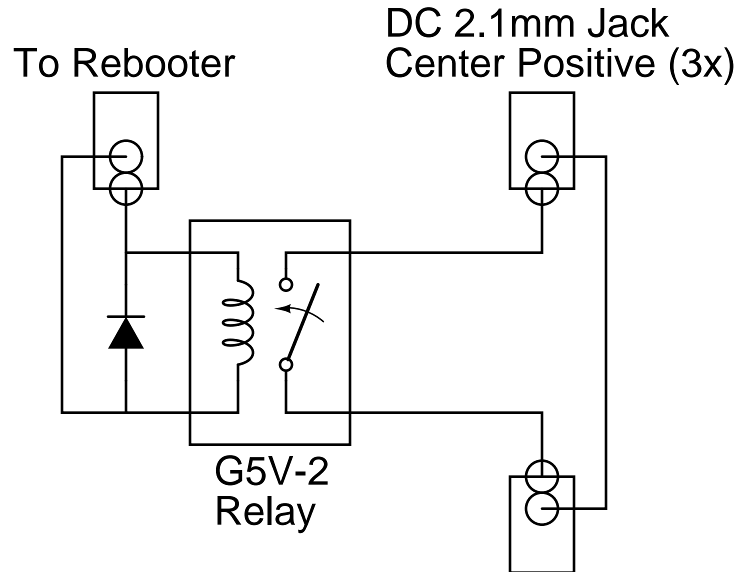

Figure 4.6: Microcontroller rebooter circuit

4.2 Linksys WRT54GL Router

This research began with the idea of building an IP telephone around the existing Linksys WRT54GL routers that are part of the current Marquette University Systems Lab infrastructure. The WRT54GL router already has the holes for headers for two serial ports. Once the serial transceiver and the serial ports are attached, the primary serial port is used as the console to interact with the firmware and operating system running on the router. The second serial port is free to be used for communication with the 68HC12 microcontroller.

4.2.1 Xinu Operating System

The Xinu operating system runs on the Linksys WRT54GL routers. Xinu provides the necessary operating system and network functionality to send and

receive data over the serial port and transmit and receive packets over the network.

4.2.2 Serial Communication

The Linksys router runs two threads for serial and network communication. One reads data from the serial port and transmits it over the network, and the other reads data from the network and transmits it over the serial port. As described above in the 68HC12 microcontroller section, the microcontroller transmits a byte over the serial line to the router. In the thread `basic_send()`, the router asynchronously reads bytes into a buffer before transmitting the full buffer to the network destination in a UDP (User Datagram Protocol) packet.

For both sending and receiving, the buffer size has been set to 1024 bytes. For our private network and network stack, experimentation showed that this was an appropriate size that was not too small to flood the network with tiny packets and not too large that the packet did not arrive at its destination in time. Other networks could produce different results. The read call to the serial device is blocking, and the thread sleeps while it waits for enough characters to fill the buffer.

```
thread basic_send(ushort uart, ushort udp)
{
    uint len = 0;
    uchar buf[BUF_SIZE];

    /* Enable all interrupts */
    enable();

    while (TRUE)
    {
        /* Read from the serial device */
```

```
        len = read(uart, buf, BUF_SIZE);

        if (len > 0)
        {
            /* Write to the UDP device */
            write(udp, buf, len);
        }
    }
}
```

The thread `basic_receive()` behaves similarly to the send thread. It reads the data from UDP packets and transmits the bytes over the second serial port to the microcontroller. The call to read from the UDP device is blocking, so the thread sleeps while it waits for incoming data from the network.

```
thread basic_receive(ushort uart, ushort udp)
{
    uint len = 0;
    uchar buf[BUF_SIZE];

    /* Enable all interrupts */
    enable();

    while (TRUE)
    {
        /* Read from the UDP device */
        len = read(udp, buf, BUF_SIZE);

        if (len > 0)
        {
            /* Write to the serial device */
            write(uart, buf, len);
        }
    }
}
```

4.2.3 Network Transmission

The basic send and receive threads described in the previous section use a buffer size of 1024 bytes. Transmitted packets contain no sequence information. Therefore, the receiving thread makes no attempt to reorder packets or interpolate for missing packets. Attempts to improve this are discussed in Section 5.5.

4.3 Desktop Tools

While the focus of this research is VoIP on embedded systems, communication with other devices such as the desktop computer is also important, particularly for benchmarking and gathering results. Since the Xinu operating system implements the standard UDP protocol, and the audio on the routers is in a μ-law compressed raw format, existing tools can be used to decode the audio and play it on a desktop computer. In much the same manner, audio can be encoded and transmitted to the router. All of the desktop work was performed on Fedora Core Linux computers.

`aplay` was used to decode and play the audio received over the network [36]. Its matching counterpart, `arecord` was used to record and encode audio and transmit it over the network. `aplay` and `arecord` are command line sound recorder and player tools that work with the ALSA soundcard driver. The tools are simple, handle raw audio well, and can handle μ-law compression without any additions or modifications.

The `nc` or `netcat` utility was used for the actual network transmission over UDP on the desktop computer [37]. We used `netcat` to listen for UDP data on a specified port and then hand it off to `aplay`. The following command line listens on UDP port 22020 and plays received data as a raw file in the μ-law format at 8 kilohertz.

```
nc -u -k -l 22020 | aplay -f MU_LAW -t raw -r 8000
```

We also used `netcat` to receive data from `arecord` and transmit the data on a specified port over UDP. The following command line records audio from a microphone at 8 kilohertz, compresses it in the μ-law format, and sends it to "host" on UDP port 22030.

```
arecord -f MU_LAW -t raw -r 8000 | nc -u host 22030
```

While `netcat`, `aplay`, and `arecord` were used primary to interact with routers running the Xinu operating system, it is worth noting that it is possible to use these tools to communicate between two desktop computers. This could be helpful for establishing a point of comparison and for testing any additional tools written to run on a desktop computer instead of on a router running Xinu.

Separating the transmission and receiving of data from the playing and recording of data means that `netcat` can also be used to send arbitrary data such as a sine wave and record and analyze incoming data.

During much of the network development and testing, `netcat` made it possible to use a single microcontroller and router. `netcat` turned the desktop

computer that was receiving the UDP traffic into an echo server that simply piped the audio back to the router. The following receives UDP traffic on port 22020 and sends it back to "host" on UDP port 22030.

```
nc -u -k -l 22020 | nc -u host 22030
```

In addition to command line tools already available, we wrote several simple tools for data generation and analysis and benchmarking including `sine`, `ulaw`, and `human`. `sine` generates a one kilohertz sine wave that either can be played through an audio out jack and fed into the ADC on the 68HC12 or compressed using `ulaw` and then transmitted over the network to a router. `sine` can output in either 8-bit or 16-bit samples. Since these tools all handle data through standard input and output, 16-bit output is performed by writing two characters. The first character is the first 8-bits of the sample shifted down, and the second character is the second 8-bits of the sample.

In a similar fashion to `sine`, the `ulaw` tool reads data and either compresses or decompresses it before writing it back out. `ulaw` either takes a 16-bit uncompressed sample and outputs an 8-bit compressed sample, or it takes an 8-bit compressed sample and outputs a 16-bit decompressed sample. It can be used to process data from `sine` or data received over the network from `netcat`.

Because the "sampling" of a digitally generated sine wave is exact, only particular harmonics are represented by the sampled data as shown in Figure 5.22. When this data is compressed and decompressed using the μ-law algorithm, it looks

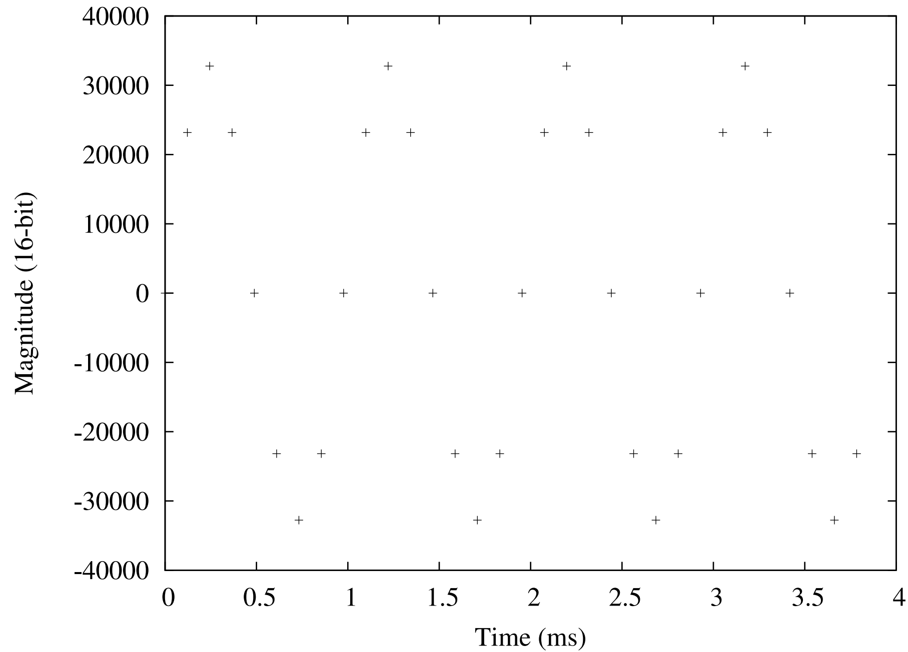

Figure 4.7: 1 kHz Sine Wave Sampled at 8 kHz

and sounds slightly different because the values of the samples have been altered slightly but consistently. This is something to keep in mind when using a strictly digital system where an analog system is traditionally used. The effect of the μ-law compression is displayed in Figure 4.8. The differences in the values before and after compression are shown in Table 4.2.

Sample	**Sine**	**μ-law**
1	0	0
2	23164	22908
3	32760	32124
4	23164	22908
5	0	0
6	-23164	-22908
7	-32760	-30076
8	-23164	-22908
9	0	0

Table 4.2: Effect of μ-Law Compression on a Sine Wave

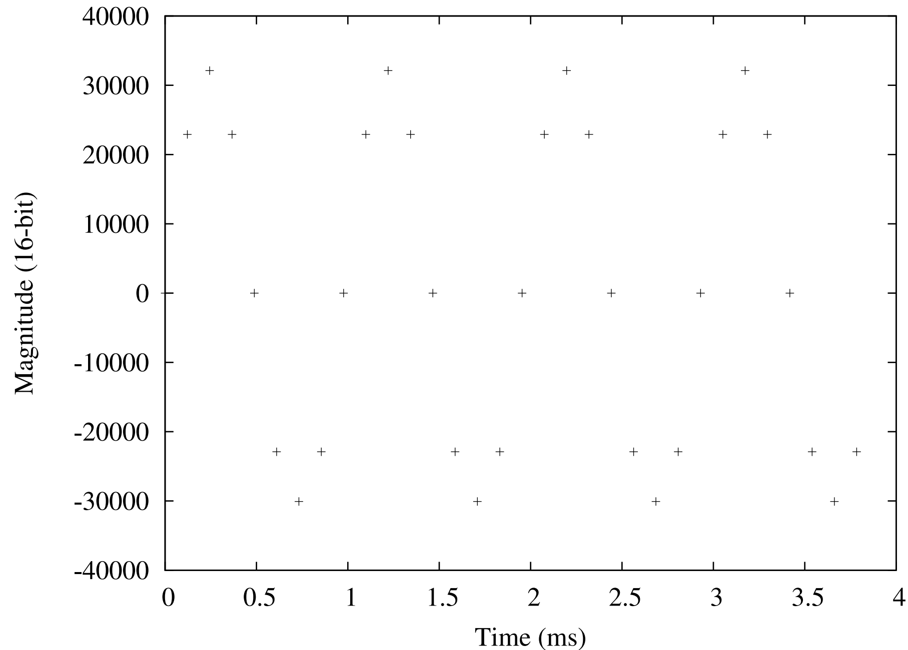

Figure 4.8: 1 kHz Sine Wave Sampled at 8 kHz After μ-Law Compression

The final tool is called `human`. It accepts 8-bit or 16-bit input and outputs a human readable form of the data. This is useful for either looking at the data or plotting it on a graph.

4.4 Summary of Framework

This chapter describes the assembly of an IP telephone using off-the-shelf embedded systems, some hardware modifications, and some code additions. The Linksys routers were selected for their network interfaces and dual serial ports and because of their existing use in the Marquette University System Laboratory. The 68HC12 microcontroller was selected because of its established use in the field and because it had serial ports and voice conversion capabilities. After the hardware was

connected together, software was written to sample and compress voice into bytes and transmit it over the serial line to the router where it was transmitted over the network to another router which received the data and transmitted it over the serial line back to the microcontroller which decompressed it and played it back through a speaker.

CHAPTER 5

Performance Analysis

Chapter 4 described how the embedded IP telephone was built. This chapter describes how we quantified the system's performance. However, because this is a *voice* over IP device, much of the initial testing and analysis was as simple as listening to the sound from the speakers. After the system began passing the ear test, we evaluated the latency and the speech quality.

5.1 Embedded VoIP Latency

Latency is one of the most important measurements of a real-time system such as this IP telephone. In addition to determining the overall latency of the system and its connections, we decomposed it and determined the latency of each of the components and connections. Each of the arrows in Figure 5.1 represents a connection or process that has a latency associated with it.

To begin the testing process, we stripped the microcontroller code back to its simplest form: reading a value from the ADC and writing it to the DAC, which is displayed below. This reveals the delay for a signal to get from the input of the ADC to the output of the DAC including any code delay, which we assume to be

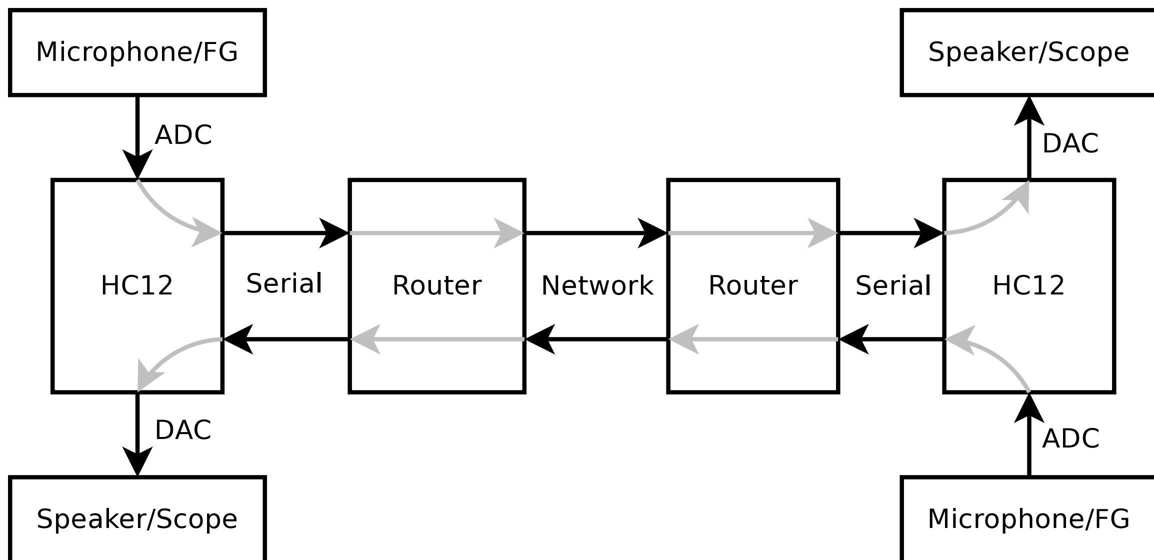

Figure 5.1: Embedded VoIP device connection and data path flow overview

insignificant.

```
int main()
{
    /* Initialization code */

    while (1)
    {
        dacWrite(adcRead());
    }
}
```

Figure 5.2 displays the active pieces of the system and the data path for testing of the ADC and DAC latency. In this arrangement, the function generator produces a 1 kilohertz sine wave, which is fed into the ADC on the microcontroller. The signal is sent out the DAC to the oscilloscope, which also has a second line in directly from the function generator. Figure 5.3 shows a 1 kilohertz sine wave and the same sine wave after the 92 microsecond delay described by Figure 5.2 and the code above. After μ-law companding was added, the delay increased to 131

microseconds as shown in Figure 5.4. Interrupts for 8 kilohertz sampling increase the delay to 140 microseconds as shown in Figure 5.5.

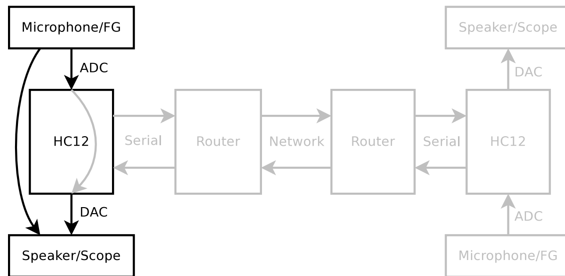

Figure 5.2: Data path flow for ADC and DAC latency test case

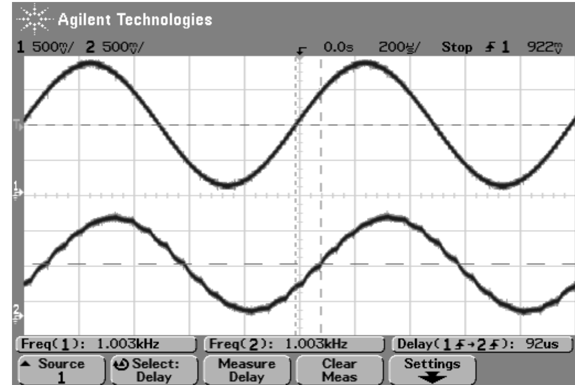

Figure 5.3: Original 1 kHz sine wave (top) and sine wave delayed 92 μs by ADC and DAC (bottom)

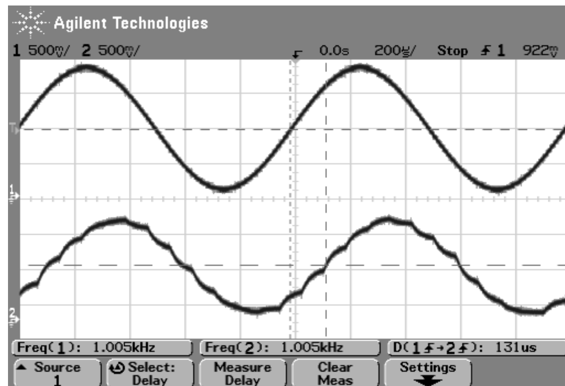

Figure 5.4: Original 1 kHz sine wave (top) and sine wave delayed 131 μs by ADC, DAC, and μ-law companding (bottom)

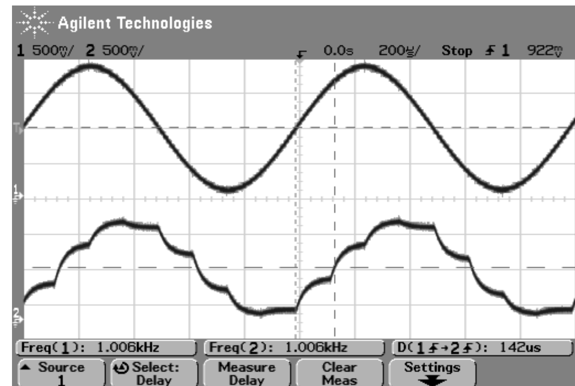

Figure 5.5: Original 1 kHz sine wave (top) and sine wave delayed 142 μs by ADC, DAC, μ-law companding, and 8 kHz sampling interrupts (bottom)

The next step in the latency testing process was to add the serial port on the microcontroller. The serial port has a loopback flag that can be turned on. This data flow arrangement is displayed in Figure 5.6, and the 389 microsecond delay is

displayed in Figure 5.7.

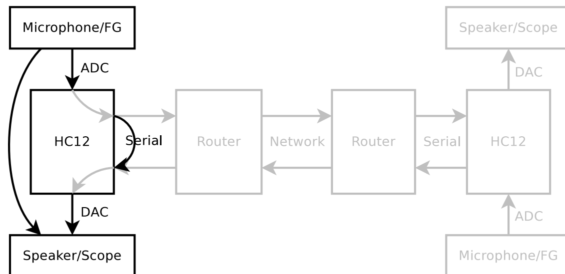

Figure 5.6: Data path flow for latency measurement through the serial loopback device on the microcontroller

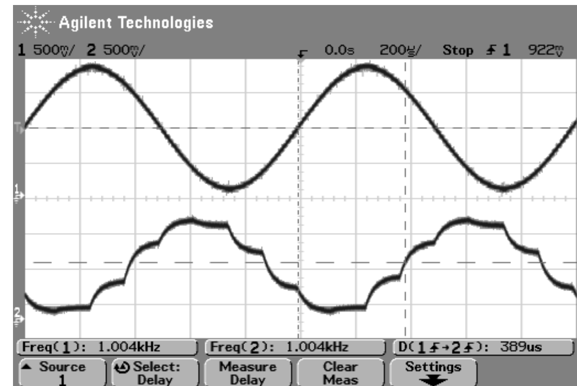

Figure 5.7: Original 1 kHz sine wave (top) and sine wave delayed 389 μs by the serial loopback on the microcontroller (bottom)

After loopback on the serial port has been tested, we add in the router. The router has a very simple thread that reads a byte from the serial port and writes it back out to the serial port effectively creating a "router loopback." The data flow is displayed in Figure 5.8, and the 574 microsecond delay is displayed in Figure 5.9.

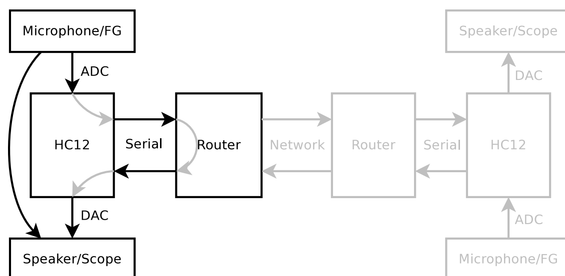

Figure 5.8: Data path flow for latency measurement through the serial ports with loopback on the router

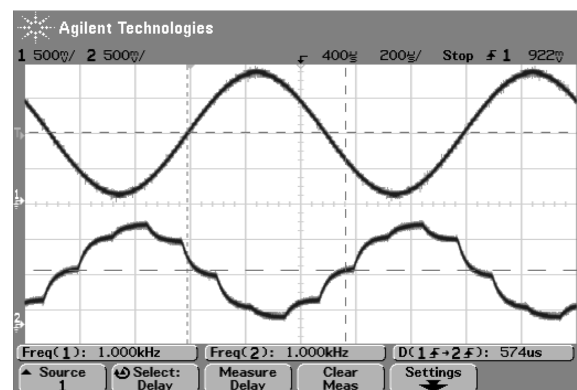

Figure 5.9: Original 1 kHz sine wave (top) and sine wave delayed 574 μs by the serial ports and router loopback (bottom)

Now that the test cases have produced longer latencies, it is important to make sure that the signal is not "wrapping" around and displaying as a shorter latency than the actual latency of the system. The "router loopback" test case had a latency greater than 500 microseconds, which is half of the time before it would wrap around on the oscilloscope because the 1 kilohertz sine wave has a period of 1 millisecond. Therefore, a new program for the microcontroller was needed. Instead of reading from the ADC, it generates a character of data on its own, sends it over the serial port, and then waits for it to return on the serial port. When the character returns, it sends it out again and toggles the DAC between high and low values. A simplified version of the code is shown below.

```
int main()
{
    /* Initialization code */

    putchar1('a');

    while (1)
    {
        if (/* SCI character available */)
        {
            temp = getchar1();
            putchar1(temp);
            dacWrite(/* toggle between high and low value
               */);
        }
        if (/* SCI overrun occurred */)
        {
            /* clear SCI overrun */
        }
    }
}
```

We repeated the "router loopback" test case that was described above to

ensure accurate results and to show that the new code works. The subtly modified data path is displayed in Figure 5.10, and the square wave that the DAC toggling produces is displayed in Figure 5.11. The delay is only 563 microseconds, which is 13 microseconds less than the previous measurement because there is now no overhead for the ADC and the DAC. (Even though the DAC is being used, its use does not generate any overhead.)

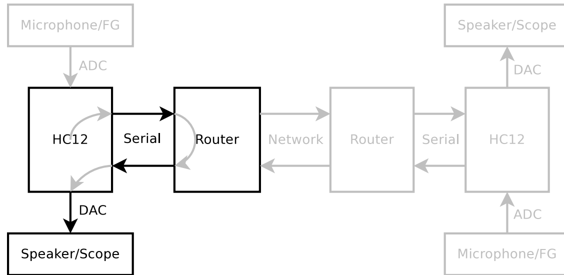

Figure 5.10: Data path flow for latency measurement through the serial ports with loopback on the router (internal measurement using returned data from the serial port)

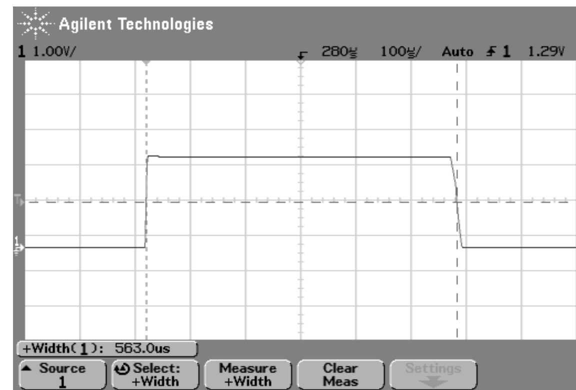

Figure 5.11: Square wave representation of the 563 μs delay due to the serial ports and the router loopback

Next, we add the Ethernet loopback device to the system. This data path is displayed in Figure 5.12, and the 891 microsecond delay is displayed in Figure 5.13. This test case again uses the method described for the second "router loopback" test.

The final test case was using the entire system of two microcontrollers, two routers, and the network. The network was a private local network, and both routers were connected directly to the same switch. Therefore, the delay due to this

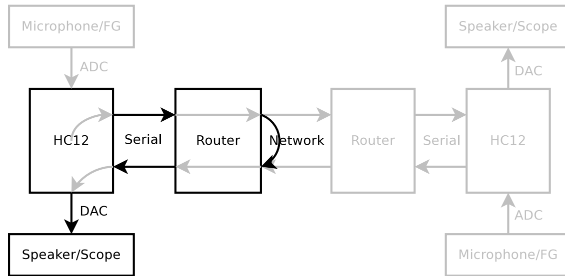

Figure 5.12: Data path flow for latency measurement through the Ethernet loopback device

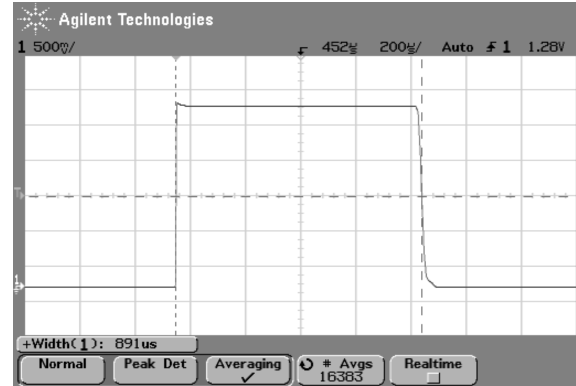

Figure 5.13: Square wave representation of the 891 μs delay due to the addition of the Ethernet loopback device

network is negligible compared with travel across the Internet. The data path is displayed in Figure 5.14, and the 1.875 millisecond delay is displayed in Figure 5.15. Because this is the roundtrip latency, the latency of the actual data path is half or about 938 microseconds. This number seems reasonable since it is slightly greater than that of the Ethernet loopback test.

At this point in the testing process, we know the entire system works. Therefore, we replaced the function generator and the oscilloscope with microphones and speakers. Two individuals were able to carry on a conversation using the two IP telephones. It is a return to the "ear test," but it is an important metric to demonstrate that the system works for its intended purpose.

The results of the different latency tests discussed in this section are summarized in Table 5.1. From the results in the table, we can conclude that the overall latency for the system is the ADC, DAC, μ-law companding, and 8 kilohertz

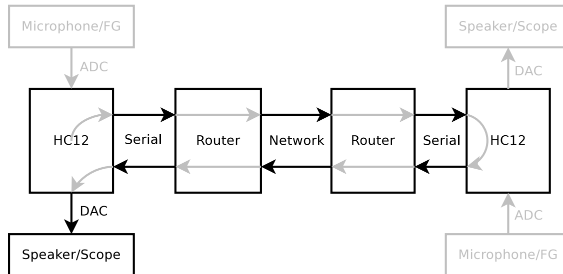

Figure 5.14: Data flow for roundtrip latency measurement using two microcontrollers, two routers, and a network

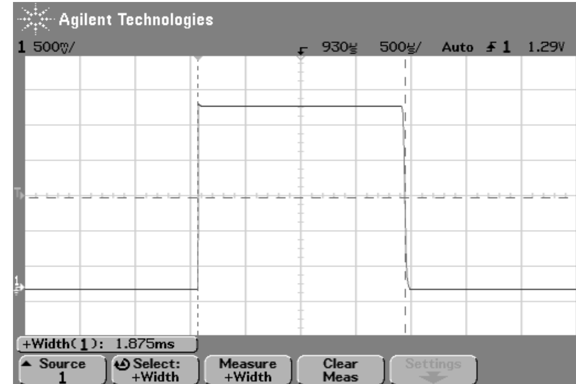

Figure 5.15: Square wave representation of the 1.875 ms delay for the round trip latency test case

sampling latency and half of the roundtrip latency, which is

$$1.079\ ms = 142\ \mu s + \frac{1.875\ ms}{2}.$$

Only the two measurements are necessary because the network roundtrip covers all of the pieces that the different loopback tests cover. The latency measurements taken from the oscilloscope were averaged to produce the best measurement. However, the deviations on the last three tests were estimated based on how the measurements on the oscilloscope fluctuated when not being averaged. In terms of VoIP, this deviation is better known as jitter.

5.2 Desktop VoIP Latency

In order to make sense of the embedded VoIP latency results in Section 5.1, they must be compared to something else. This section gathers some basic latency

Test case	Total latency (μs)	Deviation (μs)	Individual latency (μs)
ADC and DAC	92		92
μ-law companding	131		39
8 kHz sampling	142		11
Serial loopback	389		247
Router loopback	574		185
Router loopback (No ADC/DAC)	563	±6	185
Ethernet loopback	891	±11	328
Network roundtrip	1875	±15	375

Table 5.1: Summary of latency measurements from shortest data path to longest data path

results for a similar desktop equivalent. Figure 5.16 provides an overview of how the desktop testing system was arranged, which nicely mirrors the arrangement for the embedded system.

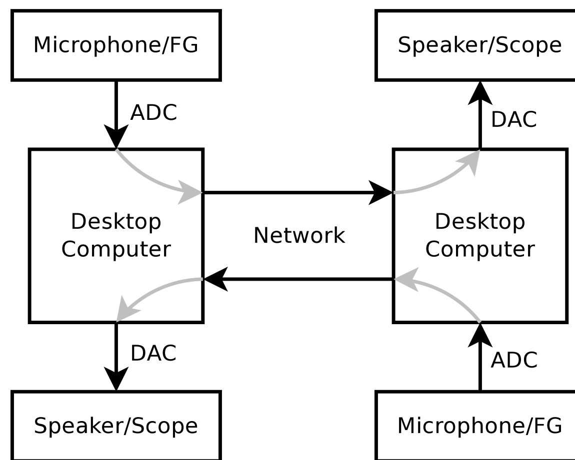

Figure 5.16: Desktop VoIP connection and data path flow overview

The first test we performed on the desktop system used only one of the computers as displayed in Figure 5.17. This test shows the latency for the computer to sample and reconstruct an audio signal. Using a somewhat distorted sine wave

provided by a third desktop computer, we were able to determine the latency to be 420 microseconds as displayed in Figure 5.18.

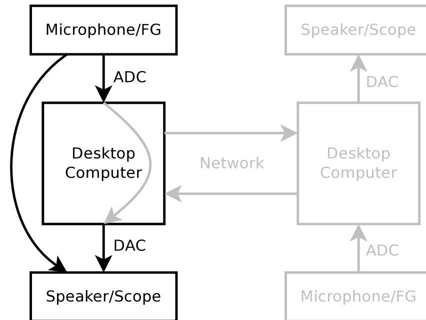

Figure 5.17: Data flow for desktop computer ADC and DAC latency test case

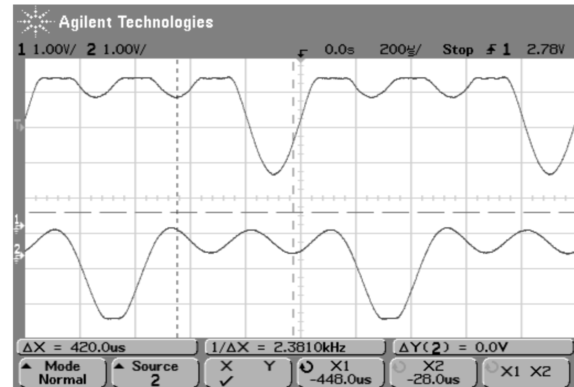

Figure 5.18: Distorted square wave representation of the 420 μs delay for ADC and DAC latency test case

The second and final test is displayed in Figure 5.19 and uses two computers connected via a network. The second computer has an audio cable that connects the speaker output to the microphone input effectively creating a loopback. The results of this test show a roundtrip time of 892 microseconds, which means that one way time is 446 microseconds.

To summarize, for local processing, our embedded VoIP system has a latency of 574 microseconds, and the desktop VoIP system has a latency of 420 microseconds. Over a network, our embedded VoIP system has a latency of 1.079 milliseconds, and the desktop VoIP system has a latency of 446 microseconds. This shows that much of the delay in our embedded system is in the network hardware or network stack code.

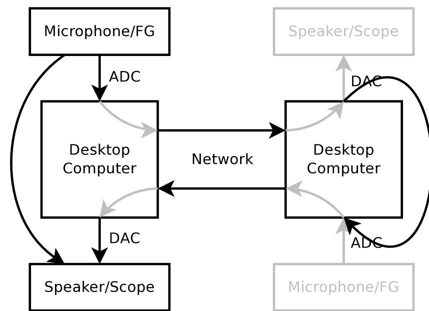

Figure 5.19: Data flow for roundtrip latency test case using two desktop computers and a network

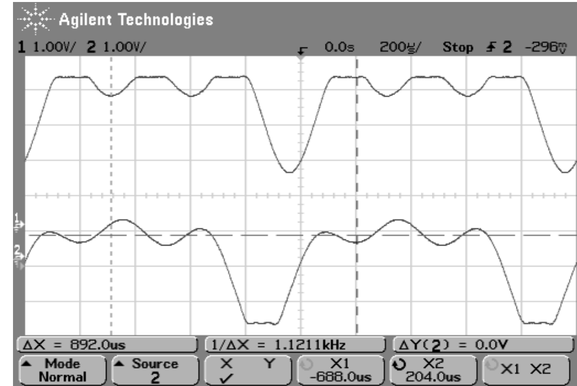

Figure 5.20: Distorted square wave representation of the 892 μs delay for the roundtrip latency test case using two desktop computers and a network

5.3 Embedded VoIP Voice Quality

In addition to latency, voice quality is another important metric of a VoIP system. Perceptual Evaluation of Speech Quality (PESQ) is an ITU (International Telecommunication Union) recommendation for a method of objectively assessing the quality of voice data [38]. As a full-reference algorithm, it analyzes the difference between the originally transmitted signal and the final received signal. The analysis is represented as a mean opinion score (MOS). The MOS is a value between 1 (poor quality) and 5 (excellent quality).

From a qualitative perspective, the sampling portion of the phone performed adequately well, leaving the reconstruction and output as the performance bottleneck. Figure 5.21 shows the test setup for evaluating speech quality. The pre-recorded signals were played through a computer, which transmitted the audio over the network to the router. The router transmitted the signal to the

microcontroller which reconstructed it. From there, it was recorded back into another computer to be compared with the original signal.

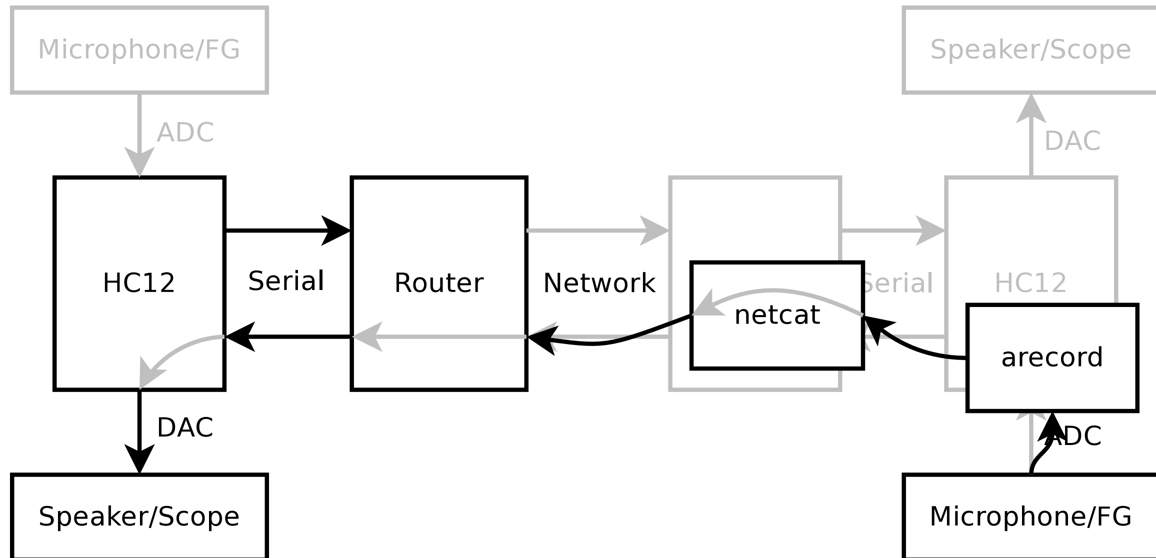

Figure 5.21: Data flow for embedded speech quality testing

Our tests revealed a MOS of 3.13, which is considered fair. The test results are summarized in Table 5.2. A score of 3.13 shows that a simple and inexpensive first order low-pass filter for reconstruction gets our embedded telephone reasonably close to 3.8, which is toll quality [39]. Our PESQ MOS of 3.13 is competitive with Skype (2.988), WinLive (3.149), GTalk (3.278), and Yahoo (3.472) IP telephony software under similar, ideal network conditions [40].

Mean	3.13
Max	3.33
Min	2.74
Standard Deviation	0.15

Table 5.2: PESQ MOS test results summary with sample size of 23

5.4 Sources of Error

This section analyzes the different sources of error in the system and how they have been mitigated or how they could be mitigated. Sources of error include sampling and digitization, clock synchronization and skew, lossy compression, and transmission over the serial line and the network.

5.4.1 Sampling and Digitization

After a signal is sampled, some amount of data is lost due to imperfect reconstruction. Figure 5.22 and Figure 5.23 show such sampling error. In both figures, the top waveform is the original 1 kilohertz sine wave. In Figure 5.22, the bottom waveform has been sampled by the ADC and reconstructed by the DAC. Figure 5.23 adds μ-law companding and interrupts for 8 kilohertz sampling. The decrease in sampling rate has a noticeable effect on the size of the blocks and the overall quality of the signal.

While this uncorrected signal may be decipherable, a simple low-pass filter circuit can decrease the error. This circuit is displayed in Figure 4.5 and described in Section 4.1.3. Figure 5.24 shows the same sine wave as before but with the addition of the smoothing circuit attached to the DAC output. As before, the sine wave has been sampled at 8 kilohertz and had μ-law companding applied to it.

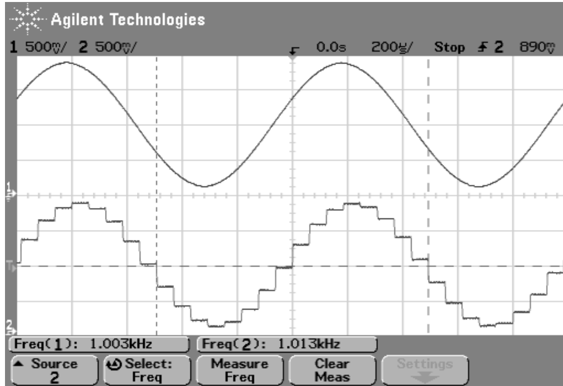

Figure 5.22: Original sine wave (top) and sampled sine wave with no correction (bottom)

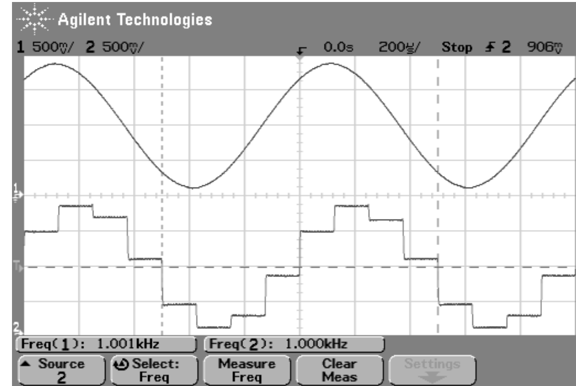

Figure 5.23: Original sine wave (top) and sampled sine wave after μ-law companding and 8 kHz sampling interrupts with no correction (bottom)

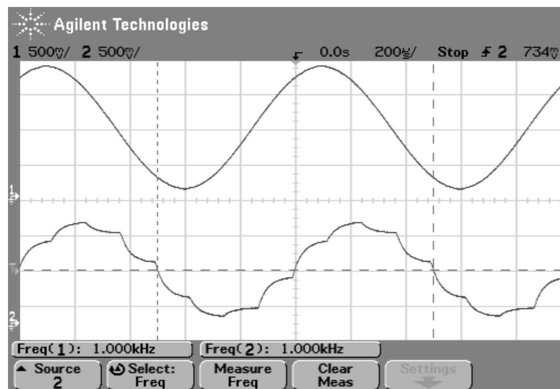

Figure 5.24: Original sine wave (top) and sampled sine wave after μ-law companding and 8 kHz sampling interrupts with smoothing capacitor correction (bottom)

5.4.2 Clock Synchronization

The system is composed of four separate clocks, which leaves plenty of room for clock skew to occur. However, because the routers simply pass the data from the microcontroller to the network, their clocks become less important, and the deviation in the clock speed of the microcontrollers becomes the focus.

The crystal that controls the frequency on the microcontroller runs at 25

MHz and probably has a frequency tolerance of 50 PPM (parts per million). Therefore, it could vary in either direction by up to

$$1.25\ kHz = 25\ MHz * 0.005\%.$$

The scaling factor for 8 kilohertz sampling is

$$3125 = \frac{25\ MHz}{8\ kHz}.$$

This means that the 8 kilohertz sampling could vary in either direction by up to

$$0.4\ Hz = \frac{1.25\ kHz}{3125}.$$

Therefore, a single sample's period could vary by

$$6ns = \frac{1}{8000\ Hz} - \frac{1}{8000.4\ Hz}.$$

Over a second, the sampling could vary by

$$48\ \mu s/s = 6\ ns * 8000\ Hz,$$

which means that after

$$2.6\ s = \frac{125\ \mu s}{48\ \mu s/s},$$

the two microcontrollers could be one sample out of sync, causing overrun or

underrun. In a way buffer underrun is self-correcting or self-synchronizing. However, buffer overrun requires additional efforts to correct. We use a large buffer to postpone the occurrence of buffer overrun. This is sufficient for “short” phone calls, which do not completely fill up the buffer. For longer phone calls, a method of synchronization similar to what is used in RTP and RTCP would need to be used.

5.4.3 Compression

The microcontrollers perform μ-law compression on the voice data before transmitting it. This is a form of lossy compression because it does not preserve the original signal during the compression. When the signal is decompressed, it will not be identical to the original signal before it was compressed.

The μ-law compression algorithm was selected because it is non-linear and preserves the important ranges for human voice. The μ-law algorithm increases the signal-to-noise ratio and is used in PSTN phone systems in North America and Japan [39]. The most common alternative is A-Law, which is used in Europe.

5.4.4 Serial

The connection between the serial ports on the microcontroller and the router provide another point for data loss and delay. However, unlike an Ethernet network, which must be shared with other devices, most of the serial overrun, underrun, loss, and delay can be avoided through the use of proper asynchronous

calls and interrupts. Some of these coding techniques are discussed in Section 6.1.2. The serial link can also be affected by interference in the line or under-powered charge-pump transceivers.

5.4.5 Network

The network that connects the two routers is the most variable segment of the trip. It is prone to data loss, delay, and duplication. All three of these are discussed in the next section. Data duplication is easiest to deal with. It is possible to detect if a packet of data has already been received and simply drop it. Data loss and delay are more difficult. Delayed data may still be able to be used, but in real-time systems, there becomes a point at which a delayed packet might as well have been lost because it simply has arrived too late. Lost data can be dealt with by repeating data or trying to interpolate between packets. The next section discusses a basic solution to some of these problems.

5.5 Network Robustness

As discussed in the previous section, networks pose the greatest source of error, especially as the number of nodes on the network and the number of hops between the source and the destination increases. This section describes some simple code changes that attempt to handle data loss, delay, and duplication.

Instead of just sending the buffer of voice data as the UDP packet payload,

we send a structure that includes a sequence number, the length of the data, and the data itself, which is described by `struct voipPkt`.

```
struct voipPkt
{
    uint seq;
    uint len;
    uchar buf[SEQ_BUF_SIZE];
};
```

In Section 4.2.2, we described basic threads for transmitting and receiving data called `basic_send()` and `basic_receive()`. These basic threads have been extended to include provisions for sequence number and timing. The new `seq_send()` thread only needs to keep track of what sequence number it is on and include it in the packets it sends. A simplified version of the code is below.

```
thread seq_send(ushort uart, ushort udp)
{
    /* Variable declarations and initializations */

    while (TRUE)
    {
        /* Read from the serial device */
        voip->len = read(uart, voip->buf, SEQ_BUF_SIZE);
        if (voip->len > 0)
        {
            /* Write to the UDP device */
            write(udp, voip, sizeof(struct voipPkt));
            voip->seq++;
        }
        resched();
    }
}
```

The code for `seq_receive()` is a bit more complex. This code is best at detecting and dropping duplicates, and it does a fairly good job of replaying the last

packet of data when no new data is available. In the future, the code could be used to do something more complex when a deadline is missed. It is important to note that for these new threads, the UDP device is put in a non-blocking mode so that the receiving thread is able to monitor the amount of time that has passed. These code segments are basically a simplified version of RTP. This code could be further extended or possibly replaced with RTP itself.

```
thread seq_receive(ushort uart, ushort udp)
{
    /* Variable declarations and initializations */

    while (TRUE)
    {
        /* Read from the UDP device */
        len = read(udp, voip, sizeof(struct voipPkt));
        if (len > 0)
        {
            if (seq < voip->seq)
            {
                /* Resynchronize the sequence numbers */
                /* (We lost at least one packet.) */
                seq = voip->seq;
            }
            else if (voip->seq < seq)
            {
                /* Drop duplicate packets */
                continue;
            }
            seq++;
            ticks = clkticks;
            time = clktime;
            /* Write to the serial device */
            write(uart, voip->buf, voip->len);
        }
        else if (timespan(clktime, clkticks, time, ticks)
           >
                 (SEQ_BUF_SIZE / 8))
        {
            ticks = clkticks;
```

```
                time = clktime;
                /* Replay the last packet if we missed a
                   deadline */
                write(uart, voip->buf, voip->len);
            }
        }
}
```

5.6 Embedded IP Telephone Cost

Name	Quantity	Cost	Total
Linksys WRT54GL	1	$54.95[1]	$54.95
Dragon12 Development Board	1	$149.00[2]	$149.00
Total:			**$203.95**

Table 5.3: Costs of components to build experimental prototype system

Table 5.3 lists the basic costs for the prototype system that includes the router and the 68HC12 development board. The prototype system is rather expensive at just over $200. Our system uses the router as it is, but it only uses a few of the components on the 68HC12 development board. Table 5.4 outlines some of the items and costs necessary to build a basic embedded VoIP microcontroller board to connect to the router. This table includes an estimate for power considerations and does not include assembly labor costs. The final cost of a simple embedded IP telephone should be around $100. This can be compared to a fairly basic Cisco IP telephone, which costs around $150[3]. It may be possible to further

[1] Price acquired from `http://www.newegg.com/` on 12/2/2009.
[2] Price acquired from `http://www.evbplus.com/` on 12/2/2009.
[3] Price acquired from `http://www.newegg.com/` for Cisco Small Business SPA504G 4 Line IP Phone With Display, PoE and PC Port - Retail on 4/9/2010.

reduce the cost of the hardware needed to attach to the router by using the GPIO (general purpose I/O) pins on the router for connection instead of the serial port.

Name	**Quantity**	**Cost**	**Total**
68HC12	1	$16.91[4]	$16.91
LM386 Amp	1	$0.65	$0.65
LTC1661 DAC	1	$1.75	$1.75
Female Serial Connector	2	$0.75	$1.50
Serial Transceiver	1	$2.15	$2.15
680 ohm resistor	2	$0.02	$0.04
390 ohm resistor	1	$0.02	$0.02
1.5 kohm resistor	1	$0.02	$0.02
10 ohm resistor	2	$0.02	$0.04
220 uF capacitor	1	$0.10	$0.10
0.1 uF capacitor	3	$0.08	$0.24
0.22 uF capacitor	2	$0.15	$0.30
Power considerations			~$20.00
Linksys WRT54GL	1	$54.95	$54.95
Total:			**~$98.67**

Table 5.4: Costs to build a basic embedded VoIP device

5.7 Summary of Performance Analysis

Qualitatively, our IP telephone works. It passes the “ear test,” and two individuals can carry on a conversation using it. However, there is much work to be done to make the IP telephone better. The latency tests show reasonable results with roundtrip delays of just over 2 milliseconds when connected on an intranet network (compared with just under 1 millisecond for a desktop VoIP system). The error analysis shows the different sources of error have been mitigated or deemed acceptable.

[4] Prices acquired from `http://www.digikey.com/` and `http://www.jameco.com/` on 11/12/2009.

CHAPTER 6

Discussion and Summary

Chapter 4 described the embedded IP telephone implementation, and Chapter 5 described how well it worked. This chapter discusses some of the problems that were encountered, some conclusions about the work, and directions for future work.

6.1 Discussion

This section provides a discussion of some of the problems that were encountered during the implementation stage of the research. It is a little reminder that what looks good on paper or on a computer screen does not necessarily work once it has been implemented in hardware.

6.1.1 A Previous Attempt

Before the 68HC12 microcontroller was selected, the same process was tried on a Cypress CY8C29466 PSoC (Programmable System on a Chip) microcontroller [41]. The PSoC chip is designed so that the blocks on the chip can be reconfigured

dynamically for different functions. The chip is attractive because it was inexpensive and contained blocks for 14-bit ADCs, 9-bit DACs, gain amplifiers, filters, timers, and serial ports. The flexibility of the PSoC design would have meant that the majority of the configuration could have been done on-chip instead of as additional circuitry on a breadboard. The processor runs at a maximum of 24 MHz, which should have been sufficiently fast. However, when the ADC, the DAC, the necessary timers, and the serial port were configured, the overall processing speed was not fast enough to sample from the ADC at 8 kHz, write the data to the serial port, receive serial data, and write it to the DAC also at 8 kHz. This limitation did not become apparent until the PSoC had been configured and programmed.

Therefore, while the flexibility of the PSoC chip would have been very useful, the external setup of the ADC and DAC on the 25 MHz 68HC12 allows the chip to be fast enough to handle the signal processing and serial transmission. In addition to simply not being fast enough, the PSoC chip also can only be programmed using the PSoC Programmer and PSoC Designer, which require Microsoft Windows XP. These requirements do not fit well with the rest of the development environment.

6.1.2 From Synchronous to Asynchronous

The following code is an example of what we thought the finished software might look like for the 68HC12 microcontroller. It turns out that there is a problem with every single line of that code, and those problems became evident only because we chose to implement this on a real embedded system.

```
int main()
{
    /* Variable declarations and initializations */

    while (1)
    {
        value = adcRead();
        putchar(value);
        value = getchar();
        dacWrite(value);
        for (i = 0; i < DELAY; i++);
    }
}
```

The call to `adcRead()` initiates an ADC conversion, waits for it to complete, and then returns the value. The call to `putchar()` waits until the previous transmit is complete, transmits the character, and waits until it completes. The call to `getchar()` waits until a character is available and then reads it. The call to `dacWrite()` waits until the transmitter is free, sends the data, and waits for it to complete. The `for` loop delays the `while` loop the appropriate amount of time to achieve 8 kilohertz sampling.

There are two cases that cause this code to fail completely:

- No characters arrive across the serial port, and `getchar()` blocks. No samples are read from `adcRead()` or transmitted with `putchar()`.
- Too many characters arrive across the serial port, and `getchar()` cannot be called fast enough to read all of them in and pass them to `dacWrite()`.

A better design uses interrupts and asynchronous calls to these functions. To start, the `for` loop delay can be replaced by a timer interrupt that occurs every 125 microseconds (8 kilohertz sampling). This reduces the fluctuation (jitter) in the sampling and playback. The biggest problem is still the blocking call to `getchar()`. This can be resolved by doing something like returning `-1` when no value is available and then not calling `dacWrite()`. That code looks something like the following.

```
int main()
{
    /* Variable declarations and initializations */

    while (1); /* Do nothing */
}

void timerInterrupt()
{
    value = adcRead();
    putchar(value);
    value = getchar();
    if (value > -1) dacWrite(value);
}
```

Still, the serial port is overrun because the characters can't be read fast enough. One solution would be to slow down the transmission of characters in the buffer in Xinu on the router. Something similar to the following will work.

```
for (i = 0; i < len; i++)
{
    putchar(buf[i]);
    for (j = 0; j < DELAY; j++); /* Busy wait */
}
```

This is an unacceptable solution because it means there is almost no additional processing time on the router, and delivery is less guaranteed because of

the busy wait that serves as a delay. Ideally, we would take advantage of serial interrupts to resolve this, but since they are unreliable due to a bug in the 68HC12 interrupt structure, we can take advantage of the empty while loop in `main()`. Additionally, ADC interrupts can be turned on so that instead of having to initiate the conversion, wait for it to complete, and then return a value, `adcRead()` can simply return the value. This looks something like the following.

```
int main()
{
    /* Variable declarations and initializations */

    while (1)
    {
        value = getchar();
        if (value > -1)
        {
            buf[r] = value;
            r++;
        }
    }
}

void timerInterrupt()
{
    value = adcRead();
    putchar(value);
    if (w < r) /* the buffer isn't empty */
    {
        dacWrite(buf[w]);
        w++;
    }
}

void adcInterrupt()
{
    /* Clear the interrupt */
    /* Start a new conversion */
}
```

This code greatly decreases the chance of overrun and can handle receiving data from the router, which is transmitting at the full 115.2 kilobaud.

6.2 Future Work

Currently, all connections are configured statically (ports and IP addresses) and initiated separately at each end. This method provides no way to signal another router or VoIP device to initiate a call. This works well in a laboratory environment but is less useful for anyone who actually wishes to place a VoIP call. Both Session Initiated Protocol (SIP) and H.323 provide specifications for call signalling and call setup [6] [8]. In the future, one of these protocols should be selected and written to run on the Xinu operating system and function with the microcontrollers attached to the Linksys routers.

UDP does not have any provisions for parity checking, error checking, and sequence numbering. Therefore, future research and work is necessary in ensuring that the data delivered is correct and that it is delivered in order. This could potentially be achieved through a custom means or through an existing protocol choice. UDP also does not have any provisions for encryption and security. Currently, nothing would prevent a malicious user from injecting packets into the voice stream or eavesdropping on a conversation. Therefore, more research must be done in securing VoIP calls and the initial call connections.

On the hardware side, one limitation on the Systems Lab infrastructure that

this embedded VoIP research has is the dedicated use of the second serial port on all routers being used for VoIP. The Systems Lab is equipped with a serial annex which allows the lab to accommodate all of the serial ports on the fronts of the Linksys routers. These serial ports can also be used to connect to the 68HC12 microcontrollers, and rather than plugging them directly into a router, they can be plugged into the serial annex. Some research is required to create a software serial bridge that would effectively bridge two serial ports on the serial annex together causing it to look like a microcontroller was connected directly to the second serial port of a Linksys router.

6.3 Summary

Quite a bit of research has been done in VoIP, but very little has been done with embedded systems outside of mobile phones. This research provides a solution for VoIP implemented on hardware with no simulation. Since it is *voice* over IP, it only makes sense that it should work with an actual microphone. Previous research on embedded platforms for VoIP have been proprietary and used custom built chips. Our implementation uses resource-constrained commodity embedded hardware and is open and fairly inexpensive.

The VoIP software running on the router is built entirely on top of the lightweight embedded Xinu operating system and runs on the Linksys WRT54GL. The code for the 68HC12 microcontroller is written entirely in C and compiled with GCC. The implementation process showed interrupts and asynchronous calls to be

necessary to make the transmission of voice across both devices possible. The open nature of the system makes it possible to extend and change it in many different ways, such as those discussed in the previous section. Only time was a limit to the implementation of many of the standard VoIP protocols.

The IP telephone code is available as part of the Xinu repository and releases. The software runs on Linksys WRT54GL routers and 68HC12 microcontrollers.

REFERENCES

[1] James K. Peckol, *Embedded Systems: A Contemporary Design Tool*, John Wiley and Sons, Inc., New Jersey, 2008.

[2] Tammy Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, Burlington, MA, 2005.

[3] Richard Swale, Ed., *Voice over IP: Systems and Solutions*, BT Exact Communications Technology Series 3. The Institution of Electrical Engineers, London, 2001.

[4] Uyless Black, *Voice Over IP*, Prentice Hall, New Jersey, 2000.

[5] Jonathan Davidson, James Peters, and Brian Grace, *Voice over IP Fundamentals*, Cisco Press, Indianapolis, 2000.

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

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

[8] International Telecommunication Union, "ITU-T recommendation H.323: Packet-based multimedia communications systems", June 2006.

[9] Simon Haykin, *Communications Systems*, John Wiley and Sons, Inc., New York, 3rd edition, 1994.

[10] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc., 2nd edition, 1999.

[11] International Telecommunication Union, "ITU-T recommendation G.711: Pulse code modulation (PCM) of voice frequencies", Nov. 1988.

[12] Dennis Brylow, "An experimental laboratory environment for teaching embedded operating systems", in *Proceedings for SIGCSE 2008: Technical Symposium on Computer Science Education*, New York, NY, USA, 2008, pp. 192–196, ACM.

[13] Dennis Brylow and Bina Ramamurthy, "Nexos: A next generation embedded systems laboratory", *SIGBED Review*, vol. 6, no. 1, pp. 1–10, 2009.

[14] Dennis Brylow, "Embedded Xinu", URL `http://www.mscs.mu.edu/~brylow/xinu/`.

[15] Wei Wang, Soung Chang Liew, and Victor O. K. Li, "Solutions to performance problems in VoIP over a 802.11 wireless LAN", *IEEE Transactions on Vehicular Technology*, vol. 54, no. 1, pp. 366–384, Jan 2005.

[16] Swapna S. Gokhale and Jijun Lu, "Signaling performance of SIP based VoIP: A measurement-based approach", in *Proceedings of GLOBECOM 2005: Global Telecommunications Conference*. IEEE, Dec 2005, pp. 761–765.

[17] Won-Tae Kim, "A mobile multimedia communication framework for IP based convergence networks", in *Proceedings for ISCE 2006: IEEE Tenth International Symposium on Consumer Electronics*. IEEE, 2006, pp. 1–5.

[18] Ahmad Belhoul, Y. Ahmet Sekercioglu, and Nallasamy Mani, "A mechanism for enhancing VoIP performance over wireless networks using embedded mobility-specific information in RSVP objects", in *Proceedings for GLOBECOM 2006: Global Telecommunications Conference*. IEEE, Nov–Dec 2006, pp. 1–5.

[19] Hussein Ali, Mohammad Inayatullah, and Shmuel Rotenstreich, "Resource allocation and QoS in mobile ad hoc networks", in *Proceedings for ISICT 2004: International Symposium on Information and Communication Technologies*. 2004, pp. 56–62, Trinity College Dublin.

[20] Chih-Wei Huang, Somsak Sukittanon, James A. Ritcey, Aik Chindapol, and Jenq-Neng Hwang, "An embedded packet train and adaptive FEC scheme for VoIP over wired/wireless IP networks", in *Proceedings of ICASSP 2006: International Conference on Acoustics, Speech and Signal Processing*. IEEE, May 2006, pp. 429–432.

[21] David Butcher, Xiangyang Li, and Jinhua Guo, "Security challenge and defense in VoIP infrastructures", *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*, vol. 37, no. 6, pp. 1152–1162, Nov 2007.

[22] Joongman Kim, Seokung Yoon, Yoojae Won, and Jaeil Lee, "VoIP secure communication protocol satisfying backward compatibility", in *Proceedings for ICSNC 2007: Second International Conference on Systems and Networks Communications*, Aug 2007.

[23] Chian C. Ho, Tzi-Chiang Tang, Chin-Ho Lee, Chih-Ming Chen, Hsin-Yang Tu, Chin-Sung Wu, Chao-Hsi Chang, and Chin-Meng Huang, "H.323 VoIP telephone implementation embedding a low-power SOC processor", in *Proceedings for EDSSC 2003: IEEE Conference on Electron Devices and Solid-State Circuits*. IEEE, Dec 2003, pp. 163–166.

[24] Roy Chaoming Hsu, Cheng-Ting Liu, Wen-Ping Huang, and Jun-Jay Yang, "An embedded software approach for the development of SIP-based VoIP

server", in *Proceedings for APSEC 2004: 11th Asia-Pacific Software Engineering Conference*, Nov–Dec 2004, pp. 688–694.

[25] Freescale Semiconductor, Inc., *MC9S12DP256B Device User Guide*, Jan 2005, V02.15.

[26] Freescale Semiconductor, Inc., *ATD_10B8C Block User Guide*, June 2005, V02.12.

[27] Wytec Company, "DRAGON12 MC9S12DP256 development board: Getting started manual", Version 1.46 For REV. E board.

[28] Linear Technology Corporation, *LTC1661: Micropower Dual 10-Bit DAC in MSOP*, 1999.

[29] Free Software Foundation, Inc., "GNU development chain for 68HC11&68HC12", 2003, URL `http://www.gnu.org/software/m68hc11/`.

[30] Motorola, Inc., *M68000 Family Programmer's Reference Manual*, 1992, REV.1.

[31] Freescale Semiconductor, Inc., *HCS12 Serial Communications Interface (SCI) Block Guide*, July 2002, V03.02.

[32] Gordon Doughman, *SCI Interrupt Errata Workaround for HCS12 Family Devices*, Freescale Semiconductor, Inc., Feb. 2003, Rev. 1.

[33] Apple, Inc., "PlainTalk microphone: Specifications", Dec 2000, URL `http://docs.info.apple.com/article.html?artnum=15884`.

[34] Apple, Inc., "Power Macintosh G3 (blue and white): Sound capabilities", July 2004, URL `http://support.apple.com/kb/TA25805`.

[35] National Semiconductor Corporation, *LM386: Low Voltage Audio Power Amplifier*, August 2000.

[36] Jaroslav Kysela and James Tappin, "Arecord and aplay ALSA utilities, v1.0.21", URL `http://www.alsa-project.org/`.

[37] Giovanni Giacobbi, "The GNU Netcat Project", URL `http://netcat.sourceforge.net/`.

[38] International Telecommunication Union, "ITU-T recommendation P.862: Perceptual evaluation of speech quality (PESQ): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs", February 2001.

[39] Kevin Wallace, *Authorized Self-Study Guide Cisco Voice over IP (CVOICE)*, Cisco Press, Indianapolis, 3rd edition, 2008.

[40] Batu Sat and Benjamin W. Wah, "Evaluation of conversational voice communication quality of the Skype, Google-Talk, Windows Live, and Yahoo Messenger VoIP systems", *MMSP 2007: IEEE 9th Workshop on Multimedia Signal Processing*, pp. 135–138, October 2007.

[41] Cypress Semiconductor Corporation, *CY8C29466, CY8C29566, CY8C29666, CY8C29866: PSoC Programmable System-on-Chip*, April 2009.

APPENDIX A

68HC12 Makefile

ada_test/makefile

```
all:   ada.load

# Include the 68HC12 Makefile
include ../compile/Makevars
include ../compile/Makerules
```

compile/Makevars

```
INCLUDE = ../include

HC12_PREFIX = m6811-elf-

CC = ${HC12_PREFIX}gcc
AR = ${HC12_PREFIX}ar
AS = ${HC12_PREFIX}as
LD = ${HC12_PREFIX}ld
OC = ${HC12_PREFIX}objcopy

# C compilation flags
CFLAGS = -m68hc12 -mshort -Wall -Wmissing-prototypes -g -
   O0 -fomit-frame-pointer -msoft-reg-count=1 -c -I${
   INCLUDE}

# Assembler flags
ASFLAGS = -m68hc12 --gstabs -al

# Loader flags
LDFLAGS = -m m68hc12elfb --defsym _start=0 --defsym io_reg
   =0x0 -defsym interrupt_vectors=0xffd6 --no-gc-sections
   -mrelax

# Objcopy flags
OCFLAGS = --output-target=srec
```

include/Makerules

```
clean:
	rm -f *.o *.s19 *.bin *.boo *.lst *.load

# This assembly rule produces a listing file:
%.o: %.s
	$(AS) $(ASFLAGS) -o $@ $< > $*.lst

# Put the text section in one s19 file.
%.s19 : %.elf
	$(OC) $(OCFLAGS) $*.elf $*.s19

# Put the data section in another s19 file and adjust its
# address to 0x3000.
%-data.s19 : %.elf
	$(OC) $(OCFLAGS) --only-section=.data --change-section
	   -address .data=0x3000 $*.elf $*-data.s19

# The load file can be transferred directly over the
# serial port to the 68HC12. It will load both the text
# and data segments and begin executing the program.
%.load : %.s19 %-data.s19
	echo -e "LOAD\r\n" > $*.load
	cat $*.s19 >> $*.load
	echo -e "\r\n" >> $*.load
	echo -e "LOAD\r\n" >> $*.load
	cat $*-data.s19 >> $*.load
	echo -e "\r\n" >> $*.load
	grep text memory.x | awk '{print $$6}' | sed 's/0x//;
	   s/,//; s/^/g /' >> $*.load
	echo -e "\r\n" >> $*.load

# Generate an elf file instead of an s19 file here because
# it lets us split them into a text and data segnments for
# the s19 files later.
%.elf: %.o memory.x ../lib/hc12.a
	$(CC) -m68hc12 -mshort -Wl,-m,m68hc12elfb -mrelax -o
	   $@ $< ../lib/hc12.a

%.s: %.c
	$(CC) $(CFLAGS) -S $*.c -o $*.s

../lib/hc12.a :
	make -C ../lib/hc12 install
```

APPENDIX B

Serial Receive Rate

The following is output from a Linksys router running Xinu. It displays the number of bytes per second that are being received from a 68HC12 microcontroller connected to the second serial port of the router.

```
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
```

```
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8000 bytes/sec
Transmit: 0 bytes/sec, Receive: 8032 bytes/sec
```