A Software-Based Ultrasound System for Medical Diagnosis

by

Samir Ram Thadani

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author .....
Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by .....
David Tennenhouse
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

A Software-Based Ultrasound System for Medical Diagnosis

by

Samir Ram Thadani

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a software-based ultrasound system suitable for medical diagnosis. The device's core functions, which include generating the transmit signal and processing and displaying the received echoes, are implemented in the VuSystem programming environment. In addition, a simulation environment was developed using a software model of the ultrasound transducer and target.

The approach is an improvement over present, hardware-based ultrasound systems because it takes advantage of the flexibility of software. Various, user-defined transmit waveforms can be used, and the receive-side processing can be customized, in real time, to the user's specifications. Furthermore, additional functionalities can easily be added to the prototype system.

Thesis Supervisor: David Tennenhouse
Title: Principal Research Scientist

Acknowledgments

I would like to thank my advisor, David Tennenhouse, for his encouragement and support. I would also like to thank my colleagues in the Software Devices and Systems group at the MIT Laboratory for Computer Science for all of their help. In particular, I would like to thank Vanu Bose for guiding me through this thesis, sharing his insights, and answering my numerous questions. I thank my friends for always encouraging me and helping me through all those late nights. Finally, and most importantly, I thank my family, whose love and support have allowed me to realize my aspirations, and whose continued guidance will no doubt shape my future accomplishments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last half century much progress has been made in medical device technology. One particular medical technology that has improved rapidly over the past 30 years is ultrasound. This progress in technology, however, has brought with it the rapid obsolescence of system designs. As a result, hospitals are often faced with the need to purchase new hardware in order to keep pace with the technology. With every new upgrade comes an increase in cost and a decrease in productivity while hospital personnel become acquainted with the nuances of each new system.

This thesis offers a solution to this technology crisis, in the form of a software-based ultrasound system. This design is compatible with existing ultrasound transducers and could be integrated with existing ultrasound post-processing software. Furthermore, its reliance on software processing makes it far more flexible, and cost-effective, than current systems. The software components of the proposed design have been implemented and their organization and performance is discussed in this report.

1.1 Motivation

Traditional medical devices, such as the one illustrated in Figure 1-1, were based on special-purpose hardware. Although these dedicated hardware systems have been the standard in ultrasound, and in other medical technology, they don't offer much in the way of flexibility or ease of upgrading. With the development of low-cost personal computer technology, much effort has gone into PC-based systems that post-process ultrasound and other medical images. However, such systems still lack some flexibility because they either rely on analog

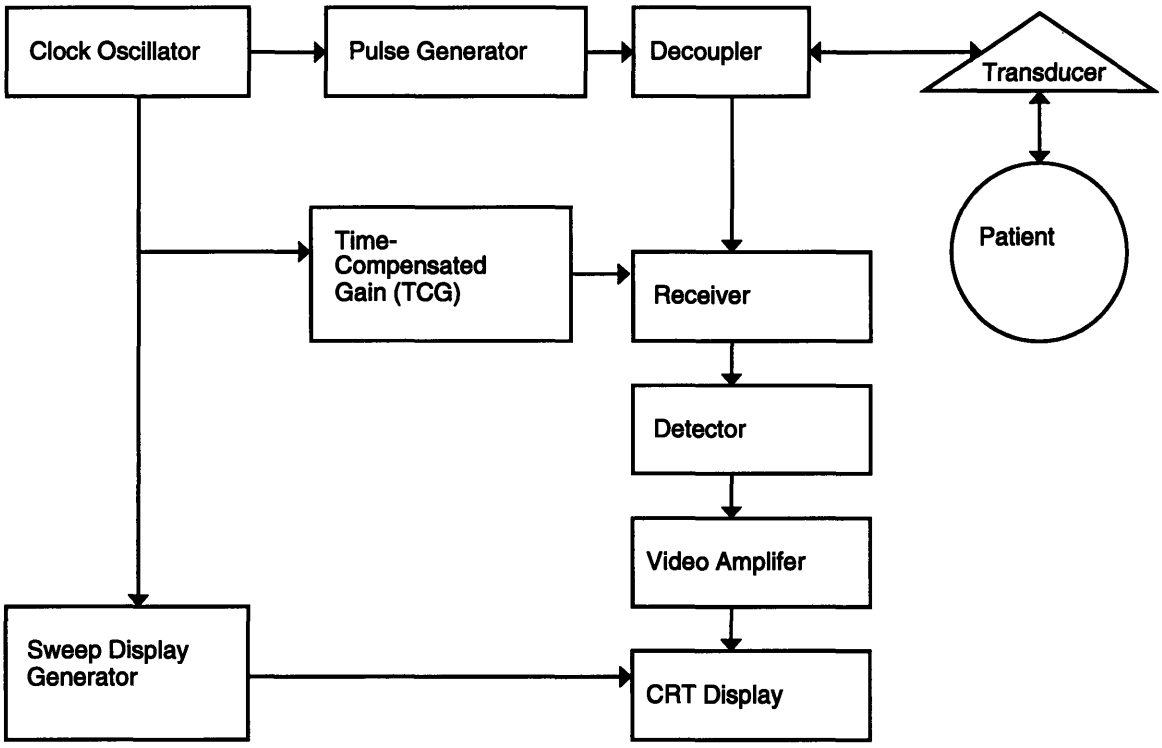


Figure 1-1: A Block Diagram of a Traditional Ultrasound System

hardware to pre-process the signal before A/D conversion or they use customized digital processing [28]. The problem with relying so heavily on hardware is that entire systems need to be replaced each time an advancement is made in ultrasound processing technology. Such changes can be very costly.

Another problem the medical industry faces, particularly with the move towards regional, ambulatory care facilities, is a lack of space and money for the many instruments needed to provide proper care to patients. Often times, a doctor or technician may require several different instruments to diagnose a particular problem. For example, both an EKG and an ultrasound exam may be required when treating a patient suspected to have a cardiac arrhythmia. Small satellite facilities may often elect not to carry certain instruments as a way of cutting costs. What these facilities need are inexpensive, general-purpose instruments that can serve multiple functions. Such instruments would give EMTs more tools to save patients' lives, since the limitations of space often prevent ambulances from having the most sophisticated equipment.

These challenges that the medical community face suggest the need for software-intensive devices that use only a minimum amount of hardware. This thesis describes one such de-

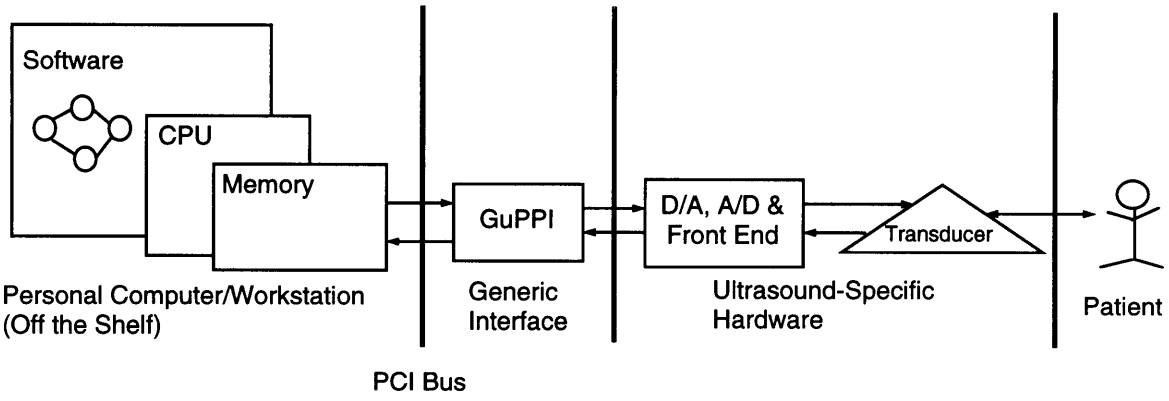


Figure 1-2: The Generalized Layout of the Prototype Ultrasound System

vice: a software-based ultrasound system for use in ophthamological examinations. The system consists of software modules to both transmit and receive ultrasound echoes. These modules, when combined with an ultrasound transducer, a daughter card containing an interface to the transducer, an amplifier, protection circuitry, and an A/D converter, a PCI bus adapter, and a PCI-based host running the Linux operating system, can form a fully functional software-based ultrasound system. All of the system's processing functions, including generating the transmitted signal and filtering and displaying the received signal, are performed in software. All of the software is developed and executed using the VuSystem [15], a programming environment well-suited for multimedia applications. This ultrasound system is particularly advantageous because it consists of off-the-shelf components and can easily be upgraded by modifying the existing software or installing new software. Figure 1-2 shows the layout of such a system.

1.2 Contributions

This thesis aims to demonstrate the viability and power of a software-based medical system. By making use of a simulation environment, it will be shown that ultrasound processing is feasible in software. The system implemented is an ophthalmic ultrasound system, however it can easily be transformed into another type of ultrasound system by running different software and using different transducers. Although the primary aim is to demonstrate the flexibility of such systems, this thesis also shows that problems such as synchronizing the clock signal used in traditional hardware ultrasound systems disappear due to the temporally-decoupled nature of the software environment. As a result, the diagnostic ca-

Figure 1-3: The Display, Control, and Program Windows of the Prototype Ultrasound System

pability of this system may be better than traditional systems, because the user has more control over transmitted pulses and received echoes. Figure 1-3 shows the screen display, user interface, and the interconnection of the software modules developed for the prototype ultrasound system.

1.3 Organization of this Report

Chapter 2 of this report provides some background information on the physics of ultrasound and ultrasound instrumentation and describes previous work in ultrasound technology. The issues and problems involved in developing a software-based ultrasound system are discussed in Chapter 3. Chapter 4 describes the first portion of Figure 1-2, namely the software that was designed for this prototype system. Chapter 5 describes the remaining portions of Figure 1-2: how to integrate the software modules with generic and ultrasound-specific hardware to create an ultrasound unit. Finally, Chapter 6 describes the performance of the software and suggests ideas for future extensions.

Chapter 2

Background

This chapter provides a general background on ultrasound technology, including relevant work in the field. Section 2.1 provides a basic introduction to the physics of ultrasound. Section 2.2 discusses how ultrasound instruments work and provides some information about the various components of these instruments. Section 2.3 discusses Ophthalmic Ultrasound, the particular type of ultrasound for which the instrument discussed in this thesis is designed. Finally, section 2.4 discusses work related to this thesis.

2.1 Ultrasound Basics

Ultrasound waves are mechanical pressure waves that are much like audible sound waves except for their frequencies. Since the frequencies of these waves are much higher than the normal human audible range (20 Hz to 16 KHz), they are known as ultrasound. Ultrasound waves are generated by acoustic transducers upon excitation by an electrical source generating short-duration, high voltage pulses. The rate at which the ultrasound waves travel through a particular medium is known as the acoustic velocity. For the most part, this velocity doesn't change based on the frequency of the ultrasound wave. In the human body the average acoustic velocity is around 1540 m/sec, with most soft tissues having a value within 3% of this average [24].

Ultrasound is useful in medicine because it provides a safe and easy way to image the human body. The reason ultrasound can be used in imaging has to do with the acoustic impedance (Z) of the propagation medium. Acoustic impedance is defined by the following

relation:

$$Z = pc$$

where p is the tissue density (g/cm^3) and c is the acoustic velocity (cm/sec). Acoustic impedance is independent of frequency, and is only dependent on the tissue's mechanical properties, because both tissue density and acoustic velocity don't depend on the frequency of the transmitted wave. When ultrasound waves travel through the body, echoes are produced. These received echoes are the result of sudden changes in acoustic impedance occurring at the boundaries of organs and at other interfaces. They can give information about the structure of the area through which the transmitted wave was being passed.

2.1.1 Echo Generation

There are two types of reflectors that can produce ultrasound echoes: specular and diffuse. Specular reflectors occur at the interface between two different soft tissues in the body. When an ultrasound pulse is incident on this interface, two beams are formed. The first beam corresponds to the transmitted signal which continues to propagate through the second medium. The second beam is reflected off the interface and travels back throughout the first medium. This idea is shown in Figure 2-1 [23].

The direction of the reflected echo is determined by the law of reflection which states that the angle of reflection is equal to the angle of incidence. Often times, a single transducer is used to both generate and receive the ultrasound signal (see Section 2.2). In these cases, the incident beam must be perpendicular to the interface (an angle of incidence and reflection of 0^o) [24].

The strength of the echo depends on the acoustic impedances of the two media. The reflection coefficient(R) measures the fraction of the incident wave that is reflected at the boundary. Its value is determined as follows:

$$R = (Z_2 - Z_1)^2/(Z_2 + Z_1)^2$$

where Z_1 and Z_2 are the acoustic impedances of media 1 and 2, respectively. A lower reflection coefficient means that the two media have similar impedances, and as a result most of the energy is transmitted. A higher reflection coefficient, meaning that a higher proportion of the incident energy is reflected, corresponds to a bigger difference in acoustic

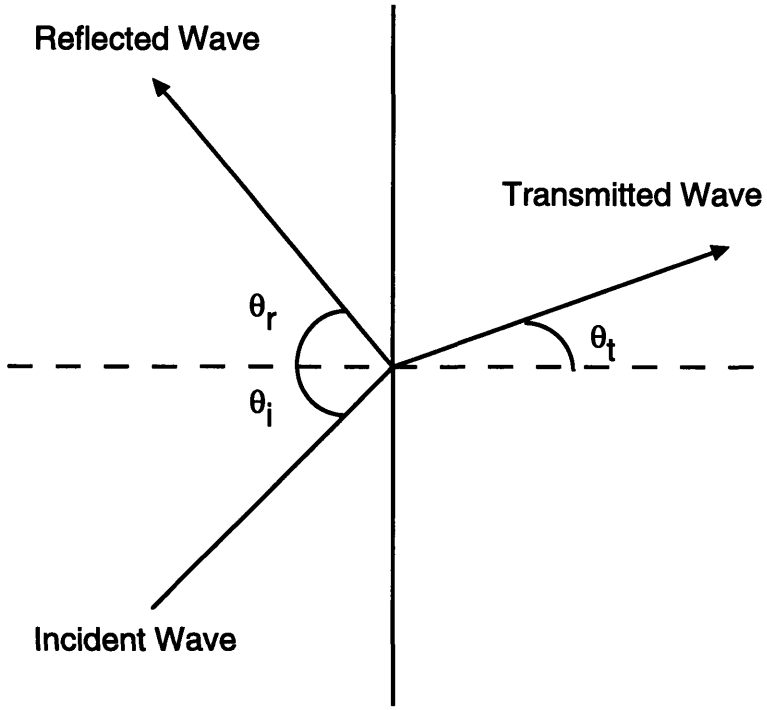


Figure 2-1: Reflection and Transmission of Ultrasound at a Boundary

impedance between the two media[24].

Diffuse reflectors are materials whose dimensions are significantly smaller than the wavelength of the incident beam. As a result, they scatter ultrasound beams in every direction, producing echoes that tend to have lower amplitudes than those produced by specular reflectors. Since most structures don't have completely uniform acoustic impedances, various parts of the structure (each corresponding to a particular acoustic impedance) can act as diffuse reflectors [24].

When the wave travels through the body, not all of the incident energy is either transmitted or reflected. Some of this energy is absorbed by the tissues and converted to thermal or heat energy, leading to attenuation of the signal. Tissue attenuation is usually around -1 dB/cm MHz [24]. However, since the signal must travel through a particular region twice (once when transmitted and once when reflected), the actual attenuation is twice this amount. Thus, a 10 MHz signal is attenuated by 20 dB for every centimeter of tissue depth.

2.2 Ultrasound Instrumentation

Since so many different parts of the body can be imaged using ultrasound technology, many types of ultrasound instruments have been developed. The most commonly used types of ultrasound instruments include cardiac, fetal, and ophthalmic systems. Although each type of system has certain features necessary for that particular modality, they are all fundamentally quite similar. The two major types of ultrasound, A-scan (amplitude) and B-scan (brightness) are discussed below.

2.2.1 A-Scan

The simplest type of ultrasound scan is the A-scan. Such systems are known as Time of Flight (TOF) imaging systems because the time it takes for a signal to return to the transducer is related to the distance the signal traveled [12].

Figure 2-2 shows the basic idea behind A-scan imaging. A pulse is transmitted by the transducer into Medium 1. It then encounters the interface between Medium 1 and Medium 2. Since there is a difference in acoustic impedance between the two media, some of the signal is transmitted into Medium 2, while some of the signal is reflected. The reflected signal travels back through Medium 1 and into the transducer. The distance this signal has traveled is $2d$ meters (corresponding to a round-trip from the transducer to the interface between the two media). In A-scan imaging, one would like to know the distance from the transducer to particular structures of interest so that the relative amplitude of the returning echoes can be plotted versus the distance into the body. Although the value of d is not known, it can easily be calculated as long as the propagation velocity of the wave is known. Thus, if the velocity of the traveling wave is v and the time at which a particular echo returns to the transducer is t seconds, then the distance from the transducer to the structure that generated the echo is $vt/2$ meters [12].

A block diagram of a basic A-scan system is shown in Figure 2-3. The impulse generator is used to establish the pulse repetition frequency (PRF), the rate at which ultrasound pulses are emitted from the transducer (this type of ultrasound is known as pulse-echo ultrasound). These short-duration pulses are then amplified so that they can excite the transducer (at a voltage ranging from 20 V to 300 V depending on the transducer). Before they are displayed, the returning echoes are amplified, filtered, and conditioned. The amplified signals have a

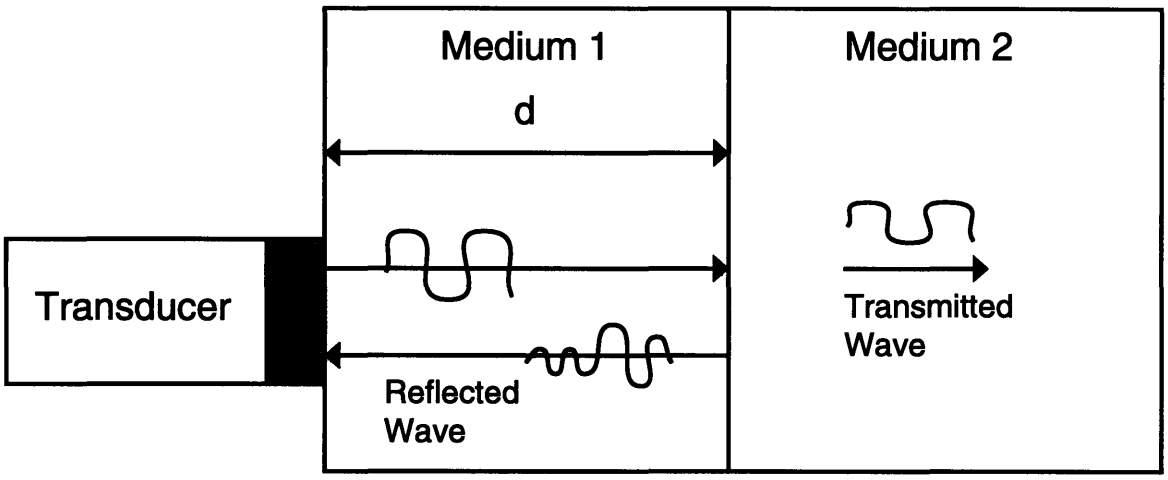


Figure 2-2: Reflection and Transmission in A-scan

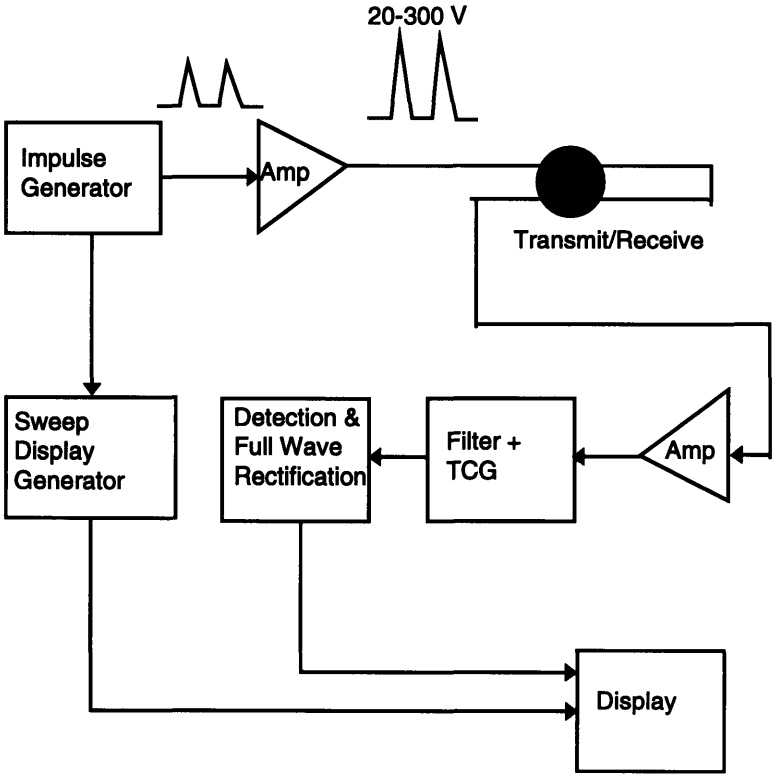


Figure 2-3: Schematic of A-scan

dynamic range of around 100 dB. The signals are then displayed on an oscilloscope with sweeps triggered by the impulse generator (determined by the PRF) [12].

Figure 2-3 depicts a system in which one transducer is used to transmit a pulse and another is used to receive the echo. However, it is possible to use a single transducer for both functions. In this case a decoupler is needed to permit the transducer to be used as both a transmitter and receiver. It acts like a switch by making sure that the transmitted pulse is only sent to the transducer and not to the receiver circuitry and that the received echo is only processed by the receiver circuitry and not sent to the pulse generator.

Pulse Repetition Frequency

Since the PRF determines the rate at which pulses are generated from the transducer, a higher PRF gives better display intensity. However, there is a limit as to how high the PRF can be set. Figure 2-4 shows reflections received from a pulse transmitted at time zero. The echo from the farthest interface is received at time t_2. Thus, if a second pulse were transmitted before this time, the returning echoes from this pulse (particularly from nearby structures) would interfere with the returning echoes from the first pulse (due to faraway structures), as shown in the second part of the figure. In order to have an unambiguous display, it is necessary to wait until all the echoes have been received due a particular pulse before generating another pulse. The maximum PRF is therefore:

$$PRF_{max} = v/2d$$

where v is the velocity of the wave in the medium and d is the furthest reflecting interface.

Time-Compensated Gain

Since there is a significant amount of attenuation as an ultrasound wave travels through a medium, echoes originating from interfaces deep in the medium will have a much smaller amplitude than those closer to the transducer. In order to offset the effects of higher attenuation at increased depths, a time-compensated gain (TCG) unit is used. The TCG, or Swept Gain, amplifies echoes from deeper tissues more than those from tissues near the surface. TCG also helps prevent echoes from an earlier pulse from interfering with the echoes returning from the current pulse since the gain for these earlier echoes is reduced to very

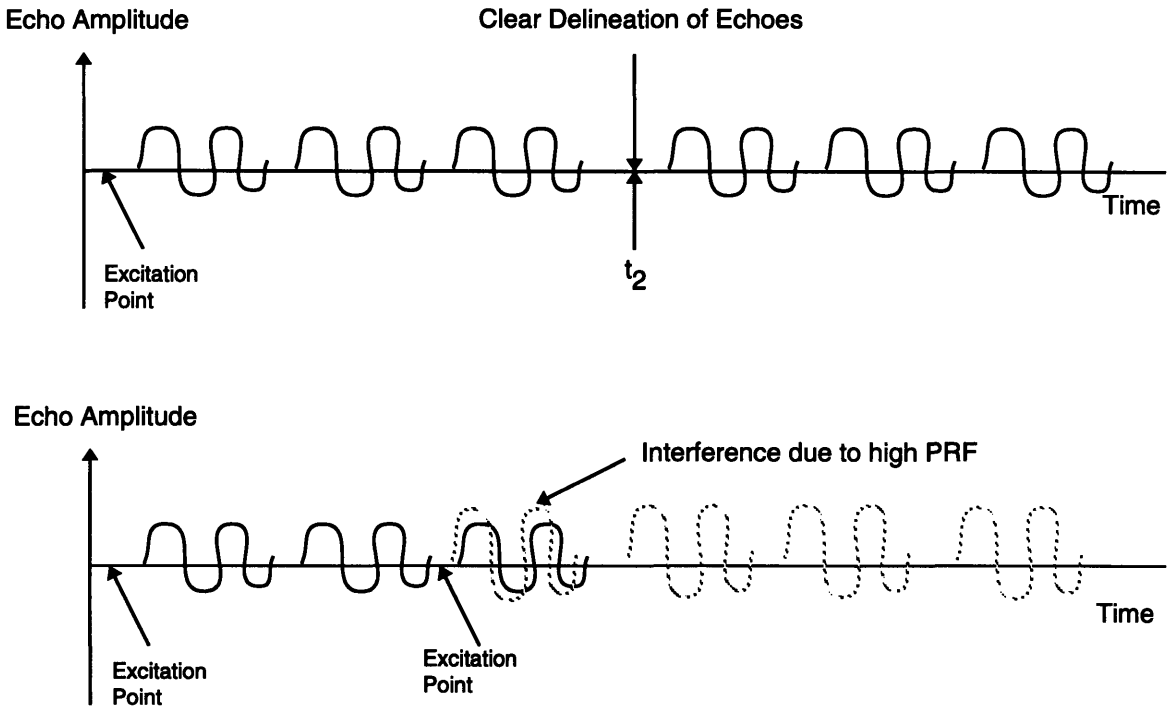


Figure 2-4: Pulse Interference due to high PRF

low levels. The TCG function includes a "dead time" so that no echoes are displayed from regions close to the transducer. The reason for nulling these signals is because they usually correspond to the skin or subcutaneous layers of fat that are generally not very interesting to observe. The dynamic range of the detected signal can be reduced by approximately 50 dB due to TCG [12]. Figure 2-5 illustrates the concept of time-compensated gain for a particular function.

Display

In basic A-scan systems, the echoes are displayed, unprocessed, versus the depth into the body. More sophisticated systems, however, make use of rectification and smoothing to enhance the visualization of the received signal. These techniques are illustrated in Figure 2-6. The smoothing operation is basically an envelope detector, or demodulator, in which noise and any unwanted oscillations are filtered out by detecting only the peaks of the signal.

Imaging problems

Since the velocity of the wave is required in order to compute the tissue depth, often times ultrasound systems make use of certain simplifying assumptions. First, the velocity of all

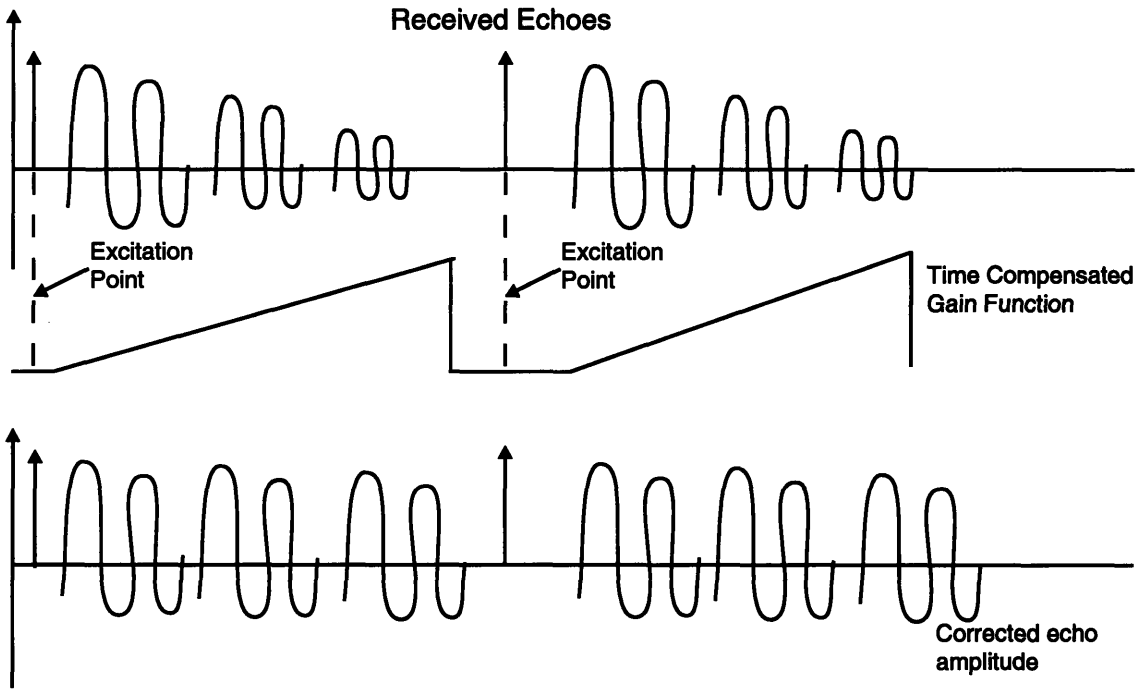


Figure 2-5: Time Compensated Gain

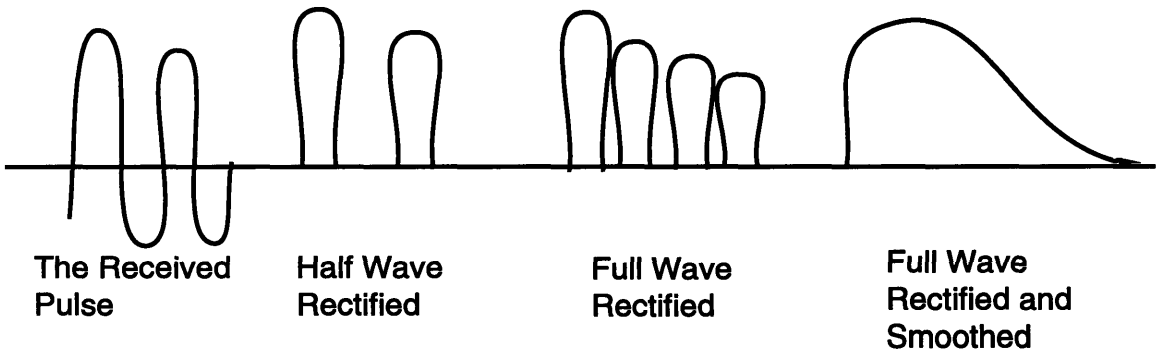


Figure 2-6: Half and full wave rectified echoes

tissue is assumed to be 1540 m/s. This assumption can introduce some error (as much as 10%) into the calculation since ultrasonic velocity is not constant in the body because different tissues have different material properties [10]. Some advanced instruments use a different velocity for particular regions, applying the appropriate velocity based on the time at which the echo was received. However, this method is still somewhat inaccurate. The second assumption is that the pulse is traveling in a straight line in the tissue. Although this assumption is necessary in order to make it easier to calculate the distance to a particular structure, it is often inaccurate since ultrasound may return to the transducer after having been reflected multiple times. Finally, the detected target is assumed to lie along the central ray of the transducer beam pattern. This assumption is necessary to prevent blurring or misregistering the position of the reflector [24]. Figure 2-7 illustrates some of the problems associated when the aforementioned assumptions do not hold.

Axial Resolution

The axial resolution measures how well an ultrasound system differentiates between two interfaces along the same axis, but separated by a distance of d. This factor is influenced by the bandwidth of the transducer, the characteristics of the excitation pulse, and the functionality of the detection circuitry [12]. A highly damped transducer with wide bandwidth will give better resolution than a lightly damped one with lower bandwidth. The excitation pulse, which is often around 100 ns in duration, must also have a wide bandwidth in order to achieve optimal resolution.

Interpretation

Although an A-scan display provides information about the structures along the path of the beam, it can often be very difficult to differentiate between structures that are very close. For this reason, and also because of the complex nature of the human body, it is important to have a skilled professional interpret the results of an ultrasound scan.

Modern Uses of A-scan

Although the A-mode gives positional information quickly using a minimum of advanced technology, the B-scan (described in Section 2.2.2) is most prevalently used in modern ultrasound imaging. However, there are still two common uses of A-scan imaging: scanning

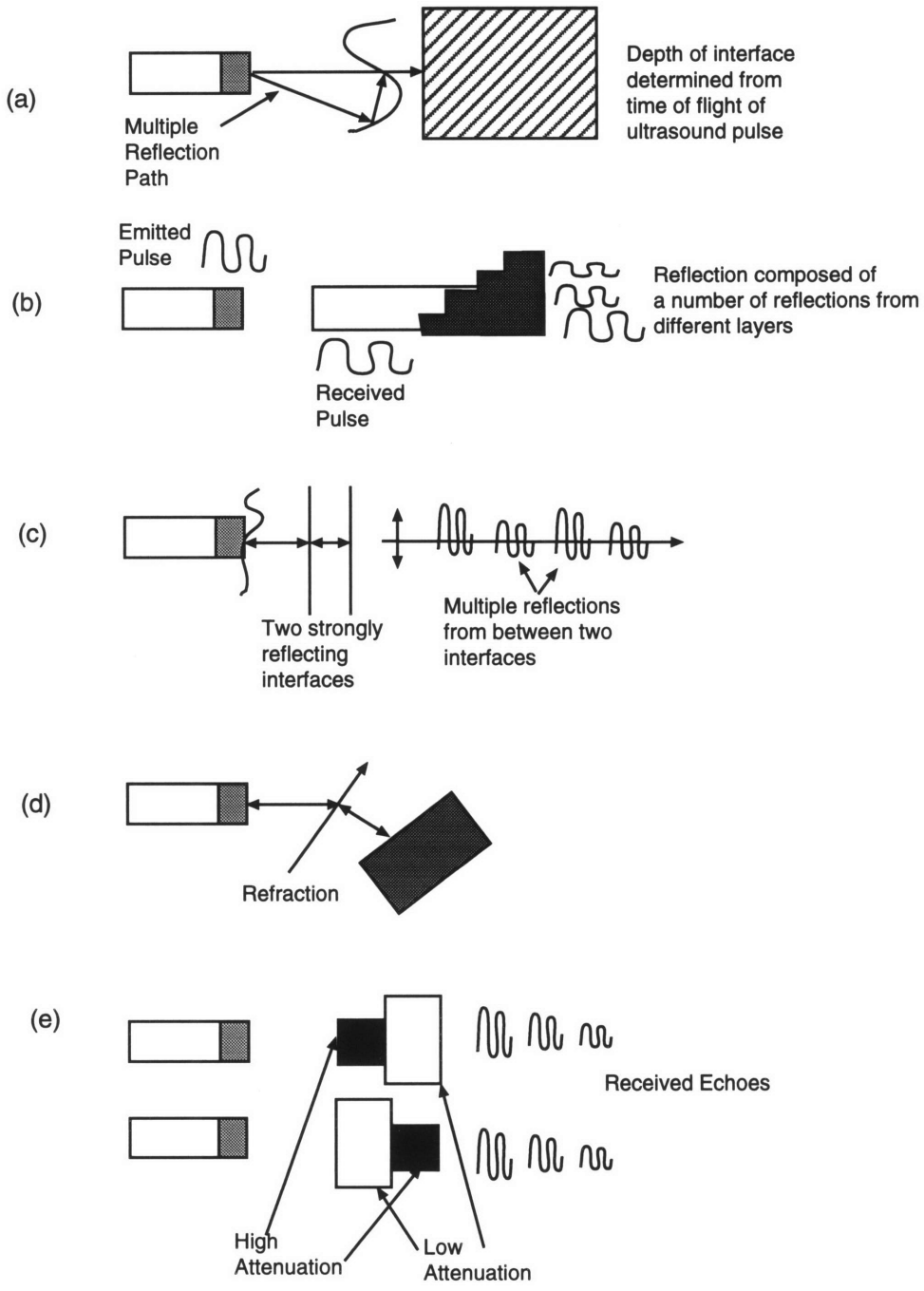


Figure 2-7: Problems in A-scan imaging

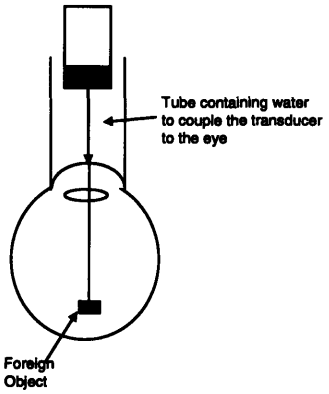


Figure 2-8: Eye scan setup

the eye and the mid line of the brain. Ophthalmic ultrasound makes use of A-scan technology because the eye is a relatively simple structure. Also, the small size of transducers required for A-mode scanning, compared to those required for the B-scan, is advantageous considering the small size of the eye. The A-scan is used primarily to measure the eye size and growth patterns, and detect the presence of tumors and the location of foreign objects within the eye. The setup for such a scan is shown in Figure 2-8. In this setup, a tube containing water is used to couple the transducer to the eye. Since the eye's small size corresponds to a small penetration depth, higher frequencies can be used when scanning the eye in order to generate better resolution.

A-scan imaging is also used to detect a shift in the mid line of the brain, due to internal bleeding within the skull, and in non-medical applications such as detecting cracks in uniform materials and detecting the dimensions of materials.

2.2.2 B-scan

In B-scan imaging systems, an A-scan device is swept across the surface of a patient's body in order to capture a series of images in a pie-shaped plane known as the scan plane. The pie-shaped ultrasound image is formed via a sector scan. The transducer is rotated about an axis (usually around 60^o) to generate A-scan images along various lines of site. Instead of displaying the amplitude along the vertical axis as in A-scans, B-scans consist of lines emanating from a common origin in which the brightness of every position along each line represents the relative amplitude of an interface at the depth given by the distance of that position from the origin. A multiplicity of such lines are spread in an arc to form a cross-

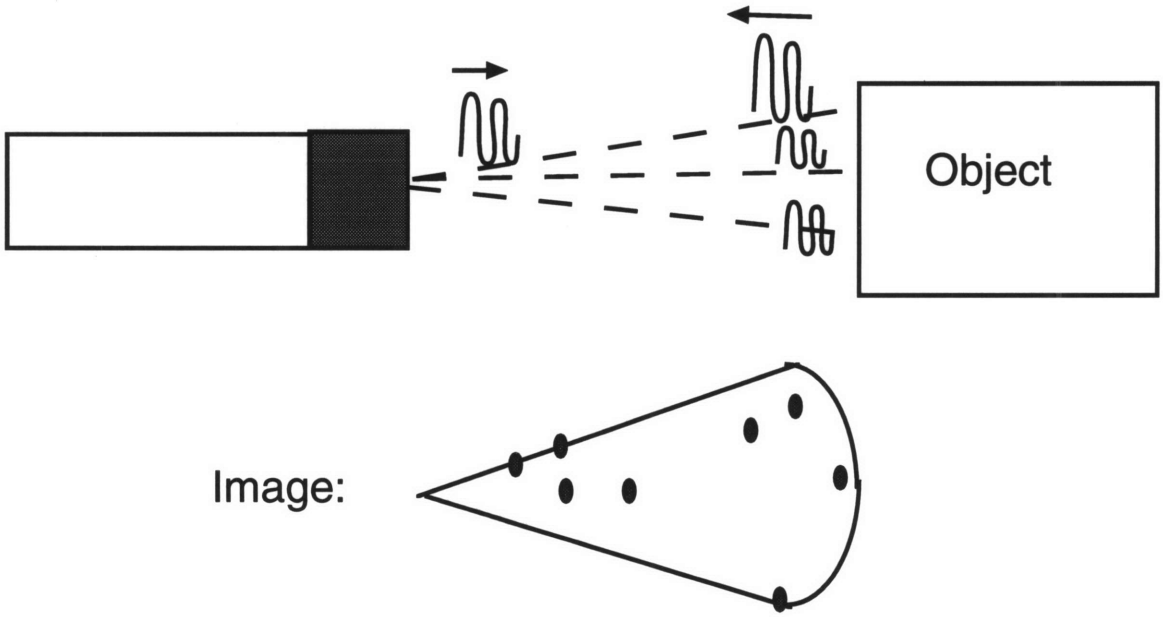


Figure 2-9: The B-mode Scan Plane

sectional picture of the scanned tissue, including such prominent features as organs and bones. This idea is illustrated in Figure 2-9.

Figure 2-10 shows how a B-scan image can be formed from the corresponding A-scan echoes. The resulting echoes along a particular scan line are full-wave rectified and used to determine the brightness of the display along that particular line. This same procedure is repeated for every scan line in a particular scan plane (created by moving the probe to adjacent positions) [12].

A block diagram of a B-mode system is shown in Figure 2-11. This system is very similar to the A-scan system except that it requires angular information from the probe which can be combined with the echo amplitude in order to produce a dot (with the appropriate brightness) at each point in the x-y display. This system also uses a range compression scheme in order to reduce the dynamic range from 50 dB after filtering and TCG, to the 20 dB that can be displayed on the CRT. This compression can be implemented using such nonlinear filters as logarithmic amplifiers [12].

In order to determine how fast a B-scan can be performed, one must know the depth of interest of the area being scanned. The B-scan is much like the A-scan in that one has to wait for echoes to return from the deepest organ of interest before scanning the adjacent line (in the case of the A-scan generating the next pulse). In order to avoid flickering, the

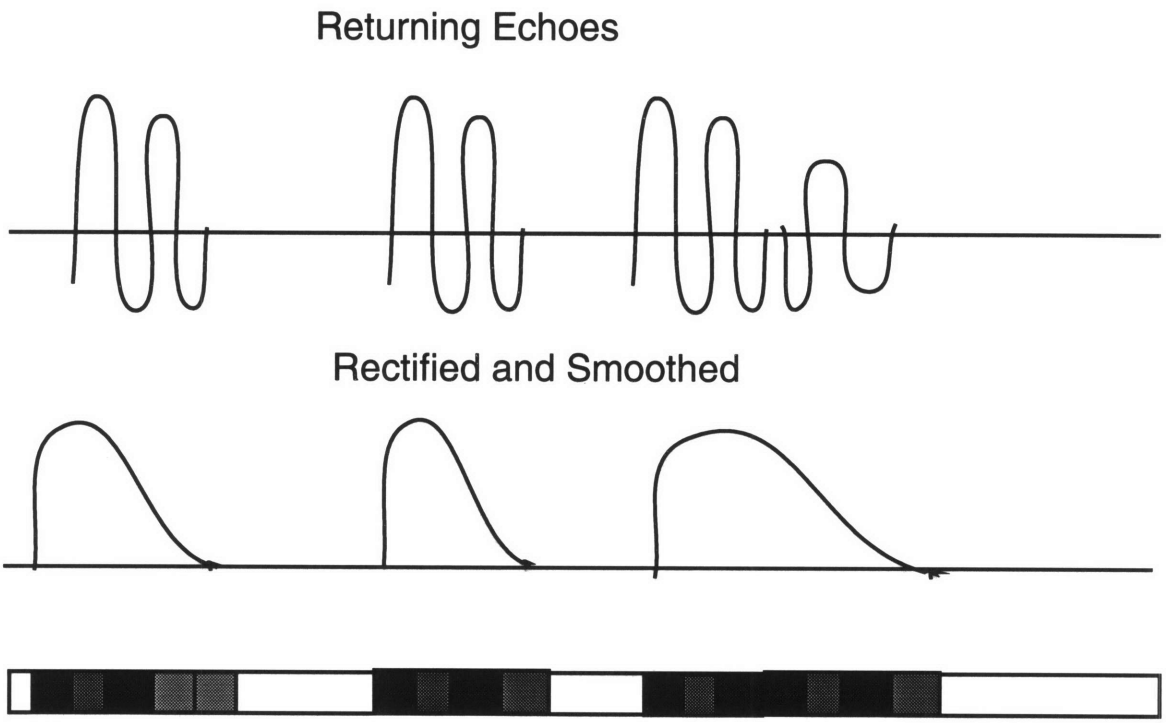


Figure 2-10: B-scan image formation

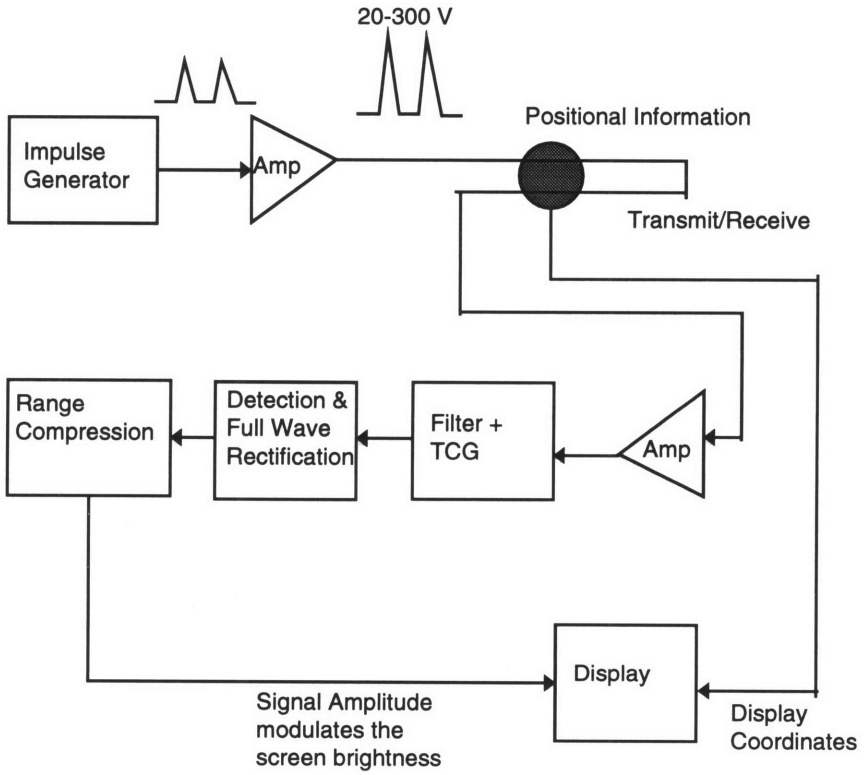


Figure 2-11: B-scan Block Diagram

image must be updated 30 times per second. Since the pulse repetition frequency is fixed for a particular depth and the time allowed to generate an image is also fixed, the following relations determine the number of lines in an image:

$$time\ for\ one\ line\ scan\ = d/v$$

and

$$number\ of\ lines\ in\ a\ scan\ = v/Rxd$$

where d is the depth of the deepest organ of interest, v is the wave velocity in the tissue, and R is the screen refresh rate.

Transducers

There are three major types of transducers used for B-mode imaging: fixed focus transducers, linear array transducers, and phased array transducers. Fixed focus transducers use either a lens or a curved transducer substrate to improve the lateral resolution by creating a focal zone. Linear array transducers consist of many transducers working in concert. Groups of transducers may be excited by a single excitation pulse, and may also receive and process the resulting echoes. This type of arrangement makes it easier to perform a translational scan and a sector scan if a curved substrate is used to form the linear array. Phased array transducers are much like linear array transducers, except that each individual transducer is excited separately in order to shape the outgoing beam in a particular way. This steering capability can also affect the received echoes in order to process them in a certain way.

2.3 Ophthalmic Ultrasound

Ophthalmic ultrasonagraphy dates back to the end of WWII. In 1956, Mundt and Hughes used the A-scan technique to detect intraocular tumors. As instruments have improved, the use of ultrasonagraphy has become more prevalent. A-scan is particularly useful for making various measurements of the eye, such as measuring the axial eye length (biometry) or measuring the width of the optic nerve and extraocular muscles. A-scan technology is particularly sensitive to detecting orbital diseases. The B-scan is used for detecting such

ailments as cataracts and vitreous hemorrhaging [3].

2.4 Related Work

This section presents the work related to developing a software-based ultrasound system. The first portion of this chapter delves into the history of ultrasound systems and discusses the early use of the PC in ultrasound technology. The next part discusses other software-based ultrasound systems, in order to provide a framework for where the prototype system being developed fits in. Finally, other ultrasound-related problems that are currently being addressed will be discussed.

2.4.1 History

In 1883, Galton became aware of ultrasound when he was studying the limits of the acoustic spectrum perceived by humans. In his research, he created one of the first man-made ultrasonic transducers. However, since electronic technology did not experience the advancements we see today, there wasn't much progress in ultrasound technology during the ensuing 30 years. During World War I, scientists were searching for both a way to detect submarines and to communicate underwater. Langevin in France came up with a way to use quartz transducers to send and receive low frequency ultrasonic waves in water. In 1925, Pierce was able to create ultrasound probes with resonant frequencies in the MHz range, using quartz and nickel transducers [23].

Sonar technology, which was used during World War II, inspired researchers to use ultrasound in medicine. After the war, the Japanese were able to build a primitive A-mode ultrasound system that had a trace of the amplitude waveform on an oscilloscope. They then developed a B-mode system by using gray scale imaging on an oscilloscope display. Using these technologies, Japanese scientists were able to detect gallstones, breast masses, and tumors. Shortly thereafter, Doppler ultrasound was developed to detect moving parts [1].

During the 1950's, ultrasound came to the United States. It was during this time that real-time imaging became feasible and the first form of two-dimensional images were possible using hand held scanners. By the early 1980s, ultrasound imaging had advanced significantly, however computers were not used in the process, so images could not be

enhanced [1].

2.4.2 The Use of the Personal Computer

Most of the early work in ultrasound technology focused on developing dedicated hardware systems. However, with the advent of the personal computer, ultrasound manufacturers began using PC technology to enhance their systems. In spite of this advancement, virtually all of these systems still consist of dedicated hardware used to generate pulses and capture the echo information. The PC is used to improve the reviewing process by allowing the user to annotate and post-process this captured data. This post-processing often includes image enhancement (by enhancing the edges). There are systems that can store, send, and receive ultrasound images via computer, however these systems involve digitizing analog ultrasound images off-line [29]. Such processes can easily be incorporated into real-time software-based ultrasound systems.

PC-based Digital Storage and Retrieval (DSR) systems are becoming increasingly popular as medical facilities run out of physical space to store analog ultrasound images. In order to make DSR systems work, or any other type of system in which the post-processing takes place on a PC, it is necessary to digitize the ultrasound image. The components required to digitize the ultrasound image are separate from those required to capture the images. Thus, these systems don't offer the flexibility of software-based systems since extra hardware is required to digitally manipulate images. Similar schemes are also required when trying to send ultrasound images over networks. More recently, Picture Archiving and Communication Systems (PACS) have been used as a way to electronically acquire, store, and display digitized images. Although these systems are revolutionary since they replace the videotape with digital storage media, they really don't involve any intelligent processing of ultrasound images [14].

2.4.3 Software-Based Systems

Until recently, inexpensive ultrasound systems based on the PC were nowhere to be found. However, in November 1995, Perception Inc., a medical equipment manufacturer, and Matrox, a supplier of PC-based imaging technology, announced the development of the first PC-based ultrasound imaging system consisting of off-the-shelf PC hardware and software [18]. The Perception system consists of a Pentium PC, with an imaging sub-system and

Perception's Virtual Console GUI running under Windows NT. The system uses proprietary hardware to convert the signal generated by the probe to S-video, digitize the video signal, transfer the image to the PC's system memory, and provide desktop video display. The GUI, which was developed using Microsoft Visual Basic, is used to control the examination procedure [17].

The Perception system is much like the system described in this report in that it attempts to use mainstream PC technology as a means of reducing the cost and increasing the flexibility of ultrasound imaging. Their Virtual Console is much like our virtual instrument panel since both replace expensive and breakable hardware-based knobs, levers, and switches with "virtual knobs" represented as software-generated icons on the computer's display. However, what differentiates the present effort from the Perception product is that the former allows raw ultrasound data to be processed while requiring a minimal amount of proprietary hardware, while the latter uses proprietary hardware to transform the ultrasound data into video images which are captured and then processed in software. The ultrasound system developed in this thesis is more flexible than the Perception system because the former allows for the possibility of analyzing the raw data and processing it differently based on certain dynamically established trends. Such a processing methodology could allow more efficient use of computational resources, resulting in improved performance, and the development of new approaches to ultrasound signal processing.

Siemens and the Imaging Computing Systems Lab at the University of Washington have developed a high-end ultrasound system that includes a programmable image processor [25]. Thus, this system can be programmed to run many applications, reducing the time it takes to bring new ultrasound applications to market. The goal of this system is to make it unnecessary to reinvent the hardware system every time major advancements occur. The first application written for the processor is called SieScape. It will allow doctors to see a panoramic view of the human anatomy when taking ultrasound images [30]. Although this system has the same goal as the prototype software-based ultrasound system, it is quite different since it involves using high-end components (40 Pentium processors), while the system outlined in this thesis strives to make use of more practical off-the-shelf components.

A system is currently being developed at Northeastern University's Ultrasound Laboratory that allows one to transmit and receive an arbitrary waveform using a computer interface for control and analysis [20]. The generated waveforms are propagated through an ul-

trasound tank with transducers that both transmit and receive. The system is Windows95-based using both C++ and Matlab files for data acquisition and control. The hardware consists of Pentium computer, high-frequency amplifiers, and fast, wide bandwidth, A/D and D/A converters. This system is not viable for commercial applications because of the use of a 55 gallon ultrasound water tank to generate the appropriate waveforms. It is meant to be used as part of a research project in which various parameters will be changed. Since many different types of waveforms will be required, it is convenient to use a software-based approach in which parameters can easily be changed.

2.4.4 Other Issues in Ultrasound

The use of the PC in ultrasound imaging has opened the door to solve other types of problems in ultrasound imaging. PCs are used in conjunction with two-dimensional images in order to produce three-dimensional ultrasound. Basically, the PC is used to keep track of position data, using some type of PC-based position sensing device. This information can be used with the ultrasound images to form a three-dimensional picture by using certain geometrical assumptions [31].

More advanced three-dimensional systems are also available. Such systems can generate three-dimensional images in real-time using off-the-shelf components. One such system, developed by Parsytec, uses a PowerPC system, parallel processing blocks, media coprocessors, and ATM communications protocols. This system uses standard raycasting techniques to form the three-dimensional image from two-dimensional scans [7].

New technologies are also using coherent phase information, in addition to amplitude information, to form more accurate ultrasound images. Such systems use computers to calculate the properties of scan lines in order to determine additional numerical values based on the fact that certain information can be found in different parts of the echo signal. Using phase information, a clinician can better visualize subtle differences in the areas being examined [6].

Chapter 3

Approach

Since the overall aim of this thesis is to demonstrate that a software-based ultrasound system is both feasible and an improvement over current ultrasound systems, it is important to understand some of the challenges involved in developing present-day, hardware-based systems. Section 3.1 discusses some of the major processing issues involved in developing ultrasound systems. Finally, Section 3.2 explains why a software-oriented approach makes it easier to perform both of these tasks and introduces the approach taken in this thesis.

3.1 Traditional Approach

Figure 3-1 depicts a traditional ultrasound system, in which a pulse generator and an amplifier are used to create a high voltage, short-duration spike that excites the transducer. The pulse generator is also used to trigger a time sweep across the CRT, which is used to calibrate the horizontal axis such that the returning echoes are properly displayed. Thus, in order to avoid errors, there must be perfect synchronization between the pulse generator and the display.

When a single transducer is used to both transmit and receive the ultrasound echoes, it is important to protect the receiver circuitry from the high voltage spike necessary to excite the transducer. For this reason, some type of switch is usually used to decouple the transmit side from the receive side. This switch often consists of a pair of parallel, reversed diodes that appear as short circuits for the high voltages associated with transmission and as opens for the low-voltage receive echo waveforms.

On the receiver side, the circuitry must be perfectly designed in order to assure a dis-

play with maximum readability. When the signals are received they first pass through an amplifier. This amplifier must have high gain, low noise, and a frequency response that can handle the wide range of frequencies in the incoming echoes. Such requirements demand very precise equipment.

As mentioned in Section 2.2.1 it is necessary to use time-compensated gain to offset the attenuation of signals deeper into the body. Since tissue attenuation, as stated in Section 2.1.1 is 1 dB/cm MHz, the TCG must be set to offset this attenuation. Assuming the pulse travels at 1540 m/s, the TCG rate should be set at 154 dB/ms for every MHz of the transmitted frequency. This rate is actually a little high due to two reasons. The first reason is that the transmitted pulse consists of a broad spectrum of frequencies as opposed to just one frequency. The frequency associated with the transmitted pulse is actually the center frequency in this spectrum. Higher frequencies are actually attenuated more, due to the dispersive absorption nature of tissue. Thus, most of the pulse's energy is carried by the lower frequencies. These frequencies are not attenuated as much, and as a result the TCG does not need to be as high. The second explanation for why the predicted TCG is high has to do with the fact that some tissues attenuate the signal less than 1 dB/cm MHz. Many clinical instruments arbitrarily partition the penetration depth into several segments, and give the user some control over the TCG within these segments. However, the user cannot arbitrarily change this value to emphasize or deemphasize certain structures [5].

The received signal follows the same basic shape as the transmitted waveform. Since the transmitted pulse has numerous oscillations, the received pulse also has many oscillations. These oscillations are not clinically relevant, so the pulses are often electronically demodulated in order to capture the envelope of the pulse. Usually this demodulation is accomplished by using a diode stage followed by a capacitor. Some instruments let the user control the time constant of the demodulation process in order to influence how the waveform appears on the screen. Although this process may improve the display, it results in loss of information about the phase and exact timing of the received pulses. The advantage of demodulation is that it shifts the frequency spectrum down, by removing the carrier frequency, such that it is centered around zero frequency. This makes it easier to design electronic components for later stages.

Since it is impossible to present the entire dynamic range of the received signal on cathode ray tubes, these signals are often compressed to a smaller range by using a logarithmic

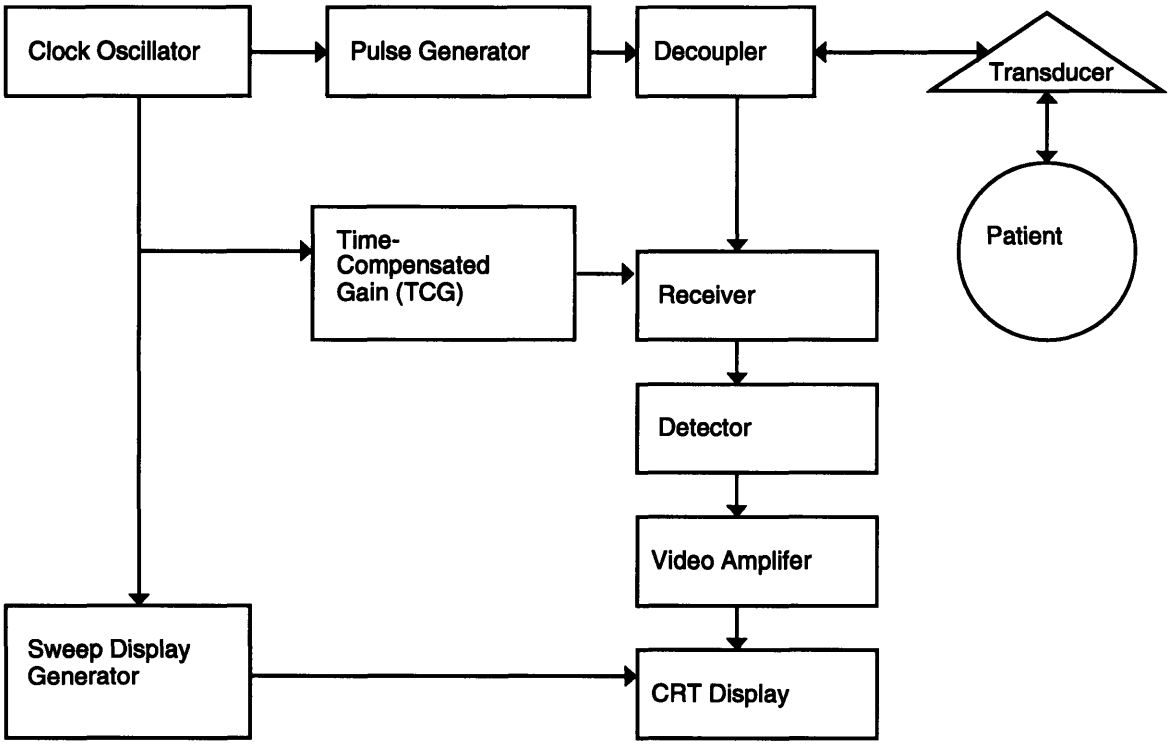


Figure 3-1: Block Diagram of a Traditional Ultrasound System

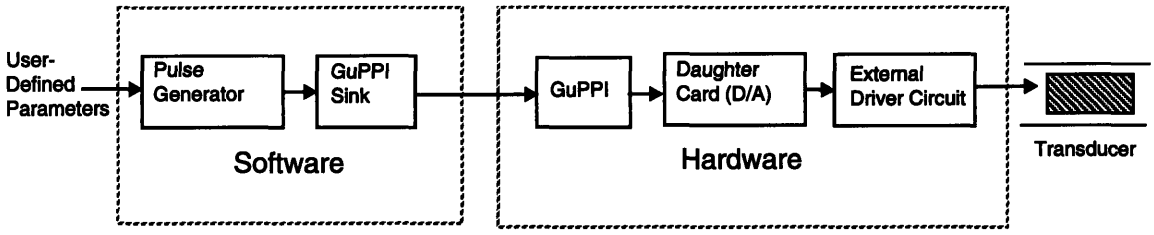
amplifier. Such a procedure allows small echoes to be seen on the same display as larger ones. However, often times noise can be amplified significantly. For this reason, many instruments allow the user to set a threshold which can be used to determine the minimum amplitude of signals displayed on the screen.

3.2 The Software Solution

Figure 3-2 is a block diagram that shows how the software described in this report can be combined with hardware to form a complete software-based prototype system. A software pulse generator fills a portion of memory with a sequence of integer "samples" that correspond to a short duration square wave pulse, at a frequency specified by the user. This pulse is then converted to an analog waveform, via the GuPPI[1] and a D/A converter. The analog signal is externally amplified to 155 V and used to excite the transducer. Once the transducer is excited, echoes are received from the subject being scanned. These received echoes are externally amplified and then converted to a sequence of digital samples using

[1]The GuPPI is a PCI board that makes it easier to continuously transfer sampled data from the computer's main memory to a hardware daughter card specific to the particular application[11].

Transmitter:



Receiver:

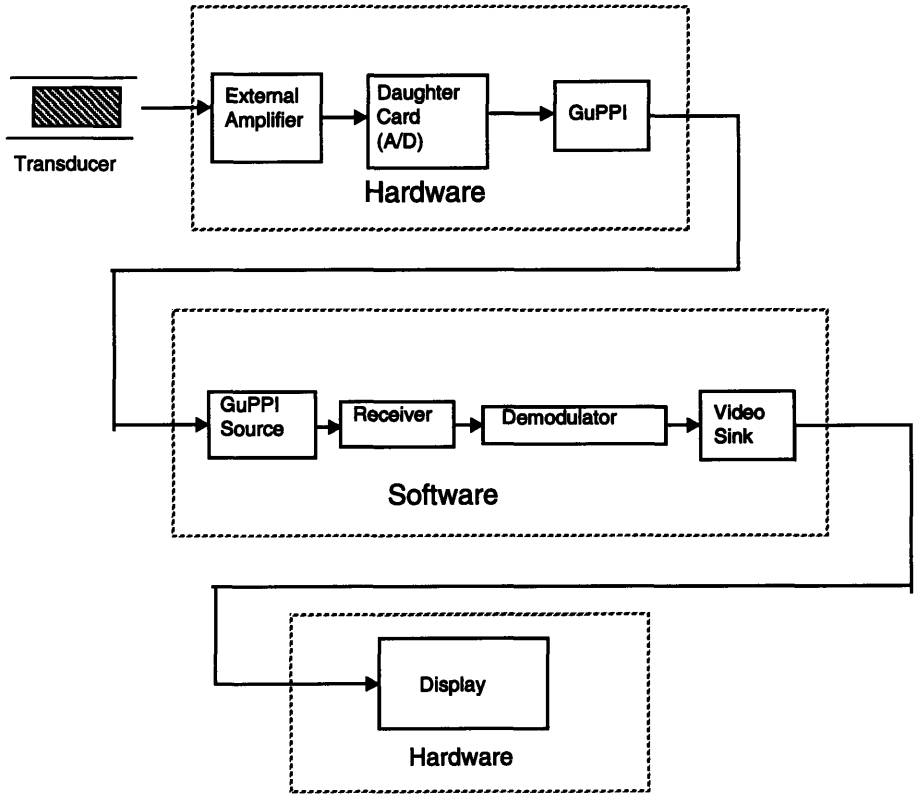


Figure 3-2: Block Diagram of the Entire Prototype Ultrasound System

an A/D converter. This digital waveform is transferred to software accessible memory using the GuPPI. The received software "echo" is amplified and a TCG function is applied to it. In order to remove unnecessary oscillations, this signal is then demodulated. The demodulated signal is then displayed on a software-generated oscilloscope-like display.

One of the aims of the software-based ultrasound system presented in this thesis is to simplify the processing required in ultrasound instruments as a first step towards developing more powerful systems with advanced signal processing and functional capabilities. The problem of having to synchronize the generation of pulses with the display, as well as the TCG unit in some systems, disappears in a software-based solution. The reason for this is because a software solution can temporally decouple the sample processing and display. In order to keep track of payloads, they are timestamped when received. Since the relative timing of samples can be regained at the display by making use of the timestamps, it is possible to process payloads without worrying about the strict time-synchronization issues involved in most real-time systems. This is clearly advantageous since synchronizing data at each stage of processing constrains the design, results in a significant amount of overhead, and can often lead to underutilized system resources.

Another inefficiency of hardware that software can rectify is the need to repeatedly regenerate the same waveform on the transmit side. In hardware systems, generating the transmit impulse usually requires a clock module to trigger the pulse generator (and the display). In effect, a significant analog computation is performed each time a pulse is generated or only one type of pulse(or possibly a few types of pulses) can be generated. In software this problem is eliminated by "generating" the samples making up the pulse once, and storing them in memory. The waveform stored in memory is changed only when parameters are changed.

The fact that the electronic components that make up the receiver side of hardware systems have to be so well designed to properly handle the incoming echoes means that the cost of these component is very high or some quality is sacrificed. Realizing perfect amplifiers and filters is much easier to do in software since functions can be computed with far greater accuracy and reliability than using electronic circuit components. This fact can be exploited in order to produce software TCG units that are more functional than their hardware counterparts. In software a TCG function can be applied and the region over which it is active can be dynamically changed, giving the user the ability to change the way

certain structures appear on the display. Such features are either too expensive or almost impossible to implement in hardware.

Although compression of the signal, as a way to reduce its dynamic range, is necessary in software, the algorithm to do so can be quite different. A software scheme can make use of advanced compression algorithms, such as those used in displaying video images. This is a significant improvement over hardware-based logarithmic amplifiers which can add noise to the system.

The particular system built and described in this report is an A-mode ophthalmic ultrasound system. An A-mode system was chosen over a B-mode one because the time required to manage the increased complexity of the hardware needed for the latter system (since the system needs to be driven such that pulses are generated along multiple lines through the scan plane) would have taken away from the effort to develop novel ways to process the information in software. Developing a software-based A-mode system is an important step towards developing software-based B-mode ultrasound. Ophthalmic ultrasound was chosen over other types of ultrasound systems for two primary reasons: ease of use and cost. As opposed to other types of ultrasound, like fetal ultrasound, where it is impossible to obtain meaningful results by scanning oneself, one can easily perform an ophthalmic scan without any special preparations. Model eyes are also readily available, making it easy to calibrate the system and evaluate its performance. A high quality ophthalmic system can be purchased for about $20,000. This is significant since the most reliable way to evaluate a prototype software ultrasound system is to compare its performance to existing technology.

Chapter 4

Software Implementation

The ultrasound software was developed in the VuSystem programming environment, making it easier to change various parameters and adding an extra degree of flexibility to the system. The VuSystem modules developed to support ultrasound processing serve three major functions: generating transmit pulses, modeling the transducer and target, and processing received echoes. Figure 4-1 illustrates the relationship between the software modules that make up the prototype ultrasound system when connected to the appropriate hardware. A description of the VuSystem along with the three major functions of the software modules and a description of the software simulation environment is presented in the following subsections.

4.1 VuSystem

The VuSystem is a UNIX-based programming environment that facilitates the visualization and processing requirements of compute-intensive, analysis-driven multimedia applications, and allows the software-based manipulation of temporally sensitive data. This system, developed by members of the Software Devices and Systems Group at the MIT Laboratory for Computer Science [15], is unique in giving the user both the programming advantages of visualization systems and the temporal sensitivity of multimedia systems.

VuSystem applications have components which do in-band processing and components which do out-of-band processing. The in-band processing is performed on all data. This type of processing is continuously performed on the stream of multimedia fragments (or ultrasound samples) that the system must handle. The out-of-band processing is performed

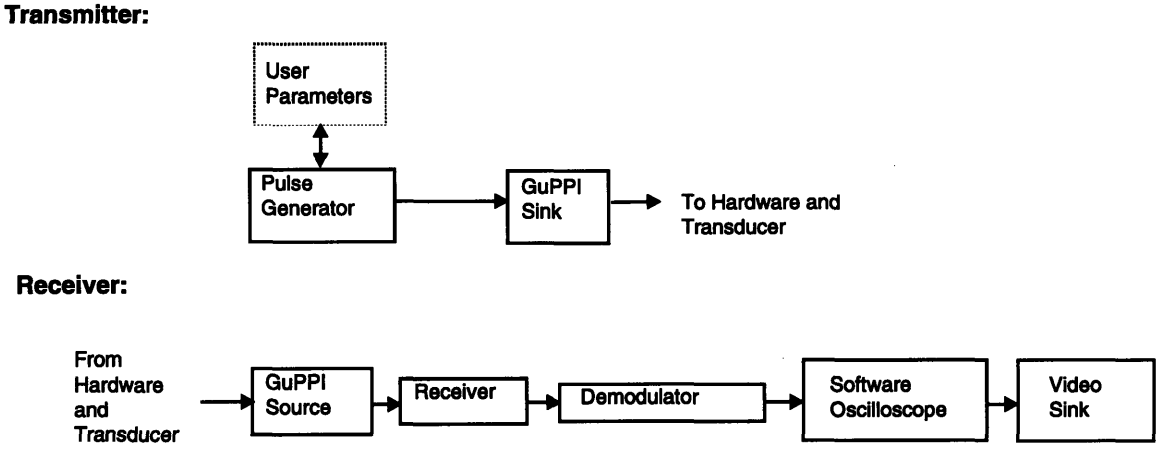


Figure 4-1: Block Diagram of VuSystem Software Modules

according to specific user events, such as mouse and key clicks.

The in-band components of applications developed in the VuSystem are written as modules that are C++ classes. By linking all of the modules in the VuSystem together, individual modules can make use of previously-written modules to perform very elaborate tasks. These software modules exchange data with each other via output and input ports. Data is passed in the form of dynamically-typed memory objects, referred to as payloads. These payloads, which contain timestamps to indicate when they arrived at a particular module, consist of a header which describes the payload (e.g., what type it is, how long it is, etc.) and the actual data being transmitted. There are different types of payloads for various media types (e.g, audio, video).

The three main types of modules in the VuSystem are Sources, Sinks, and Filters. Sources have one output port and no input ports. These modules, which generate payloads for processing, are often interfaced to input devices. Sink modules have one input port but no output ports. They are responsible for freeing up the memory allocated to payloads. Usually sinks are interfaced to output devices such as displays or hardware-based transmitters. Filter modules contain one or more input and output ports. Any processing that needs to be done on payloads is performed by filter modules. The VuSystem uses its module data protocol to pass payloads from upstream modules to downstream modules.

VuSystem filter modules make use of `WorkRequired` and `Work` member functions. The `WorkRequired` member function determines whether or not the function needs to perform work on the incoming payload, since the filter will only work on certain types of payloads. If the filter doesn't need to or can't work on the payload, it is passed on to a downstream module. Otherwise, the filter performs some work on the incoming payload in the `Work` member function.

The out-of-band components of the VuSystem are written in an extended version of the Tool Command Language (Tcl), an interpreted scripting language. Tcl scripts are used to configure and control the application's in-band modules and the graphical user-interface (GUI). Using Tcl makes it easier to combine modules together to develop applications, particularly since Tcl has a simple interface to C++.

The VuSystem facilitates code reuse by allowing the programmer to combine basic modules to perform specialized tasks. If necessary, customized modules can be developed to perform more complicated tasks. The GUI in the VuSystem makes it easy for the user to

change parameters as the application is running. It is also possible to dynamically change the way an application works by connecting and disconnecting various modules. The VuSystem is advantageous because it is designed to run on any general-purpose UNIX workstation running X-Windows, and doesn't require any specialized hardware for real-time processing.

The VuSystem is useful in a software-based ultrasound system because it gives the user the ability to easily control the parameters of the system (e.g., TCG, PRF). It also adds an extra level of flexibility since the system could easily be switched from one application to another (e.g., from an ophthalmic ultrasound system to a cardiac one).

4.2 Pulse Generation

The transmitter module makes it convenient to generate different types of waveforms (e.g., sine, square, etc.) with varying amplitudes and frequencies. This makes it easier to use one machine for various types of ultrasound and gives the user the flexibility to alter certain parameters based on what they are seeing on the display or would like to see. A high voltage, gated sinusoid (at the resonant frequency of the transducer) is required to excite the transducer in some ultrasound probes. However, many ultrasound probes require only a short-duration, high voltage pulse to excite the transducer. Figure 4-2 illustrates the difference between the two methods of pulse generation. Since the hardware requirements of the former method make it difficult to implement (see Chapter 5) and most probes only require the latter method, only the short-duration pulse method is described here.

4.2.1 The Short Duration Pulse Approach

It would prove to be computationally expensive if the same waveform were continuously generated by the transmitter module. For this reason, the waveform is computed according to user-specified parameters (pulse repetition frequency, duration, amplitude, sample rate, offset, and payload size) and stored in memory. More likely than not, these parameters will change infrequently. Thus, the series of samples to be transmitted is buffered in memory and only recomputed if the user changes the parameters. The memory consists of a sequence of samples which can be sent to a D/A converter when the hardware is integrated with the software.

The pulse repetition frequency presently used is 10 Hz. As stated in Section 2.2.1, the

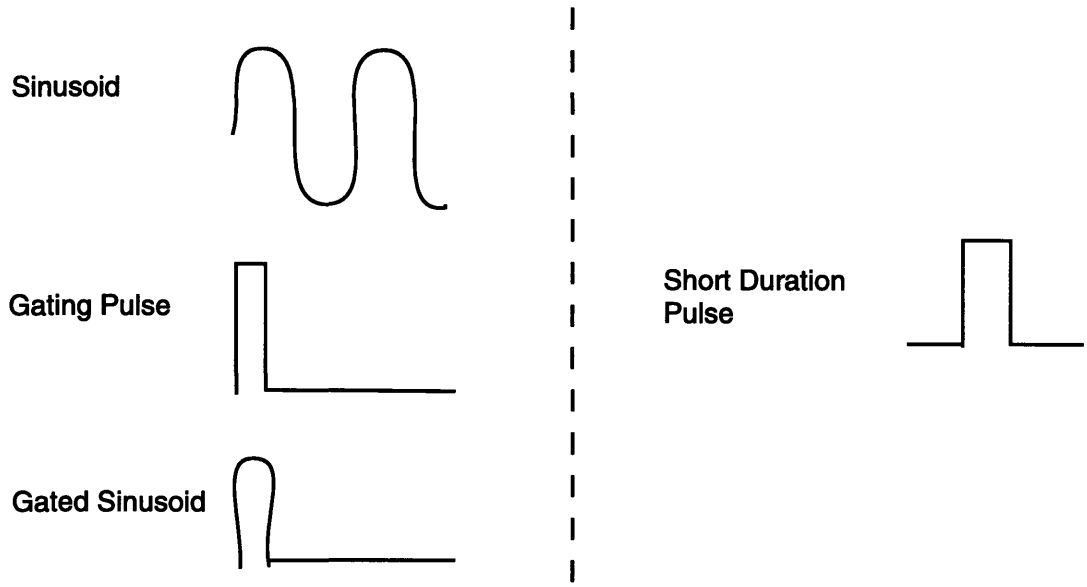


Figure 4-2: The Gated Sinusoid and Short-Duration Pulse Methods of Pulse Generation

maximum pulse repetition frequency is given by the following equation:

$$PRF_{max} = v/2d \tag{4.1}$$

where v is the velocity of the wave in the medium and d is the furthest reflecting interface. Thus, with a PRF of 10 Hz, and a speed of 1540 m/s, a depth of 77 m can be scanned. Obviously, the eye is not this large (it is only around 3 cm), but the extra margin ensures that echoes generated from one transmitted pulse do not interfere with those from another, and affords the software considerable time to complete its processing between pulses. B-scan ultrasound systems, and some A-scan systems as well, have PRFs that are as high as 4 kHz. In the case of the B-scan, such a high PRF is understandable since 128 lines are usually scanned for each sector plane, which is displayed at 30 Hz. However, for older A-scan systems, the reason why the PRF is so high has to do with persistence on the oscilloscope display. In order to maintain the intensity of the trace of the A-scan waveform on the display, it is necessary to generate the waveform at a very high rate. However, in the case of a software-based system, issues of persistence are no longer important since digitized data is being displayed on a CRT.

Since the signal used to excite the transducer is a short-duration pulse, it is important to have a sample generation rate such that there is enough resolution to represent the pulse.

In order to get pulses on the order of 1 μsec in duration, it is necessary to have a sample rate of at least 1 MHz. However, a rate of 1 MHz would represent such pulses with only one non-zero sample. As a way of increasing the accuracy of these short-duration pulses, and allowing for the possibility of pulses less than 1 μsec in duration, a sample generation rate of 5 MHz was used.

As stated above, in order to avoid extra computations, the waveform to be sent out is stored in memory. If the user happens to change one of the parameters, this waveform is then recomputed. However, since there is some latency associated with the scheduling of modules in the VuSystem, it makes sense to store more than one period of the waveform in memory. This way, over a given time interval, multiple pulses can be sent out, instead of having to accurately and frequently schedule the playout of a buffer containing a single pulse worth of samples.

4.2.2 Module Details

This subsection describes the detailed operation of the pulse generation module. The first part describes the control panel which allows the user to alter system parameters in real-time. The next section describes the characteristics of the payloads generated by this module and passed on to downstream modules in the program. Then, the code written to create this module is described. Finally, the targeted performance of the module is discussed.

Control Panel

The Pulse Generator control panel, like all control panels in the VuSystem, allows the user to change the system parameters as the system is running. The Pulse Repetition Frequency control panel, allows the user to specify the frequency at which payloads are transmitted (and thus the frequency at which the transducer is excited with a pulse). This control allows the user to select a frequency from 0 Hz (i.e., pulses are never transmitted) to 5 kHz, with a default value of 10 Hz. The Pulse Duration control allows the user to specify the length of the pulse, in μsec, ranging from 0 to 10 μsec, with a default value of 1. The offset control allows the user to shift how the waveform is displayed on the virtual oscilloscope display. This control takes on values between -32767 and 32767, with a default value of 0. The amplitude control allows the user to set the value of the outgoing pulse. Although the data samples generated by the Pulse Generator module are unsigned shorts, the user can

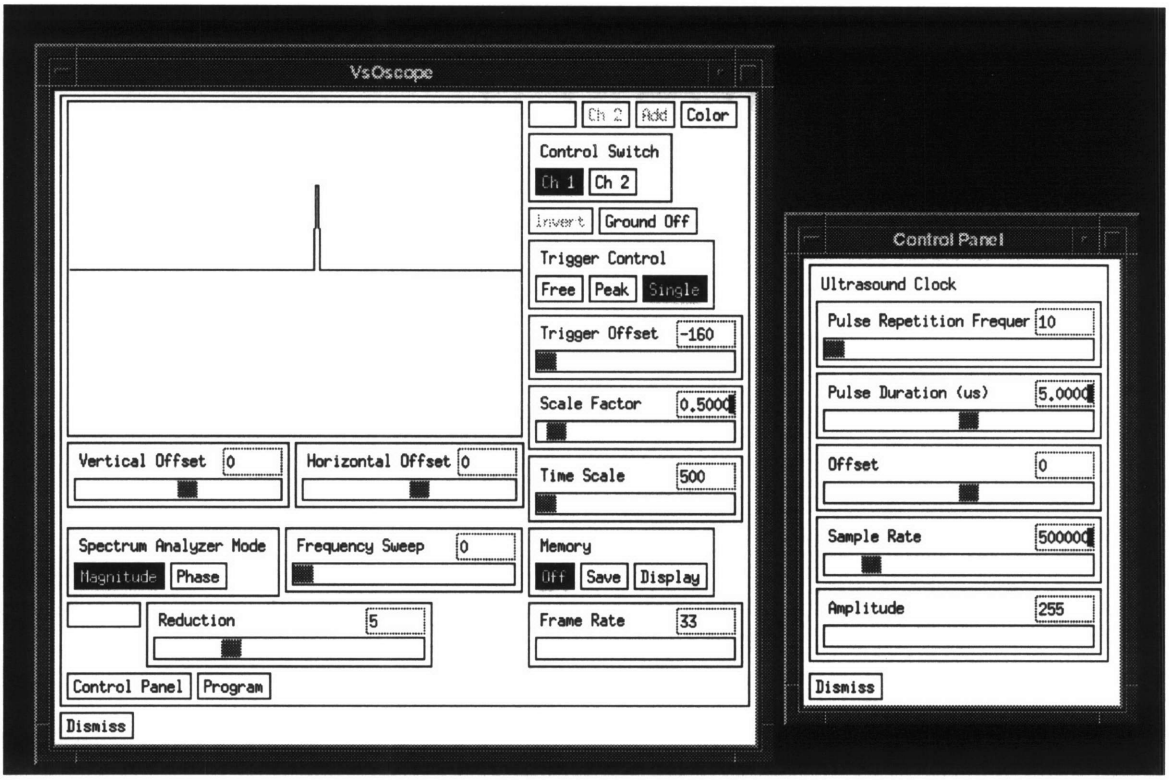


Figure 4-3: Pulse Generator Display, including its VuSystem Control Panel

pick an 8-bit value (from 0 to 255) to represent the amplitude of the pulse. This value is then scaled into the 16-bit value generated by the module. The default amplitude is set to 255. Figure 4-3 shows a screen shot of the generated pulse as well as the Pulse Generator control panel.

Output Payloads

The Pulse Generator module is a VuSystem "source" module with one output port through which payloads are passed on to the next module. These payloads contain the pulses used to excite the transducer (as described in Section 4.2.1. The output payloads are of type **`VsSampleData`**, which are the payloads generally used in signal processing applications in the VuSystem. These payloads have a header which indicates: the starting time of the payload (obtained by keeping track of the current time), the channel number (set to 0), the number of bits per sample in the payload (set to 16), the sampling rate (specified by the user), the number of bytes required to store the data (set to twice the number of samples in the payload since each sample is two bytes), the encoding type (set to ShortAudioSampleEncoding which indicates that samples are shorts and are integer-valued), the byte order (MSB first or LSB

first, set using a parameter that assumes the proper value based on the CPU design being used), and the total number of channels used (set to 1). Assuming that the user chooses to store only one payload, the data portion of each payload (i.e., the size of the payload stored in memory) is 8,192 bytes (8 KB).

Code Description

The Pulse Generator module is a C++ class with `Start`, `Stop`, and `TimeOut` member functions. The `Start` and `Stop` member functions are used to start and stop the operation of the module. The `Start` function is called by the VuSystem to initialize the module at the start of in-band media processing. It first sets the time interval in which payloads will be generated by the module. This time interval corresponds to the user-specified Pulse Repetition Frequency. It then calls `VsEntity::Start` to invoke the `Start` member function in any children of the Pulse Generator module. In-band processing is reset when `Start` calls `StopTimeOut` to cancel any scheduled timeout operations and release any payloads in the module. In order to initialize in-band processing, `Start` creates a `VsStart` payload with the current time as the starting time, and calls `Send` on the output port to send it downstream. The in-band flow of data begins when `Start` calls `Idle` if its input parameter, `mode`, is false.

When in-band processing has concluded, the VuSystem invokes the `Stop` member function to terminate processing smoothly. It is much like the `Start` member function since it calls `VsEntity::Stop` to ensure that the `Stop` member function of any child modules is called. The `StopTimeOut` member function is invoked to cancel any scheduled timeout operations. Then, it deletes any payloads inside the module. Finally, if the `mode` parameter, which is the input argument to the `Stop` function, is false, a new `VsFinish` payload is generated and sent downstream, timestamped with the current time in order to indicate the time at which in-band processing was terminated.

The `TimeOut` member function is called to perform operations that are time-sensitive, such as sending payloads to downstream modules. The `Idle` member function calls the `StartTimeOut` scheduler interface function so that `TimeOut` is called after the time interval calculated in the `Start` member function has elapsed. What happens is that the VuSystem scheduler calls `TimeOut` each time the elapsed interval has expired, such that payloads are generated at the appropriate rate. The `StopTimeOut` scheduler function is used to cancel a scheduled timeout.

Since the waveform is stored in memory, memory is allocated only the first time the module is run or if any of the parameters are changed. This is done by setting a flag each time a parameter is changed and deciding to allocate memory based on its value. The `TimeOut` member function first checks to see whether or not this flag has been set or if this is the first time the module has been run. If either of these cases is true, memory is allocated for the payload, so that next time it can be sent out without unnecessary computations (assuming parameters have not been changed).

When creating the payload to be stored in memory, the `TimeOut` function first generates the segment of the data corresponding to the pulse, followed by a string of zeros for the time period in which the pulse is off (i.e., the time period during which the transducer is receiving echoes). The `TimeOut` function must finish creating the output payload in the allotted time interval. The following code fragment illustrates the work of the `TimeOut` member function in the case in which a payload needs to be created (when a flag has been raised or the module is running for the first time) and is then sent to the next module:

```
// Generate a payload as long as there currently isn't a payload to be
// sent and the allotted time for sending the payload has not expired
while (payload == 0 && currentTime < nextTime) {
  if (firstCall == 1 || flag == 1) {

    // Grabbing a pointer to the memory block allocated for the payload,
    // in order to generate the payload data
    u_short* payload_data = (u_short*)memory_block.Ptr();
    u_short counter = 0;

    // Calculating the number of samples that make up the pulse by
    // first taking the duration (which is in microseconds) and
    // converting it into seconds, and then multiplying by the sampling
    // rate.  The appropriate data values are set according to the user-
    // specified amplitude (an 8-bit quantity which is scaled into an
    // unsigned short)
    u_short pulseSamples = (u_short)((duration / 1000000) * sampleRate);
    int value;
    while (counter < pulseSamples) {
      value = (int)((amplitude/(float)255)*(float)32767+(65535/2)+offset);
      if (value > 65535) value = 65535;
      if (value < 0) value = 0;
      *payload_data++ = (u_short) value;
      counter++;
    }

    // Setting the remaining data values to the zero value (which is
    // 32767 on a 16-bit scale)
    while (counter < payloadSize) {
      *payload_data++ = (u_short) ((65535/2) + offset);
      counter++;
    }
    firstCall = 0;
```

```
      flag = 0;
    }
    // Creating the new SampleData payload and specifying the values of the
    // header fields
    new_payload = new VsSampleData(time, 0, 2*payloadSize,
    sampleRate, VsShortAudioSampleEncoding,
    16, HOSTORDER, 1);
    new_payload->Samples() = payloadSize;
    new_payload->Data() = mem;
    new_payload->ComputeDuration();
    payload = new_payload;

    // Incrementing the time to prepare for generating the next payload
    time += timeStep;
    nextTime += timeStep;

    // If you are able to send the payload (via the output port), send it
    // and set the payload to zero
    if (outputPort->Send(payload)) payload = 0;
  }
```

This code fragment shows that first the data is written into a memory block. Then, the header information is generated and stored, together with a pointer to the memory block, in a new **VsSampleData** object. Once the parameters for the payload have been set, the current time and the time until the next payload needs to be sent out are both updated. Finally, the new payload object is sent to a downstream module by calling the **Send** function of the output port.

Performance Targets

The size of the payload generated by the Pulse Generator module, assuming only one payload is stored in memory, is 8,192 bytes. For this payload size, and for a sampling rate of 5 MHz, the time required to send the payload out, assuming the sampling rate corresponds to 16-bit samples, is approximately 0.8 msec. Since this module generates non-continuous bursts of data, consisting of high values representing the pulse and zeros everywhere else, the performance requirements are not as stringent as those of modules that continuously generate data. Using the default PRF of 10 Hz, the Pulse Generator module has 100 msec to send out the 0.8 msec payload[1].

Given the above numbers, it seems quite reasonable to expect the system to be able to easily handle Pulse Repetition Frequencies of 10 Hz and higher. Higher PRFs can become difficult to attain since there is a significant amount of overhead associated with generating

[1]The GuPPI is responsible for holding the waveform to a default value during intervals between payloads.

payloads (e.g., scheduling timeouts), even if they are stored in memory. One way to minimize the overhead per payload would be to store larger-sized payloads in memory (corresponding to multiple cycles of the transmit waveform).

4.3 The Receiver

The receiver module allows the user to select the amount of amplification that will be performed on every incoming sample, and the maximum amount of amplification that can be performed. The user may also select what type of amplifier will be employed (log or linear), whether or not TCG will be used, and the slope of the function (in dB/ms) and onset of the delay (in mm) if TCG is used.

In traditional ultrasound systems, the clock signal is not only used to determine when a pulse is to be sent, but it is also used to properly synchronize the Receiver/TCG unit and create the time base for the display. However, in the VuSystem implementation, such synchronization is unnecessary since the received sample payloads are time-stamped with their time of arrival. They can be processed on a loosely synchronized basis so long as the payload's time stamp can be used to regenerate the timing on the display.

The operation of the receiver is fairly straightforward. This module assumes that the speed of sound is 1540 m/s. This assumption, along with knowledge of the sampling rate, is necessary in order to determine the first sample at which TCG is to be applied (if it is used and if a delay has been set).

If TCG is not being used, then all samples are amplified by the same factor (specified using the initial amplification control). If TCG is being used, then only the initial set of samples, up to the sample before the first sample at which TCG is to be applied (set by the delay) are amplified by this initial amplification. A TCG function is applied to the remaining samples. With a linear function, earlier samples (those corresponding to closer structures) are amplified less than later ones. With the logarithmic function, small differences between closely spaced samples can more easily be seen. The slope of both functions is specified by the user.

4.3.1 Module Details

Control Panel

A screen shot of the Receiver control panel along with the output of the Receiver module is shown in Figure 4-4. The topmost control is a switch that allows the user to select the type of amplification function to be used (linear or logarithmic), with the default option being linear. The initial gain control lets the user specify the initial gain, in decibels, which is applied to the samples. This value ranges from 0 to 100 dB, with a default of 40 dB. The third control is the maximum gain control, which lets the user choose the maximum amount of gain to be applied to any sample, regardless of whether or not TCG is turned on. This control can be set between 0 and 120 dB, with a default of 100 dB. The delay control lets the user specify at what point the TCG function will first be applied (in mm). This value is converted into a sample delay, by using the sampling rate. The delay ranges from 0 to 5 mm, with the default delay being 1 mm. The next control is the slope control which lets the user specify the slope of the TCG function to be used (ranging from 0 to 20 dB/msec), with a default value of 1 dB/msec. Finally, the TCG switch allows the user to select whether or not TCG will be applied to the samples.

Input/Output Payloads

The Receiver is a filter module, so it has one input and output port. In a fully functional ultrasound system, the Receiver would get its payloads from the GuPPI Source module. However, when the system is run in simulation mode, using a simulated transducer (see Section 4.5), payloads come from the Transducer module. The size of the payloads into the Receiver module depends on the payload length, which is set in the Pulse Generator module (in the simulation environment). In order to speed up processing, the payload size is an integer multiple of the operating system memory page size.

In the default case, payloads are 4096 samples, or 8 KB in size. This size payload is appropriate to capture echoes from various depths, given that the sampling frequency on the receive side is 20 MHz. With these parameters, and assuming that the acoustic wave is traveling at 1540 m/s, echoes returning from depths of up to approximately 16 cm can be captured in a single payload. The received payloads are of type `VsSampleData`, and their headers contain information about the starting time, channel number, number of bits per

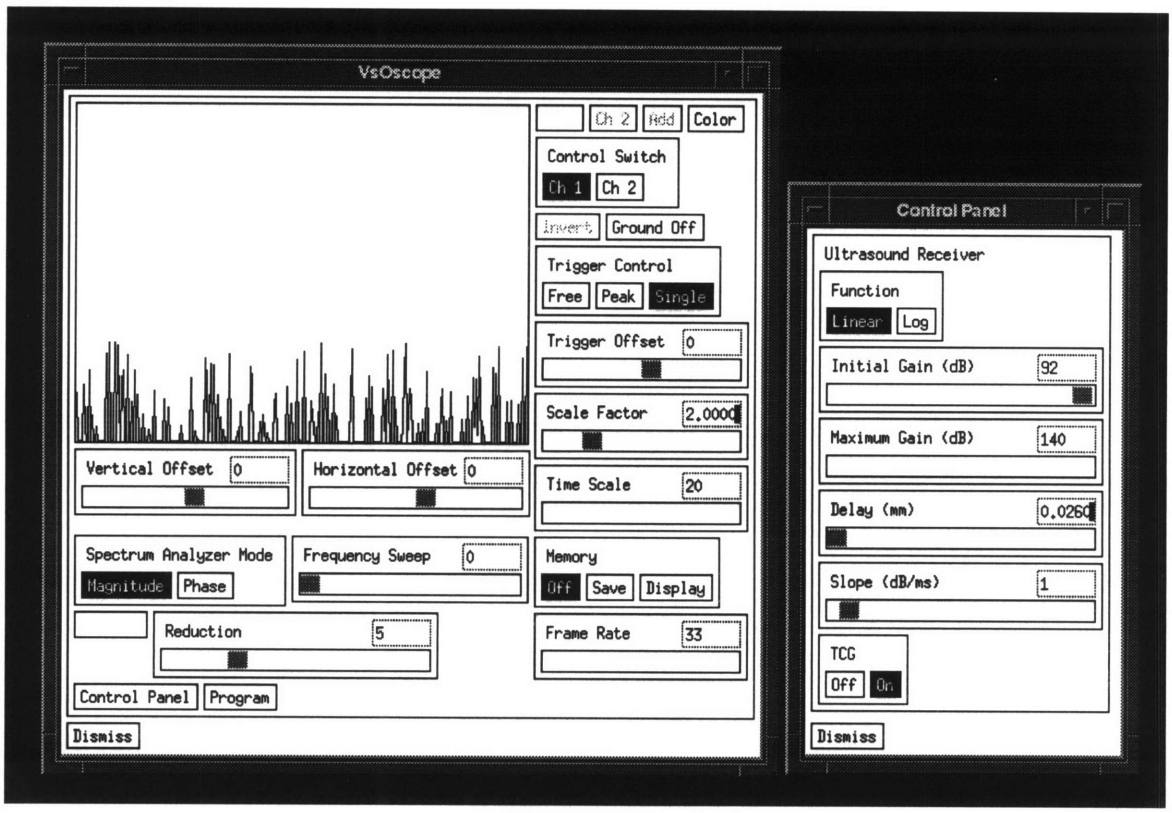


Figure 4-4: The Receiver Display, including its VuSystem Control Panel

sample, the sampling rate, the number of bytes required to store the data, the encoding type, the byte order, and the total number of channels used. The values of these parameters are the same as those set in the Pulse Generator module (see 4.2.2). The output payload from the Receiver is of the same type as the input payload, and passes on the same memory buffer.

Code Details

This module's operation is dependent on the features selected by the user. The first case is if linear amplification is selected and TCG is not in effect (either because it is off or because the current sample is less than the sample corresponding to the onset of TCG, determined by the user-specified delay). In this case, all samples are amplified by the amount specified by the initial amplification (set by the user). In the second case, in which the logarithmic amplifier is selected and TCG is not in effect, the logarithm of each sample is taken before it is multiplied by the initial amplification.

The remaining cases are for when TCG is in effect. The following line of code creates

the actual amplification based on the user's input for the slope of amplification (which is in dB/msec):

```
float slopeAmp = pow(10,((float)(((i - sampleDelay) *
 ((float)(1/(float)sampleRate)) *
 pow(10,3) * slope)/20)));
```

In this case, for each sample the total amplification is **slopeAmp** times the initial amplification. The factor of **i** corresponds to the sample number. Thus, the later the sample (samples that arrive later are from deeper in the body), the larger the amplification. This is exactly what TCG is supposed to do. The final case is when TCG is selected in conjunction with logarithmic amplification, in which case the output is the logarithm of the input sample times the total amplification (described above).

In every case, computations are done as floats, and then quantized to unsigned shorts in order to keep the payload characteristics unchanged. If the total amplification is ever greater than the user-defined maximum amplification, the value is clipped to the maximum amplification.

Performance Targets

In order to ensure the maximum throughput of the system, the Receiver module, on average, must be able to process payloads as quickly as it receives them. Assuming that payloads are generated at a rate of 10 Hz, the Receiver module must be able to process payloads at this same rate. Ten payloads per second corresponds to 100 msec per payload. As mentioned above, the Receiver payload size is 4096 samples, because this is the smallest integer multiple of the operating system page size that will allow the entire region of interest to be captured in a single payload. This payload size corresponds to at most 24 μsec of processing time per sample. Even with the overhead of computing the amplification and looping through the data portion of the payload, these requirements are reasonable since only one multiplication operation is performed on each sample.

4.4 The Demodulator

The demodulator removes the oscillations of the echo due to the frequency of the incident pulse, and leaves only the envelope of the received signal (which is the most relevant information). The demodulator breaks the incoming payload (which corresponds to the

amplified echo) into slices. It then attempts to fit each slice to a sinusoid and detect the peak within this region.

The first step in demodulation is determining the size of the slice. The size of the slice depends on the bandwidth of the signal returning from the probe. This bandwidth is influenced by the fundamental frequency of the signal as well as the quality factor (Q) of the transducer. Q is the ratio of the center frequency to the bandwidth. The bandwidth can be found using the following equation:

$$bw = f_1/(2 * Q) \tag{4.2}$$

Since the center frequency is 10 MHz, and the Q is 3.9 [26], the bandwidth is around 1.3 MHz. In order to meet the Nyquist criterion, the envelope must be sampled at twice this rate, or 2.6 MHz. The slice is determined by scaling the sampling frequency (f_s), which is 20 MHz, by the envelope sampling frequency (2.6 MHz). Thus, the number of samples in the slice is given by:

$$slice \geq f_s/(2 * bw) \tag{4.3}$$

This corresponds to a slice of around eight samples, meaning one output sample is produced for every eight input samples.

Once the slice length is known, Maximum Likelihood (ML) detection is used to predict the peak for a given slice. The following is the basic formula for determining the amplitude in ML detection:

$$amp_{ML} = (2/slice) * \sqrt{(\sum_{n=0}^{slice-1} input * sin(2 * \pi * f_1 * n))^2 + (\sum_{n=0}^{slice-1} input * cos(2 * \pi * f_1 * n))^2} \tag{4.4}$$

where *input* is the value of the particular sample. Using this formula, an amplitude (amp_{ML}) is generated for each slice. These amplitudes are then grouped together into an outgoing payload which corresponds to the envelope of the incoming payload. In order to maintain a constant size of outgoing payloads, some samples are leftover and are grouped with the samples from the next payload. This insures continuity in processing and requires only a small amount of memory to store the samples.

4.4.1 Module Details

Input/Output Payloads

The demodulator is another filter module that has one input port and one output port. The incoming payload is received from the Receiver module. It is the same size as the input payload to the Receiver module. Since demodulation produces one output sample for every eight input samples, according to the above calculations, a new output payload is generated. The size of this output payload changes depending on the number of integer multiples of the slice in the data set. However, it is still of type `VsSampleData`. These payloads are sent to the Oscilloscope module and then to the Video Sink module so that they can be displayed on the screen.

Code Details

After the initial calculations, in which the size of the slice is determined, the data is analyzed. If, for some reason, the size of the data is smaller than the size of the slice, the data is discarded. For every slice of data, an amplitude is calculated, using ML estimation (as described above). The following lines of code demonstrate this calculation:

```
// Determining the Carrier Frequency to be used in Demodulation
float carrier = (TWOPI*(float)frequency) /(float)samplingFrequency;

// The payload has length ''len'' and is split into slices, so as long
// as the remaining length of the payload is still greater than or
// equal to the slice size, we can still split the payload into a
// slice to be used for ML estimation
while(len >= slice) {
  float numerator = 0;
  float denominator = 0;
  for (u_int i=0; i < slice; i++) {
    // Implementing the sums from the Algorithm in Equation 4.4
    numerator = numerator + (float)(*data) * (float)sin(carrier*i+PI/4);
    denominator = denominator + (float)(*data) * (float)cos(carrier*i+PI/4);
    data++;
  }

  // Estimating the Amplitude and setting the data value in the output
  // payload to this amplitude value
  float amp = (2.0/(float)slice) * (float)sqrt(pow(denominator,2) +
                                               pow(numerator,2));
  if (amp > 65535.0) amp = 65535.0;
  *outputData = (u_short) amp;
  outputData++;
  len = len - slice;
}
```

The frequency used for the carrier frequency is the center frequency of returning echoes (in our case 10 MHz). The amplitudes are quantized such that they fit in unsigned shorts. The leftover data values from one payload are saved in memory and combined with the initial data values from the next payload.

Performance Targets

Since the demodulator generates one output payload for every input payload, it must maintain the same payload rate as the other modules. However, since the output payloads are roughly 90% smaller than the input payloads, the bit rate of the demodulator is actually slower than that of the other modules. For each input data value, 6 floating point calculations are performed to get the values in the summations in Equation 4.4. In addition, 4 floating point operations are performed on each data slice and one floating point operation is performed on each output data value. Since the size of the incoming payloads is 4096 samples, this corresponds to 27,200 calculations per payload. Assuming a PRF of 10 Hz, this means that the module must perform an average of 272,000 floating point operations per second to keep up with the incoming data.

4.5 Software Simulation Environment

In order to properly test the functionality of the software part of the prototype system and demonstrate the viability of software ultrasound, without requiring any hardware to be connected, a test environment was developed featuring a model transducer/target module. This software module takes as an input the excitation pulse and outputs a model of the returned echo. Figure 4-5 is a block diagram that shows where the simulated transducer/target fits in the software system. The module was designed by using the equivalent circuit shown in Figure 4-6 [5]. In this circuit, C_o is the parallel-plate capacitance of the device (set to 0.0047 μF), R_m is the resistance representing the radiation of acoustic power (set to 50Ω), R_k accounts for leakage current (set to 10kΩ), and R_a accounts for internal absorption in the material (set to 50Ω). The impedances of L and C, whose values are 1.5x10^{-5}H and 1.645x10^{-5}, respectively, cancel out at the resonant frequency of the transducer. Using impedance and frequency domain methods, the impulse response of this circuit (h[n]) was computed. It was found to be (((-3.33E6*3.71E3)/ (6.66E6-3.71E3)) *

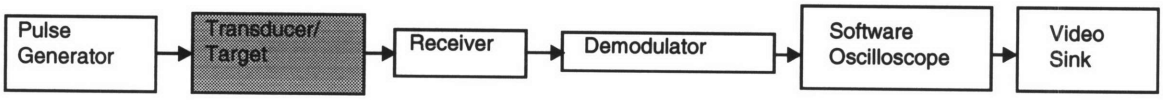


Figure 4-5: Block Diagram of Simulation Environment

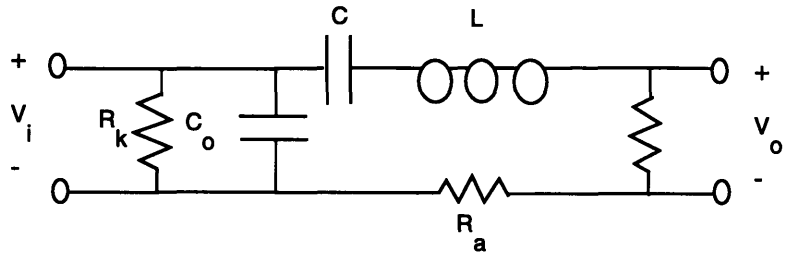


Figure 4-6: The Equivalent Electrical Circuit Modeling a Transducer and Target

$\exp^{-3.71E3*n}$) + (((-6.66E6*3.33E6)/(3.71E6-6.66E6)) * $\exp^{-6.6663*n}$).

4.5.1 Simulation Details

Input/Output Payloads

The module that models the effects of the transducer and target is a VuSystem "filter" module that has one input port and one output port. The input payloads to this module contain 16-bit unsigned shorts generated by the Pulse Generator module. The format of the output payloads is exactly the same as those of the input payload. Furthermore, since this module is passed a pointer to the input payload, and passes the same pointer downstream, it can make all the changes directly to the memory location where the payload is stored, instead of having to make additional copies of the payload.

Code Description

The transducer/target is "simulated" through the application of a generated filter. It would be computationally expensive to regenerate this filter every time a payload needs to be processed. Instead, the filter is generated once and stored in memory, until the size of incoming payloads changes (in which case the size of the FFT changes). Since the filter is computed in the time domain (see above), it can be convolved with the data in order to generate the output payload. However, a faster method involves using the Fast Fourier Transform algorithm [21]. Using this method, the Fourier Transform of the filter is multiplied with the Fourier Transform of the incoming data. Then, the inverse Fourier Transform of this output is taken in order to get the desired output in the time domain. The first step involves computing the FFT of the desired filter, which can be stored in memory. The following code illustrates how the FFT of the filter is generated and stored:

```
// Storing the time-domain version of the filter in an array whose
```

```
// FFT can be computed, where filterlength is the desired filter length
for (u_int n = 0; n < filterlength; n++) {
  *(fft_ptr+n) = ((-3.33E6*3.71E3)/(6.66E6-3.71E3))*
    exp(-3.71E3*n*((float)1/(float)SampleRate)) +
    ((-6.66E6*3.33E6)/(3.71E3-6.66E6))*
    exp(-6.66E6*n*((float)1/(float)SampleRate));
}

// The size of the FFT is the payload size plus the length of the
// filter, rounded to the nearest power of two (to make the algorithm
// more efficient
size = (int) pow(2,(int)ceil((log(payloadSize+filterlength-1)/log(2))));

// Zero Pad filter to the right length
for (u_int i = signalLength; i < size; i++)
  *(fft_ptr+i) = 0.0;

// Taking FFT of filter to get it in Frequency domain (the results are
// returned in the same array)
realft(fft_ptr-1, size, 1);
```

Once the filter has been computed and stored in memory, it can be applied to the data in the incoming payload. First, the FFT is taken of the data. The overlap-add method of convolution (using the FFT) is employed when applying the filter to the data. Since the data has been segmented into payloads, some of the data must be saved in order to properly implement this algorithm. The amount of data to be saved is equal to the size of the FFT minus the size of the payload. The size of the filter is set to 1024 samples, and the default size of the incoming payload is 4096 samples. Thus, the minimum size of the FFT would have to be 5120 samples. However, since the most efficient FFT algorithms use FFTs that are integer powers of 2, the size of the FFT is set to 8192 samples, in which case 4096 samples are saved.

Since, the FFT algorithm is a floating point algorithm, the input and output data are floats. However, the output payload of the transducer module must be unsigned shorts. Thus, after the output data has been inverse FFTed (to get back into the time domain), it is quantized to fit within a unsigned short.

4.6 Display

The ultrasound waveform is displayed using the Oscilloscope module developed by Andrew Chiu of the Software Devices and Systems group at the MIT Laboratory for Computer Science. This module acts like a regular oscilloscope, except that it is completely implemented in software. It accepts payloads of type `VsSampleData` and generates payloads that con-

tain display bitmaps of type `VsVideoFrame`. The latter are typically passed to a VuSystem `VsVideoSink` module, which in turn passes them to an X-Windows based display process. Alternatively, the oscilloscope module can be connected to a `VsFileSink` module, which causes the images to be written to a file for subsequent processing and viewing by a variety of pre-existing VuSystem image processing and display/player applications. Similarly, the output of the receiver or demodulator modules could be directed to a `VsFileSink` if it were desirable to save the raw sampled information for future use. Figure 4-7 is a screen shot showing the return echoes in the simulation environment using the software oscilloscope display.

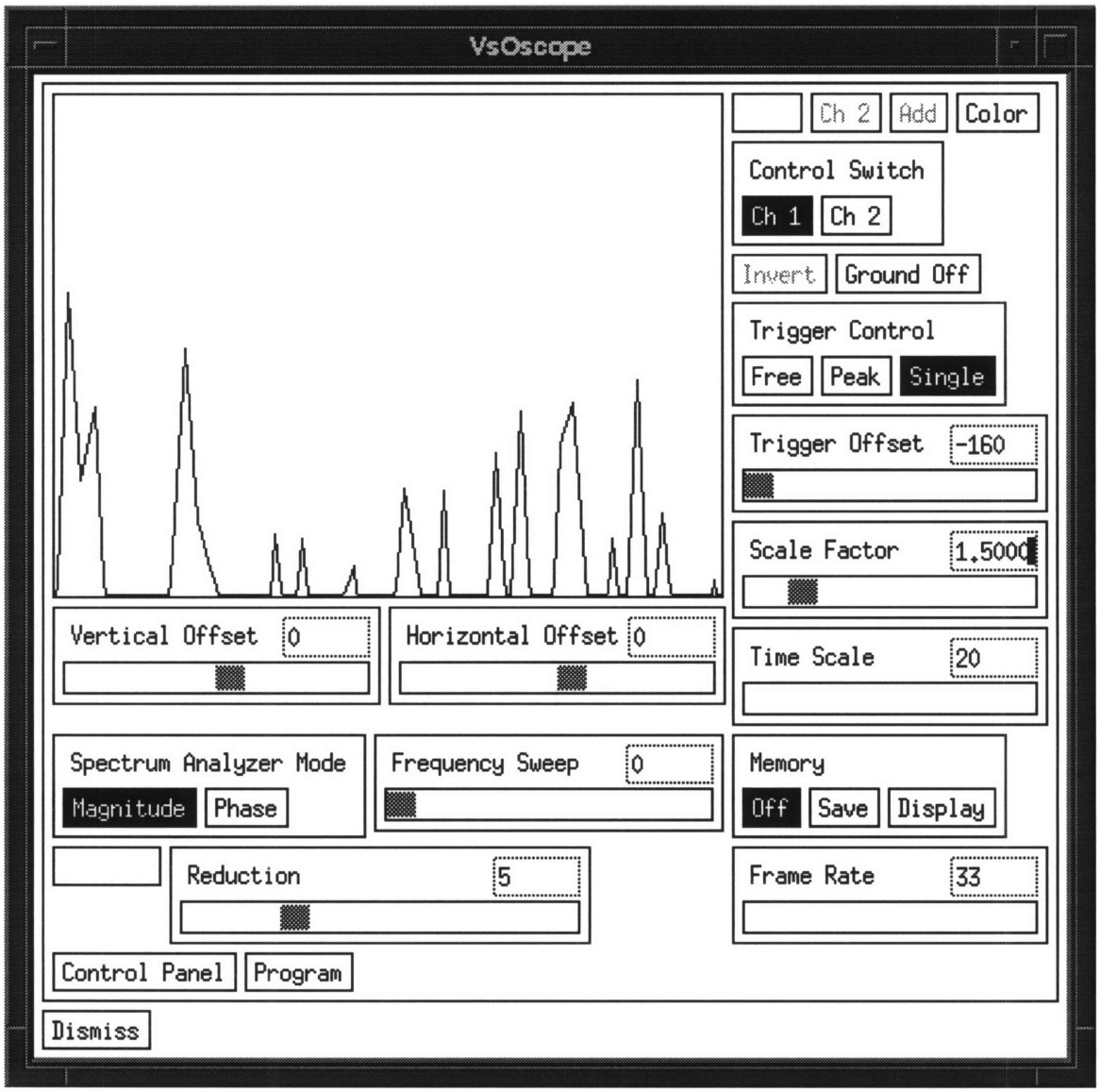


Figure 4-7: Return Echoes in the Ultrasound Simulation Environment

Chapter 5

Hardware and System Integration

5.1 Hardware

Although the ultimate goal of software devices is to push the hardware/software boundary as close as possible to the A/D converter, there are some limitations as to what software can currently do. For this reason, additional hardware can be built to make the system an operational ultrasound unit. In order to connect the hardware domain with the software domain, the GuPPI [11], a general-purpose PCI-bus Interface, a prototyping daughter card, and associated software modules should be employed.

The main reason additional hardware is necessary is because digital logic operates at very low voltage levels (usually between 0 and 5 V) and the transducer requires a very high excitation pulse (around 150 V). In order to create such a waveform, an external high voltage driver circuit was built. Receiver circuitry was also built to amplify the low-level ultrasound echoes to the full range of the A/D converter. These components can be integrated with the software to drive an ultrasound probe.

5.2 The GuPPI

In order to connect the software modules to the aforementioned hardware, the GuPPI can be used [11]. The GuPPI, which was developed by Michael Ismert of the Software Devices and Systems group at the MIT Laboratory for Computer Science, allows the continuous transfer of data between the host processor/main memory and the hardware specific to the application. The GuPPI, when interfaced with the D/A converter on the daughter card,

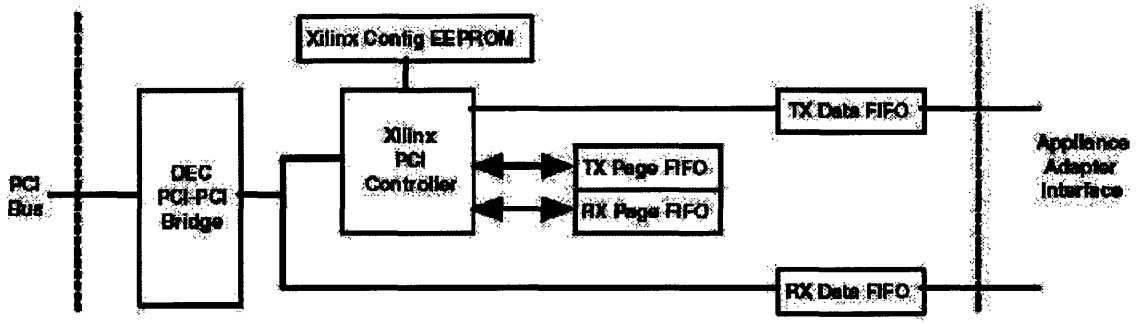


Figure 5-1: A Block Diagram of the GuPPI

allows samples generated in the VuSystem pulse generator module to be converted into analog waveforms used to excite the transducer.

The GuPPI, as shown in Figure 5-1, consists of four 32 bit FIFO banks. Two of these banks are used to buffer incoming and outgoing data traveling between the PCI bus and the back-end bus. The other two banks hold the page addresses for instructions that read and write to main memory. The back-end interface is used to connect to I/O devices specific to the application. In this case, the GuPPI is connected to the daughter card, containing the D/A and A/D converters, which is then connected to the ultrasound probe. The FIFO banks provide various control signals, flags, and status bits, some of which can be set by user-level software.

5.3 The Daughter Card

The prototyping daughter card, developed by Vanu Bose of the Software Devices and System Group, consists of A/D and D/A converters, programmable logic, as well as other circuitry needed to make these parts work with the GuPPI. Transmission and reception, the process of transferring data from the GuPPI to the D/A or A/D converter, are each controlled by an Advanced Micro Devices MACH230, a programmable logic device [2]. The MACH chips are capable of communicating with each other so that the transmit and receive sides can be synchronized to ensure samples are received right after the transmit pulse is generated. The Analog Devices AD9713, a high-speed (80 MSPS) 12-bit D/A converter, can be used to generate the analog representation of the short-duration pulse. The received echoes can be digitized using the Analog Devices AD9042, a 12-bit A/D converter. The 10 MHz received echoes can be properly handled by the AD9042 since the converter has a maximum data

rate of 40 megasamples per second (MSPS), which corresponds to a Nyquist frequency of 20 MHz.

5.4 Pulse Generation Circuitry

The signal produced by the D/A converter in the daughter card has a range of ± 2 V. The problem of trying to go from this voltage level to the high voltage level demanded by the transducer in the ultrasound probe is the reason why the gated sinusoid method is often not employed. With the gated sinusoid method, a high frequency, high voltage excitation pulse is used to excite the transducer and cause it to resonate. However, in order to excite the transducer, this excitation pulse must have a peak amplitude of 150 V. It is not practical to build a high voltage amplifier that amplifies a 10 MHz signal on the order of 30 dB. The best high voltage amplifiers give some amount of gain up to 1 MHz or so, thus falling short of the requirements of a gated sinusoid system. The power/bandwidth requirements of such systems present a significant challenge.

With most A-scan probes, all that is necessary to excite the transducer is a high voltage pulse. The pulse causes the piezoelectric crystal to oscillate at its resonance frequency particularly if it has a high Q (quality factor). The pulse must be short duration because otherwise the crystal would still be ringing when echoes were received. This added constraint means that the circuitry designed to amplify the pulse must be fast enough to produce pulse durations on the order of 100 ns.

One possible method to generate the high voltage, short duration pulse makes use of a monostable, FET, capacitor, diodes, and a 155 V power supply. The basic idea is that the waveform used to excite the transducer is generated by rapidly discharging a high voltage capacitor through the transducer. One of the goals in designing the hardware is to keep the design as simple as possible. The entire circuit designed for pulse generation is shown in Appendix A.

5.5 Receiver Circuitry

Figure 5-2 shows an overview of the receiver circuitry and its relation to the other hardware components. When an excitation pulse is being transmitted, the receiver is susceptible to the high voltage. In order to protect the receiver, a pair of parallel, reversed diodes is used

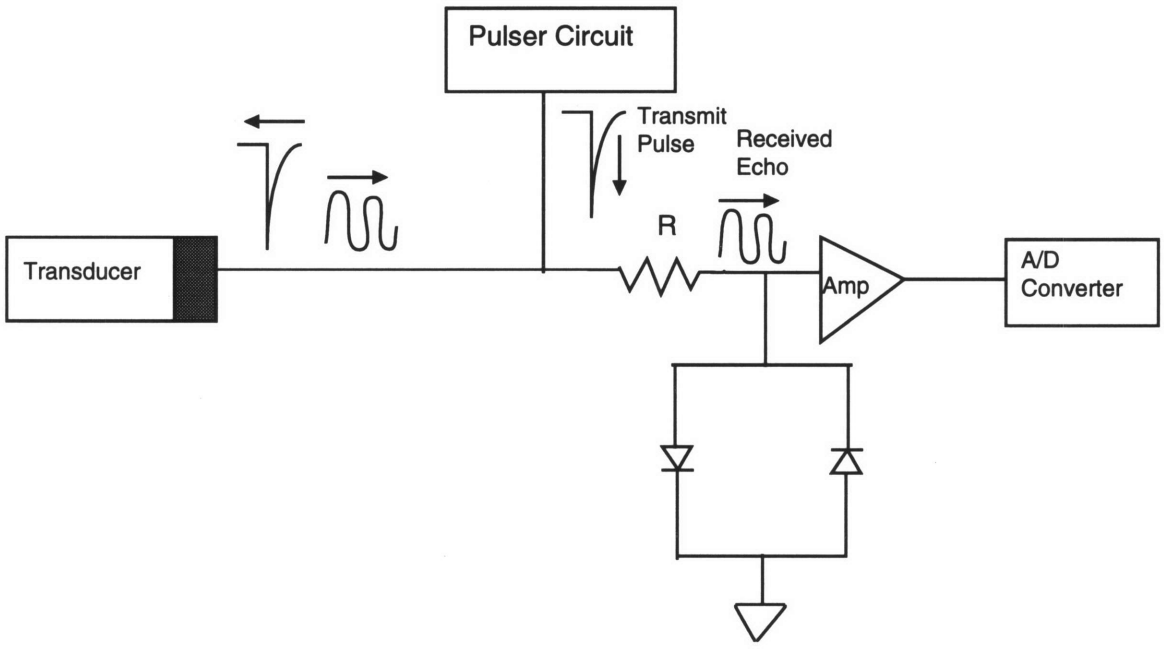


Figure 5-2: An Overview of the Receiver Circuit

[5]. The diodes appear as short circuits for input voltages above 0.7 V in magnitude (like the high voltage transmit pulse), and as open circuits for the low-voltage received echo waveform. The value of R needs to be higher than the input impedance of the transducer, in order to minimize wasted transmit power in R. Also, so that the maximum amount of the received signal is in fact received, the impedance of R in series with the amplifier should be smaller than the impedance of the transmitter circuit in the absence of an output pulse.

Since the received signal is on the order of 10 millivolts or so, it is important to amplify this signal before it is sent to the D/A converter. The reason for this is because the range of the D/A converter is -2 V to +2 V. Thus, if all of the samples are very low voltages, the differences between them will not be captured as well. For this reason, an external amplifier is required.

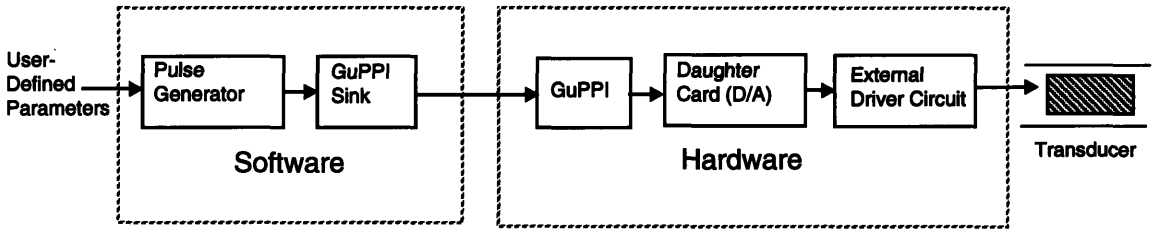
This receiver circuitry, when combined with the pulse generation circuitry, makes it possible to receive echoes from an ultrasound probe. The configuration of all of the external hardware designed for the prototype system is shown in Appendix A.

5.6 Integration

In order to use the GuPPI, two additional software modules must be incorporated into the system. These modules are the GuPPI Sink and the GuPPI Source modules. The GuPPI Sink module receives its input from the Pulse Generator module. It then configures the GuPPI DMA engine so that it can access the samples in the payload and present them to the D/A converter. The GuPPI Source module gets the received signal from the GuPPI. It then packages this information into payloads which are sent to the receive module. The size of these payloads is a multiple of 4096 (which is the GuPPI page size). A block diagram of the entire software/hardware system is shown in Figure 5-3.

Using customized hardware, such as the GuPPI, the daughter card, and the external amplifier may initially seem contrary to the overall goal of designing a software-intensive system. However, as software devices become more prevalent, the expectation is that every computer will come with a GuPPI (or something very similar). In this case, the remaining components could be integrated and sold with the transducer. Since improvements are usually made to the processing apparatus rather than the I/O devices, it is believed that the cost of upgrading a particular device will be the cost of upgrading the software since the same I/O device and associated daughter card will be reused.

Transmitter:



Receiver:

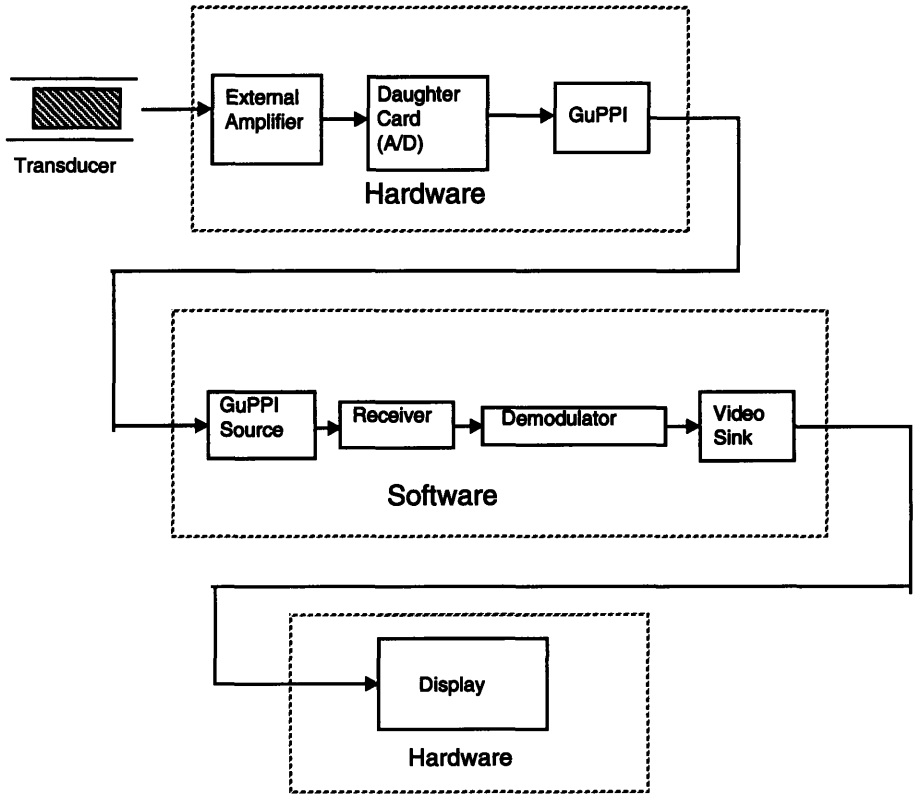


Figure 5-3: Block diagram of entire ultrasound system

Chapter 6

Results and Conclusion

In this work, I have developed a software-based ultrasound design that could leverage the flexibility of software. This report has described the prototype software ultrasound system. The following sections discuss the novel aspects of this system, report on key aspects of its performance, provide additional insights, and suggest possible future extensions.

6.1 Novel Aspects

This thesis has shown that it is possible and advantageous to develop ultrasound systems in software. Such systems harness the flexibility and extensibility of software, while maintaining the performance of traditional systems. The feasibility of software-based ultrasound was demonstrated by using a simulation environment to transmit and receive ultrasound echoes. By giving the user more control over the transmission and processing of ultrasound echoes, this system has demonstrated the potential of software-based ultrasound systems.

6.2 Performance Results

The performance of the prototype ultrasound system was evaluated by characterizing the performance of individual modules as well as the performance of the entire system in the simulation environment. These tests yield valuable information about the viability of software-based ultrasound, as well as possibilities for future improvements in such systems, such as implementing software-based B-scan technology.

6.2.1 System Performance

In order to examine the overall system performance, a rate meter module was used in between the Pulse Generator module and the Transducer module and between the Oscilloscope module and the Video Sink. This made it possible to see how fast the Pulse Generator was generating payloads and how fast these payloads were being displayed on the screen. When examining the overall system performance, with all modules connected together, it was found that the system was easily able to handle a pulse repetition frequency of 10 Hz. At this PRF, the system was using 40% of the CPU. When the PRF was set to 20, the system had a harder time keeping up. The payload rate through the system varied between 19.5 and 20 payloads per second. At this PRF, and at higher PRFs approximately 80-90% of the CPU was being used. At higher PRFs, the system throughput didn't improve above 20 payloads per second.

A PRF of 20 Hz is adequate for A-scan ultrasound, since often times such systems are used to generate only a few scans. However, an important consideration when evaluating these systems is whether or not they can keep up with the desired sampling rate. For the filtering modules, the Nyquist sampling rate is 20 MHz. Since the payloads are 4096 samples in size, in order to keep up with the sampling rate, modules must process data at a rate of one payload every 0.2 msec. As will be shown in Section 6.2.2 (and in Table 6.1), the modules in the prototype system were between one and two orders of magnitude slower than would be required for continued sample processing.

6.2.2 Module Performance

The performance of the software modules was characterized using the simulation environment described in Section 4.5. In order to calculate the performance of individual modules, without having to factor in the effects of other modules, `vslt`, a modified version of the `vslooptest` VuSystem Tcl script written by Bill Stasior in the Software Devices and Systems Group was used. With this script, it is possible to send a certain number of payloads through a filter module, and measure the time it takes to accomplish this task. This yields meaningful results about how many payloads per second the module can handle. In order to use the `vslt` script, the data that the module processes must be stored in a file. Figure 6-1 shows the generalized layout of modules used to evaluate the performance of the prototype

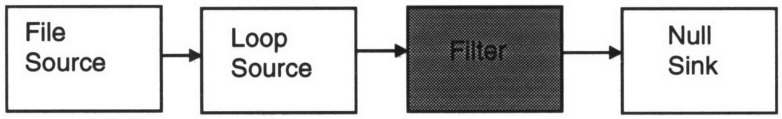


Figure 6-1: Setup used to Evaluate the Performance of Filter Modules

system filter modules.

All input payloads are 4096 samples in length. On the transmit side, with a sampling rate of 5 MHz, this corresponds to a payload duration of 0.8 msec. On the receive side, with a sampling rate of 20 MHz, this corresponds to 0.2 msec. However, since payloads are sent out and received one at a time, instead of in a continuous stream, it is possible to do some processing off-line without having to worry about real-time constraints.

For each prototype system filter module being tested, data generated by the upstream modules was written to a file. So, for example, in order to test the Receiver module, the Pulse Generator module was connected to the Transducer module, whose output was written to a File Sink module. In the `vslt` script, a File Source module reads the file in order to generate the appropriate payloads, which are then passed to the `LoopSource` module, and finally through the filter of interest into a `Null Sink` module that frees the memory associated with the payload. A separate file containing the expected input data was used to test each filter module. Below is the command entered at the command line used to generate performance data:

```
vssh vslt -numSend 100 -numPayloads 10 <source_file> -filter
<filter_file -options> >> <output_file>
```

What this does is send 100 payloads through the `filter_file`, using the first 10 payloads in the `source_file` (thus each of the first ten payloads is processed ten times by the filter).

All three filter modules (Demodulator, Receiver, and Transducer), were tested using the default parameters stated in Chapter 4. Since the Receiver has many different options, various combinations of these options were tested when evaluating the performance of the Receiver. Furthermore, the options estimated to give the worst performance, were then used to test the Demodulator. Performance data was generated by averaging the results of five trials for each set of parameters, using the above script on a Pentium Pro 200 MHz machine. In all cases, the payloads being processed are `VsSampleData` payloads, 8 KB in size. The results are summarized in Table 6.1. When parameters are listed, the remaining parameters should be assumed to have default value.

Module	Parameters	Performance
Transducer	Default	21.00 ms/payload
Receiver	Default	2.84 ms/payload
Receiver	Gain=100	3.33 ms/payload
Receiver	Gain=100, Log	6.13 ms/payload
Receiver	TCG, Delay=0.01, Slope=10	11.77 ms/payload
Receiver	TCG, Delay=0.01, Slope=10, Log	15.74 ms/payload
Receiver	Gain=100,TCG,Delay=0.01,Slope=10,Log	15.96 ms/payload
Receiver	Gain=80,TCG,Delay=0.01,Slope=10,Log	15.54 ms/payload
Receiver	TCG,Delay=0.1,Slope=10	2.88 ms/payload
Demodulator	Default	11.83 ms/payload
Demodulator	Gain=100,TCG,Delay=0.01,Slope=10,Log	12.26 ms/payload

Table 6.1: Module Performance Measurements

These numbers seem to indicate that the TCG and logarithmic functions dramatically reduce the performance of the system. This makes sense since both of these operations add additional floating point computations. The one case in which TCG didn't seem to have much of an effect on performance (when the payload evaluation period was 2.88 ms/payload) was the case in which the delay was greater than the size of the payload (in which case TCG was never applied).

Since TCG and logarithmic amplification are often not used in ophthalmic ultrasound because the eye is so small, the performance of the system is not necessarily constrained by these worst-case parameters. There are many possible ways to improve the performance of the system in order to give better results. The easiest solution is to replace floating point computations with integer ones. Floating point arithmetic was used for greater accuracy, however the results were always cast to unsigned shorts since the Oscilloscope display module handles this data type. Using unsigned shorts, or even unsigned integers, for all computations would not cause a significant decrease in accuracy, and could potentially improve the speed of the modules by a factor of four.

Performance could also be improved by using larger-sized payloads. There is some overhead associated with each payload, which tends to waste valuable processing time. If larger payloads were used, this overhead would be spread out across more samples, resulting in a smaller overhead per sample. With large payloads, however, there is the danger that a slow module might take too long to process it, resulting in back pressure that slows down the system.

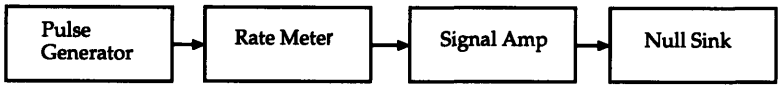


Figure 6-2: Setup used to Evaluate the Performance of the Pulse Generator Module

Demodulation is so slow because of the numerous floating point operations performed on each payload (see Section 4.4.1), which is partly due to the small slice size (8 samples) used in ML estimation. Transducers often have a higher Q than used in the simulator module (a Q of 3.9). For such transducers, the bandwidth is smaller, and thus the slice size larger than lower Q transducers. Thus, with higher quality transducers, the demodulator will perform fewer computations, which will improve the system throughput[1].

Another interesting test that was performed was to see how fast the Pulse Generator module could generate payloads in the absence of any of the ultrasound filters. The setup for this experiment is shown in Figure 6-2. Using the rate meter, and experimenting with various PRFs, it was found that the source module could generate up to 4000 payloads/sec while using approximately 96% of the CPU. The 4096 sample payloads generated were sampled at 5 MHz. This corresponds to approximately 0.8 msec of required processing time in order to keep up with the sampling rate. The measured performance of 4000 payloads/sec, corresponds to 0.25 msec of processing per payload. Thus, the Pulse Generator is easily able to keep up with the desired sampling rate, and could generate payloads sampled up to approximately 15 MHz and still keep up with the required data rate. A Pulse Generator module that can generate payloads at this rate is essential for B-scan ultrasound, in which the PRF is often around 4 kHz.

The Transducer module, which is only used for simulation purposes, is the slowest module mainly because it performs convolution operations on the data using the FFT, which involves numerous floating point operations. This decrease in performance is not too important, however, since the Transducer is just used in the simulation environment. In an actual ultrasound system, the transducer module would be replaced by connections to the necessary hardware.

[1]Although the performance of the system is not ideal for the 10 MHz received samples in ophthalmic ultrasound, other modes of ultrasound, such as cardiac and fetal, use lower frequencies, sometimes around 1 MHz, which would corresponds to a Nyquist rate of 2 MHz. The prototype system should be better able to handle the received signals of such modalities.

6.3 Performance Summary and Additional Insights

The performance of the prototype system seems to indicate that some form of software-based ultrasound is definitely possible. As processors become faster and less expensive, and extensions such as the Intel MMX technology make numerical computations faster, advanced software-based ultrasound systems will become even more viable.

While developing the software ultrasound system, I have learned some important lessons:

- Although hardware was built to interface between the software and an ultrasound probe, connecting to the probe proved to be a far greater challenge than expected. This is partly a reflection on the fact that the medical industry is often not very forthcoming with detailed information about their products. This points to the need for more widely accepted industry interface standards for the design and construction of medical instruments.

- Processing ultrasound echoes in software is not as difficult as expected, especially in the case of A-scan technology. However, in order to make the jump into B-scan imaging, either significant performance improvements need to be made or processing power needs to be increased.

6.4 Future Work

The implementation described in this thesis provides a foundation for software-based ultrasound systems. However, since this system is meant as a prototype system that demonstrates the feasibility of software ultrasound, there is much that can be done to enhance its utility in a medical setting. The most useful extension would be connecting the ultrasound software modules to the proper hardware in order to create an operating ultrasound unit. The extensions described below would incorporate advances in ultrasound technology, while taking advantage of the flexibility software has to offer. Section 6.4.1 describes ways in which the current implementation could be improved by adding features to the user interface. Section 6.4.2 describes the possibility of incorporating additional scanning modalities into a software-based system. Section 6.4.3 describes possible improvements in the signal processing algorithms in order to obtain better results. Finally, 6.4.4 describes ways in which a software-based ultrasound system would facilitate future innovation within

ultrasound technology.

6.4.1 The User Interface

An important feature of the ultrasound receiver, described in Section 4.3, is the Time Compensated Gain (TCG)) function. This function, which allows the user to set the amplification of received echoes, is either a linear or logarithmic function specified by a particular delay and slope. The user is able to change these parameters based on observing the received signal. However, he or she is constrained to a particular set of values determined by these parameters.

One possible method which gives the user far more control over how the TCG function is set is to allow him or her to set the amplification of the received echoes at various depths. In such a system of operation, the user would highlight a particular region of the display, perhaps by drawing a box around that region or clicking on the endpoints of the region, and then select a value for the amplification from a set of acceptable values. An alternative method, and one that is used in some modern ultrasound systems, is to have pre-determined regions in which the user must specify the amplification. Thus, there would be a separate amplification control for each region.

The advantage of giving the user more control over the TCG is that it allows him or her to take a closer look at particular regions of interest. For example, if the user, upon examining the returning echoes, realized that certain echoes were extremely weak, he or she could choose to increase the amplification of those regions, without needing to change the amplification of other regions. Thus, the user would not be constrained to using a linear or logarithmic function, but could instead implement pretty much any function he or she desired in order to obtain the necessary clarity to properly interpret the results.

6.4.2 Alternate Scanning Methods

A-mode scanning was used in the prototype ultrasound system because of its ease of use and utility in ophthalmic ultrasound applications. However, the most prevalent form of ultrasound scanning is the B-scan. The requirements for incorporating this modality into the software-ultrasound framework, as well as a description of multiple-array ultrasound systems are presented in the following sections.

B-Mode Scanning

As described in Section 2.2.2, B-scan images are two-dimensional images that are derived from multiple A-scan images at various angles, in which the amplitude is represented by the brightness of a pixel. In order to create this two-dimensional picture, the patient is scanned at various angles, producing a pie-shaped sector scan. In older technology, the ultrasound operator would actually have to properly scan the probe across the surface of the patient in order to produce the correct sector image. However, modern probes facilitate the process of generating a sector scan by automatically rotating the position of the scan apparatus. This rotation is accomplished by using two Linear Variable Differential Transformers (LVDT). The LVDTs operate in a push-pull manner in order to create two degrees of freedom (in the horizontal and vertical directions), allowing for angular scanning [27].

As opposed to the A-scan, in which only one control signal, namely the excitation pulse, is required for proper operation, the B-scan requires five different control signals. Each LVDT requires two control signals, while the transducer still requires an excitation signal. As parallel architectures improve and become more cost-effective, and the GuPPI and D/A converters become faster, achieving reasonable results for B-mode scanning should be feasible. With such advances, a B-mode system can be added to the prototype system with only a few modifications to the software. On the transmit end, the control signals would have to be synchronized and sent to the GuPPI. Each control signal could have its own payload type. In order to improve speed, these control signals should be buffered and only changed if the user changes any relevant parameters.

The receiver would have to take into account the fact that there are multiple scan lines in an image. One possible solution would be to store each scan line as a separate payload to be processed. These scan lines could then be assembled together into a video payload. The translation between scan line and video payload would involve translating the data values into an appropriate pixel value (or indexing into the proper color map). With the VuSystem, it would be very easy to show the individual scan lines as a changing A-mode display in a software oscilloscope display, while using the collection of scan lines that make up a sector scan as part of the video B-mode image. This way, simultaneous A- and B-mode display, a feature available on many ophthalmic ultrasound systems, would be possible.

Phased Array Ultrasound Imaging

Most ultrasound systems use either a single transducer to transmit and receive signals or one transducer for each function. However, some advanced systems use a technique known as phased array imaging to produce a more accurate representation of the area being imaged. In B-mode scanning, the transducer is rotated in the sector plane, while in phased array scanning multiple transducers are used to create the same effect. On the transmit side, each of the transducers is excited separately, creating a differential delay (phase delay) that allows the beam to be steered through different angles. Designing this part of the system in software isn't too difficult since the same excitation pulse needs to be sent out multiple times. The hardware is a bit trickier since the proper transducer must be excited by the appropriate pulse. However, using a multiplexer, this should be possible [13].

The receive end is a little more involved since the received waveforms from the transducers must be properly added together to create a coherent waveform. However, in a B-scan, this procedure needs to occur for every point of the image. Such requirements create a large demand for processing resources. However, as processor speeds increase, these techniques will become more and more feasible. A major advantage of using the VuSystem for such an application is that in virtual time, the real-time processing constraints of the system are relaxed, making it easier for the software to keep up with the processing requirements.

6.4.3 Improvements in Signal Processing

Once a software-based B-mode ultrasound system has been developed, developing a three-dimensional ultrasound system should become easier. Three-dimensional ultrasound images are created by combining two-dimensional images using certain geometric assumptions. Since all of the displayed ultrasound images will be in digital form, applying various three-dimensional imaging algorithms to ultrasound scans will not be very difficult. Furthermore, it may be possible to apply certain edge detection, computer vision, and pattern recognition algorithms to improve the quality of the images and to detect certain clinically-relevant features in real-time. Bill Stasior in the Software Devices and Systems Group at the MIT Laboratory for Computer Science has done much work on applying such algorithms to non-medical media streams. Current applications of this technology include a room monitor that records images based on whether or not a room is occupied and a whiteboard recorder that

keeps track of changes to an office whiteboard [16]. In current, hardware-based ultrasound systems, such techniques can only be performed if the analog images are digitized and then processed on a computer. However, such systems don't offer the advantage of being able to perform these signal processing tasks in real-time, which may often give clinicians valuable information that can influence what additional scans are necessary.

6.4.4 Future Applications of Software-Based Ultrasound

Although software-based ultrasound systems are inherently beneficial, such systems become even more valuable when they are combined with other technologies in order to form more sophisticated medical diagnostic and treatment tools. For example, a software-based ultrasound system can be seamlessly integrated into network-based medical consultation systems. The method in which ultrasound scans are obtained can be controlled remotely since the system is software-based. Thus, a doctor not present during the scan can make adjustments to the parameters used to collect the images, while a technician conducts the scan in a separate location. This would yield more significant clinical data since the doctor is able to shape how the results are collected, without having to physically be in the room where the scan is taking place. Since telemedicine is important for the future of medicine, a software-based ultrasound system is an important first step in realizing this possibility.

Another possible application of software-based ultrasound is as part of a data-collection system. In such a system, ultrasound images would be combined with data from various monitoring devices (e.g., EKG, respiration, blood pressure, etc.) to form one comprehensive, real-time image. Such systems are extremely useful to clinicians since they have the luxury of having all the relevant data correctly aligned in one place, thus saving them the time of having to manually put together all of the information relevant for making a particular clinical decision.

Since ultrasound is used in many non-medical applications, such as stress testing materials, a software-based system would have many applications outside the medical field. Such systems, especially portable ones implemented on laptop computers, would be useful in many remote locations where a large-scale system is often quite cumbersome. Developing advanced software-based ultrasound systems is consistent with an overall goal of developing virtual instruments in software. Although hardware systems have been the standard for customized applications, particularly in the medical field, they are quite inflexible when

systems need to be changed. By comparison, software systems, such as the one described in this report, are much more flexible and offer the potential to improve the processing and implementation of these devices.

Appendix A

Circuits

A.1 Pulse Generation

The circuit used for pulse generation is shown in Figure A-1. The input to the pulser circuit comes from the monostable, which is triggered by the output pulse generated by the Pulse Generator module. The software-generated pulse triggers the active-high transition trigger input (pin 2, which is input B, shown in Figure A-2) on the 74LS123 monostable. The active-low transition trigger input (input A which is pin 1), is set low since it is not used, while the active-low clear input (pin 3) is set high for the same reason. The width of the output pulse (pin 13) is determined by the external capacitance and resistance at pins 14 and 15. In order to generate a pulse approximately 100 ns in width, a 10 kΩ resistor and a 12 pF capacitor are connected as shown in Figure A-2 [19].

The probe may be excited by either a positive or negative voltage (as long as the difference in voltage between the two terminals of the transducer is enough to drive it). The second part of the circuit shown in Figure A-1 actually generates the negative sharp voltage pulse to the transducer [5]. When the Harris RFP8N20L N-channel logic-level FET [8] is off (the default state), the high voltage 100 pF capacitor charges up to the positive high voltage (155 V), via the charging resistor (12 kΩ).

The output of the monostable, a 5 V pulse, is used to turn the FET on. The use of a TTL device to drive a high-voltage FET is the reason why a logic-level FET is used. These FETs are fully turned on at logic-level voltages, whereas most high-voltage FETs require significantly higher gate voltages in order to switch large voltages.

When the FET is turned on by the monostable output, the left side of the capacitor

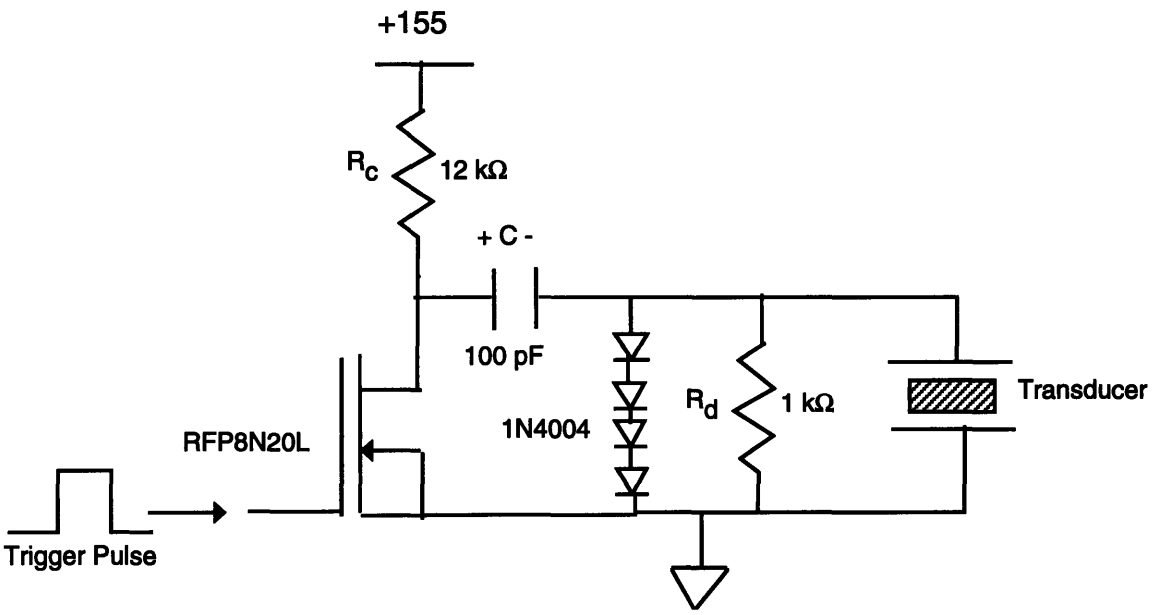


Figure A-1: Circuit Used to Excite the Transducer

is pulled to ground by the FET's source terminal. Pulling the left side of the capacitor to ground causes a discharge on the right side of the capacitor, resulting in a large negative voltage (approximately 155 V) applied to the top terminal of the transducer. Since the bottom transducer terminal is at ground, the large negative voltage is applied across the terminals of the transducer as the capacitor discharges. This idea can also be explained mathematically. The current through the capacitor is given by the following relation:

$$i_c = C\frac{dV}{dt} \tag{A.1}$$

When the FET is turned on, the voltage at the left terminal of the capacitor instantaneously drops down (i.e., there is a negative step in voltage). With a negative voltage step (V = - u_{-1} (t)), the current is proportional to an impulse (i=-C$\delta(t)$). This means that there is a negative current spike across the capacitor, allowing it to discharge instantaneously. The damping resistor (1 kΩ) across the terminals of the transducer is used to shape the trailing edge of the pulse. This resistor is set at a value large enough such that any received echoes are not driven down (as would be the case with very small resistors that would effectively short the terminals). Figure A-3 shows the negative high voltage pulse used to excite the transducer.

The four high voltage Motorola 1N4004 diodes across the terminals of the transducer are

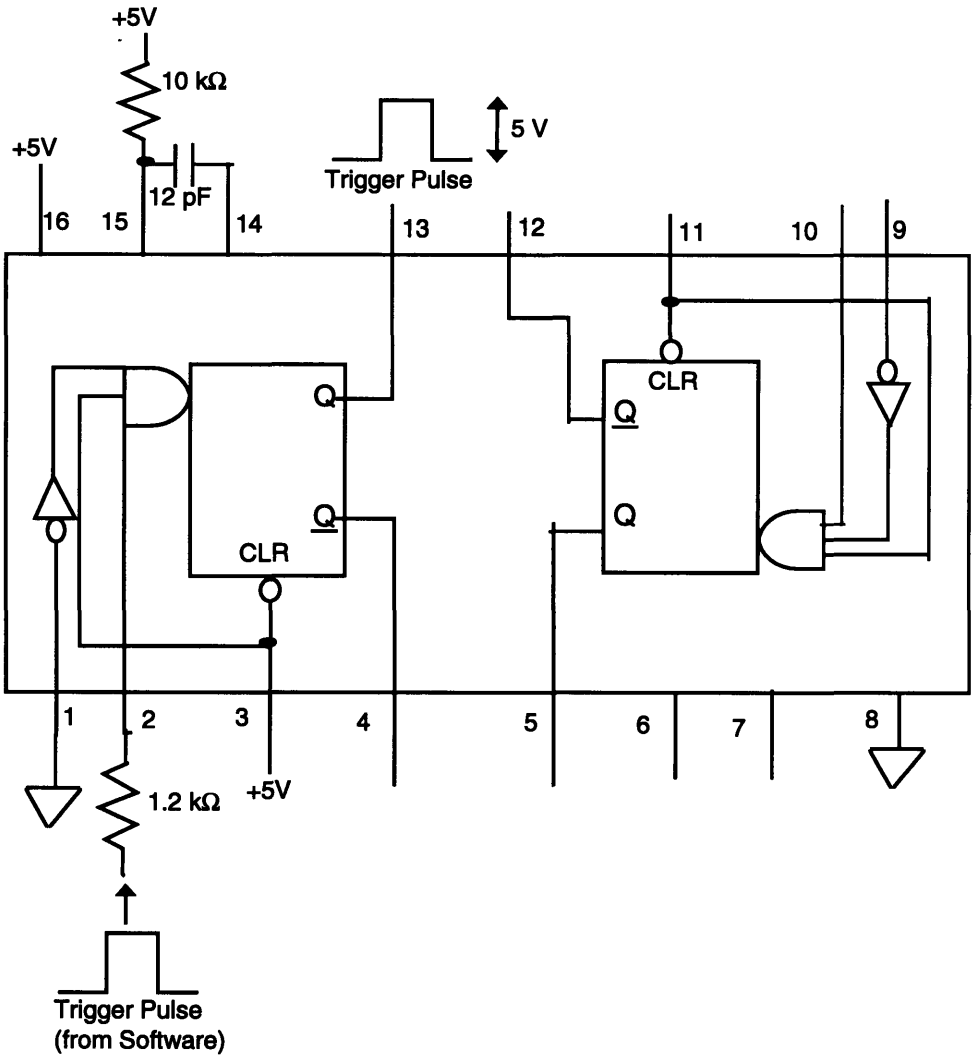


Figure A-2: Monostable Circuit used to Generate Driving Pulse

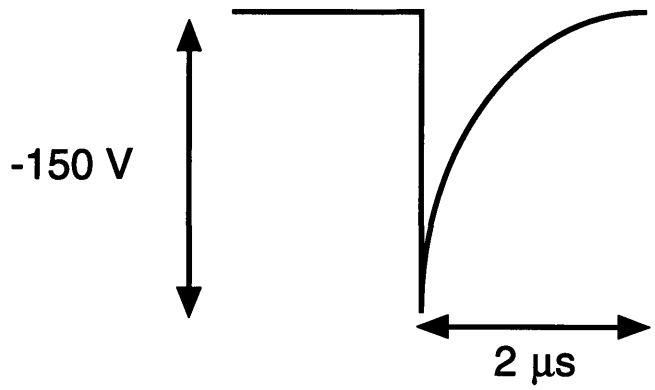


Figure A-3: Negative Excitation Pulse

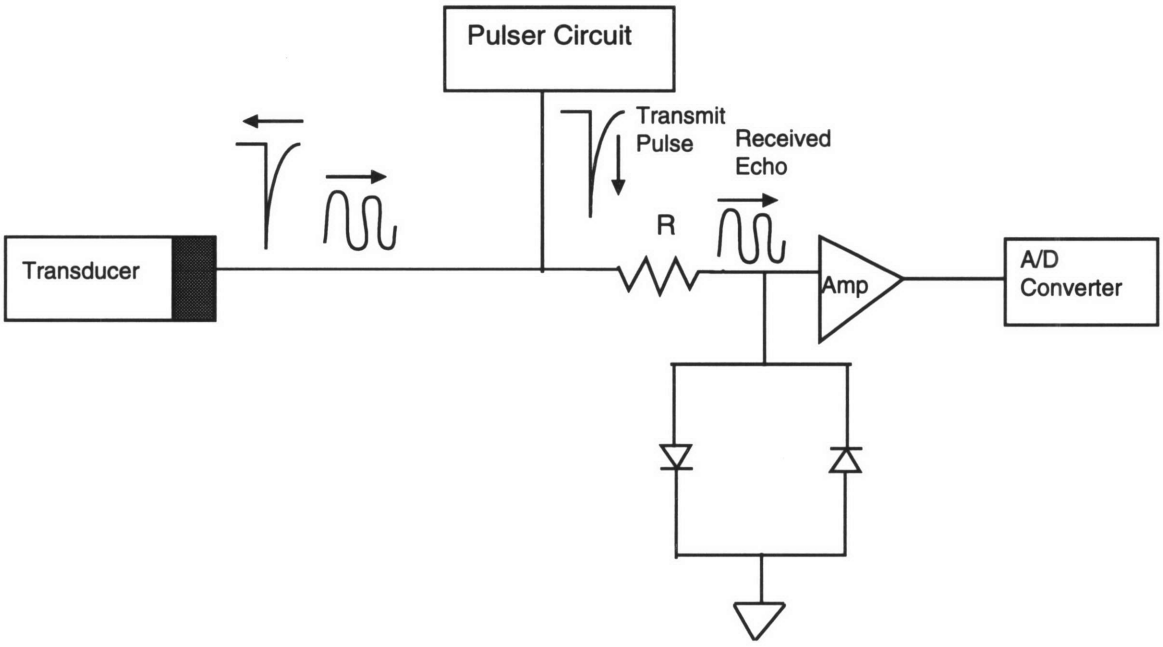


Figure A-4: An Overview of the Receiver Circuit

used to protect the transducer from the positive voltage swing that occurs when the FET turns off and the capacitor begins to charge up again (the rising edge of the pulse shown in Figure A-3). With the diodes, the voltage across the two terminals of the transducer can never go higher than four diode drops (4*0.7 V = 2.8 V) above ground. A 10 Ω resistor is connected from the gate of the FET to ground to provide a path for the gate to discharge, in order to reduce its turn-off time, and thus reduce the duration of the excitation pulse. A small resistor value is used to decrease the FET's decay time (which is also influenced by the input impedance of the FET).

A.2 Receiver

Since the input impedance of the off-output transmitter circuit is governed by the damping resistor (1 kΩ) and the impedance of the transducer can be modeled as a capacitance of value 0.0047 μF, the value of R in Figure A-4 is set to 5.6 kΩ. There is a large drop in voltage across the resistor, diminishing the need for high voltage diodes to protect the receiver. The diodes used were standard 1N914 fast switching diodes.

In order to amplify the signal to the range governed by the D/A converter (±2 V), either a high bandwidth operational amplifier or fast transistor amplifier can be used. The config-

uration of all of the external hardware required for a fully functional prototype ultrasound system is shown in Figure A-5.

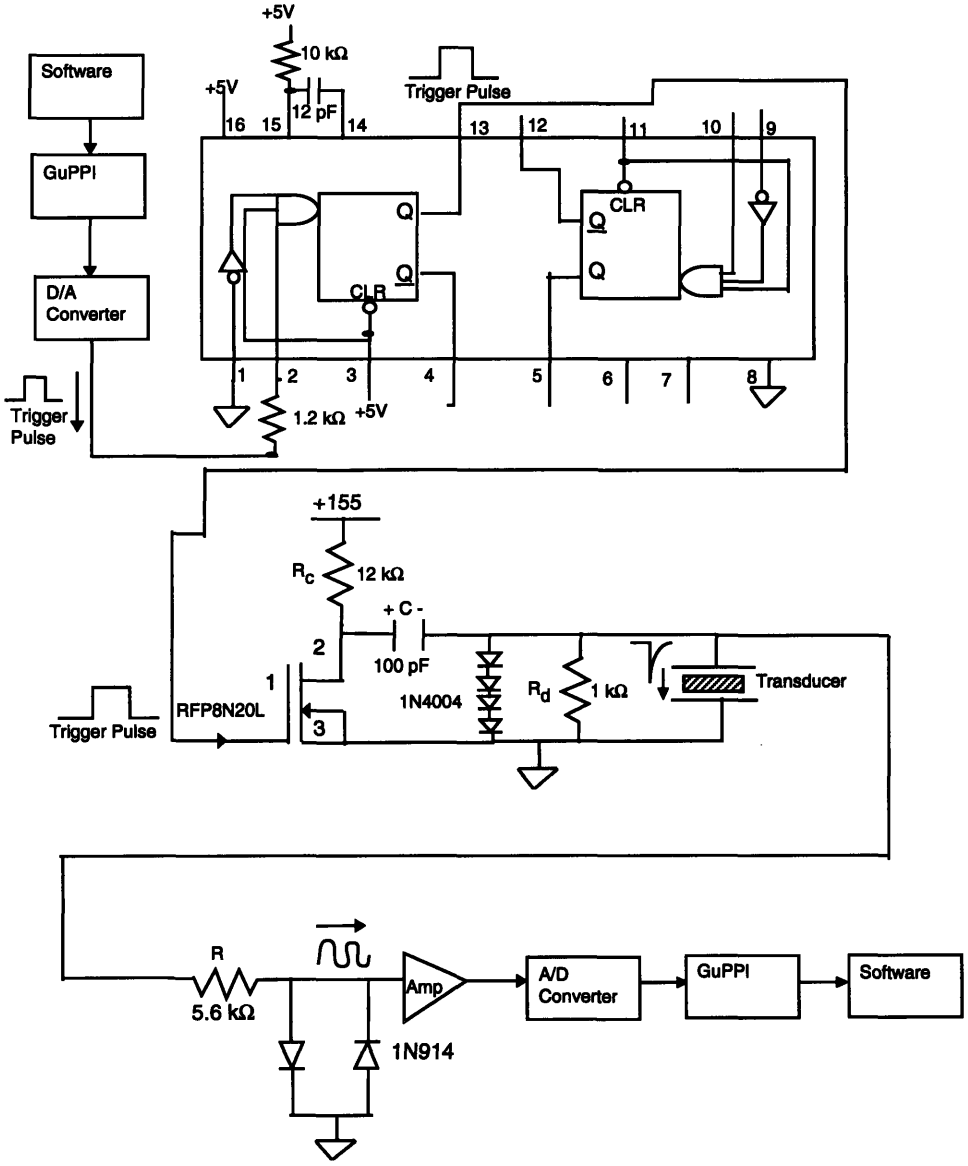


Figure A-5: Schematic of entire hardware system

Appendix B

Programming Code

B.1 Pulse Generation

B.1.1 Header File

```
#ifndef _VSULTRASOUNDCLOCK_H_
#define _VSULTRASOUNDCLOCK_H_

#ifdef __GNUG__
#pragma interface
#endif

#include <vs/vsEntity.h>
#include <vs/vsAudioFragment.h>
#include <vs/vsSampleData.h>

class VsUltrasoundClock :public VsEntity {
  VsOutputPort* outputPort;
  VsPayload* payload;
  VsTimeval timeStep;
  VsSampleData* fr;
  VsMemBlock mem;

  u_int sampleRate;
  u_int prf;
  short offset;
  u_int payloadSize;
  float duration;
  u_char amplitude;

  u_int flag;
  u_int firstCall;

  VsTimeval time;
  VsTimeval nextTime;
  VsIntervalId intervalId;

  static VsEntity* Creator(Tcl_Interp*,VsEntity*,const char*);
  static VsSymbol* classSymbol;

  friend int VsUltrasoundClockSampleRateCmd(ClientData,Tcl_Interp*,int,
    char*[]);
  friend int VsUltrasoundClockPRFCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundClockOffsetCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundClockDurationCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundClockPayloadSizeCmd(ClientData,Tcl_Interp*,
  int,char*[]);
  friend int VsUltrasoundClockAmplitudeCmd(ClientData,Tcl_Interp*,int,char*[]);
  VsUltrasoundClock(const VsUltrasoundClock&);
  VsUltrasoundClock& operator=(const VsUltrasoundClock&);
```

```
public:
  VsUltrasoundClock(Tcl_Interp*, VsEntity*, const char*);
  virtual ~VsUltrasoundClock();
  virtual VsSymbol* ClassSymbol() const { return classSymbol; };
  virtual void* ObjPtr(const VsSymbol*);
  static VsUltrasoundClock* DerivePtr(VsObj*);
  virtual void Start(Boolean);
  virtual void Stop(Boolean);
  virtual void Idle(VsOutputPort*);
  virtual void TimeOut(VsIntervalId);
  static int Get(Tcl_Interp*, char*, VsUltrasoundClock**);
  static void InitInterp(Tcl_Interp*);
};

inline VsUltrasoundClock*
VsUltrasoundClock::DerivePtr(VsObj* o) {
  return (VsUltrasoundClock*)o->ObjPtr(classSymbol);
}

inline int
VsUltrasoundClock::Get(Tcl_Interp* in, char* nm, VsUltrasoundClock** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}

#endif /* _VSULTRASOUNDCLOCK_H_ */
```

B.1.2 Main Code

```
#ifdef __GNUG__
#pragma implementation
#endif

extern "C" {
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/types.h>
}
#include <vr/vsUltrasoundClock.h>
#include <vs/vsTcl.h>
#include <vs/vsOutputPort.h>
#include <vs/vsStart.h>
#include <vs/vsFinish.h>
#include <vs/vsTclClass.h>

int
VsUltrasoundClockSampleRateCmd(ClientData cd, Tcl_Interp* in, int argc,
    char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?sampleRate?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->sampleRate) != TCL_OK)
      return TCL_ERROR;
    src->flag = 1;
  }
  src->Stop(False);
  src->Start(False);
  return VsReturnInt(in, src->sampleRate);
}

int
VsUltrasoundClockPRFCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?prf?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->prf) != TCL_OK)
```

```
      return TCL_ERROR;
    src->flag = 1;
  }
  src->Stop(False);
  src->Start(False);
  return VsReturnInt(in, src->prf);
}

int
VsUltrasoundClockOffsetCmd(ClientData cd, Tcl_Interp* in, int argc,
char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?offset?");
  if (argc > 1) {
    if (VsGetShort(in, argv[1], &src->offset) != TCL_OK)
      return TCL_ERROR;
    src->flag = 1;
  }
  return VsReturnInt(in, src->offset);
}

int
VsUltrasoundClockDurationCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?duration?");
  if (argc > 1) {
    if (VsGetFloat(in, argv[1], &src->duration) != TCL_OK)
      return TCL_ERROR;
    src->flag = 1;
  }
  return VsReturnFloat(in, src->duration);
}


int
VsUltrasoundClockPayloadSizeCmd(ClientData cd, Tcl_Interp* in, int argc,
     char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?payloadSize?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->payloadSize) != TCL_OK)
      return TCL_ERROR;
    src->flag = 1;
  }
  return VsReturnInt(in, src->payloadSize);
}


int
VsUltrasoundClockAmplitudeCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundClock* src = (VsUltrasoundClock*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?amplitude?");
  if (argc > 1) {
    if (VsGetUnsignedChar(in, argv[1], &src->amplitude) != TCL_OK)
      return TCL_ERROR;
    src->flag = 1;
  }
  return VsReturnInt(in, src->amplitude);
}


VsUltrasoundClock::
VsUltrasoundClock(Tcl_Interp* in, VsEntity* pr, const char* nm)
  :VsEntity(in,pr,nm),outputPort(new VsOutputPort(in,this,"output")),
   payload(0),timeStep(250000),
   sampleRate(5000000),prf(10),offset(0),payloadSize(4096),duration(1),
```

```
   amplitude(255), intervalId(0)
{
  firstCall = 1;
  flag = 1;
  mem.Alloc(2*payloadSize);
  CreateOptionCommand("sampleRate",
      VsUltrasoundClockSampleRateCmd, (ClientData)this,0);
  CreateOptionCommand("prf",
      VsUltrasoundClockPRFCmd, (ClientData)this,0);
  CreateOptionCommand("offset",
      VsUltrasoundClockOffsetCmd, (ClientData)this,0);
  CreateOptionCommand("duration",
      VsUltrasoundClockDurationCmd, (ClientData)this,0);
  CreateOptionCommand("payloadSize",
      VsUltrasoundClockPayloadSizeCmd, (ClientData)this,0);
  CreateOptionCommand("amplitude",
      VsUltrasoundClockAmplitudeCmd, (ClientData)this,0);

}

VsUltrasoundClock::~VsUltrasoundClock() {
  if (outputPort != 0) { delete outputPort; outputPort = 0; }
  if (intervalId != 0 || payload != 0) Stop(True);
}

void*
VsUltrasoundClock::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsEntity::ObjPtr(cl);
}

void
VsUltrasoundClock::Start(Boolean mode) {
  float temp = (float)((1000.0/prf)*1000.0);
  timeStep = VsTimeval((int)temp);
  time = VsTimeval::Now();
  nextTime = time + timeStep;
  VsEntity::Start(mode);
  if (intervalId != 0) { StopTimeOut(intervalId); intervalId = 0; }
  if (payload != 0) delete payload;
  payload = new VsStart(time, 0);
  if (outputPort->Send(payload)) payload = False;
  if (!mode) Idle(outputPort);
}

void
VsUltrasoundClock::Stop(Boolean mode) {
  VsEntity::Stop(mode);
  if (intervalId != 0) { StopTimeOut(intervalId); intervalId = 0; }
  if (payload != 0) { delete payload; payload = 0; }
  if (!mode) {
    payload = new VsFinish(VsTimeval::Now(), 0);
    if (outputPort->Send(payload)) payload = 0;
  }
}

void
VsUltrasoundClock::Idle(VsOutputPort* op) {
  if (payload != 0 && op->Send(payload)) payload = 0;
  if (payload == 0 && intervalId == 0)
    intervalId = StartTimeOut(nextTime);
}

void
VsUltrasoundClock::TimeOut(VsIntervalId id) {
  if (firstCall == 1 || flag == 1) {
    mem.Free();
    mem.Alloc(2*payloadSize);
  }

  if (intervalId == id) intervalId = 0;
  VsTimeval lowerBound = VsTimeval::Now()-VsTimeval(1000000);
  while (time < lowerBound) {
```

```
    time += timeStep;
    nextTime += timeStep;
  }

  while (payload == 0 && nextTime <= VsTimeval::Now()) {
    if (firstCall == 1 || flag == 1) {
      u_short* dst = (u_short*)mem.Ptr();
      u_short counter = 0;
      u_int pulseSamples = (u_int)((duration / 1000000) * sampleRate);
      int value;
      while (counter < pulseSamples) {
value = (int)((amplitude/(float)255)*(float)32767 + (65535/2) + offset);
if (value > 65535) value = 65535;
if (value < 0) value = 0;
*dst++ = (u_short) value;
counter++;
      }
      while (counter < payloadSize) {
*dst++ = (u_short) ((65535/2) + offset);
counter++;
      }
      firstCall = 0;
      flag = 0;
    }

    fr = new VsSampleData(time, 0, 2*payloadSize,
  sampleRate, VsShortAudioSampleEncoding,
  16, HOSTORDER, 1);
    fr->Samples() = payloadSize;
    fr->Data() = mem; fr->ComputeDuration(); payload = fr;

    time += timeStep;
    nextTime += timeStep;
    if (outputPort->Send(payload)) payload = 0;
  }
  if (payload == 0 && intervalId == 0) intervalId = StartTimeOut(nextTime);
}


VsEntity*
VsUltrasoundClock::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new VsUltrasoundClock(in, pr, nm);
}

VsSymbol* VsUltrasoundClock::classSymbol;

void
VsUltrasoundClock::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in, Creator, "VsUltrasoundClock", "VsEntity");
}
```

B.1.3 Tcl Code for Control Panel

```
VsUltrasoundClock instanceProc panel {w orient args} {
  apply Viewport $w \
    -height 600 \
    -allowVert true \
    $args

  Form $w.form

  Label $w.form.label \
    -label "Ultrasound Clock" \
    -borderWidth 0

  VsLabeledScrollbar $w.form.prf \
    -label "Pulse Repetition Frequency" \
    -value [$self prf] \
    -continuous [true] \
```

```
    -converter "vsRoundingLinearConverter 0 5000" \
    -inverter "vsLinearInverter 0 5000" \
    -callback "$self prf" \
    -width 200 \
    -fromVert $w.form.label

  VsLabeledScrollbar $w.form.duration \
    -label "Pulse Duration (us)" \
    -value [$self duration] \
    -continuous [true] \
    -converter "vsLinearConverter 0 10" \
    -inverter "vsLinearInverter 0 10" \
    -callback "$self duration" \
    -width 200 \
    -fromVert $w.form.prf

  VsLabeledScrollbar $w.form.offset \
    -label "Offset" \
    -value [$self offset] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter -32767 32767" \
    -inverter "vsLinearInverter -32767 32767" \
    -callback "$self offset" \
    -width 200 \
    -fromVert $w.form.duration


  VsLabeledScrollbar $w.form.sampleRate \
    -label "Sample Rate" \
    -value [$self sampleRate] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 1000000 30000000" \
    -inverter "vsLinearInverter 1000000 30000000" \
    -callback "$self sampleRate" \
    -width 200 \
    -fromVert $w.form.offset

  VsLabeledScrollbar $w.form.amplitude \
    -label "Amplitude" \
    -value [$self amplitude] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 255" \
    -inverter "vsLinearInverter 0 255" \
    -callback "$self amplitude" \
    -width 200 \
    -fromVert $w.form.sampleRate
}
```

B.2 Ultrasound Transducer/Target

B.2.1 Header File

```
#ifndef _VSULTRASOUNDTRANSDUCER_H_
#define _VSULTRASOUNDTRANSDUCER_H_

#ifdef __GNUG__
#pragma interface
#endif

#include <vs/vsEntity.h>
#include <vs/vsSampleData.h>
#include <vs/vsFilter.h>


class VsUltrasoundTransducer :public VsFilter {
  u_short firstCall;
  u_short tmp;
```

```
  VsMemBlock fftData, leftovers;
  float* fftData_Begin;
  unsigned long signalLength, payloadSize, size;
  static VsEntity* Creator(Tcl_Interp*,VsEntity*,const char*);
  static VsSymbol* classSymbol;
  VsUltrasoundTransducer(const VsUltrasoundTransducer&);
  VsUltrasoundTransducer& operator=(const VsUltrasoundTransducer&);

  friend int VsUltrasoundTransducerSignalLengthCmd(ClientData,Tcl_Interp*,int,
     char*[]);

protected:
  virtual Boolean WorkRequiredP(VsPayload* p);

public:
  int outputEncoding;
  VsUltrasoundTransducer(Tcl_Interp*, VsEntity*, const char*);
  virtual ~VsUltrasoundTransducer();
  virtual VsSymbol* ClassSymbol() const { return classSymbol; };
  virtual void* ObjPtr(const VsSymbol*);
  virtual Boolean Work();
  static VsUltrasoundTransducer* DerivePtr(VsObj*);
  static int Get(Tcl_Interp*, char*, VsUltrasoundTransducer**);
  static void InitInterp(Tcl_Interp*);
  void BuildFilter();
};

inline VsUltrasoundTransducer*
VsUltrasoundTransducer::DerivePtr(VsObj* o) {
  return (VsUltrasoundTransducer*)o->ObjPtr(classSymbol);
}

inline int
VsUltrasoundTransducer::Get(Tcl_Interp* in, char* nm, VsUltrasoundTransducer** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}

#endif /* _VSULTRASOUNDTRANSDUCER_H_ */
```

B.2.2 Main Code

```
#ifdef __GNUG__
#pragma implementation
#endif

extern "C" {
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
}

#include <vs/vsFilter.h>
#include <vs/vsSampleData.h>
#include <vr/vsUltrasoundTransducer.h>
#include <vs/vsOutputPort.h>
#include <vs/vsTcl.h>
#include <vs/vsTclClass.h>

#define SQ(x) ((x) * (x))
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

inline void
four1(float data[], unsigned long nn, int isign)
{
  unsigned long n,mmax,m,j,istep,i;
  double wtemp,wr,wpr,wpi,wi,theta;
  float tempr,tempi;
```

```
  n=nn << 1;
  j=1;
  for (i=1;i<n;i+=2) {
    if (j > i) {
      SWAP(data[j],data[i]);
      SWAP(data[j+1],data[i+1]);
    }
    m=n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }
  mmax=2;
  while (n > mmax) {
    istep=mmax << 1;
    theta=isign*(6.28318530717959/mmax);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1;m<mmax;m+=2) {
      for (i=m;i<=n;i+=istep) {
j=i+mmax;
tempr=wr*data[j]-wi*data[j+1];
tempi=wr*data[j+1]+wi*data[j];
data[j]=data[i]-tempr;
data[j+1]=data[i+1]-tempi;
data[i] += tempr;
data[i+1] += tempi;
      }
      wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
}



inline void
realft(float data[], unsigned long n, int isign)
{
  unsigned long i,i1,i2,i3,i4,np3;
  float c1=0.5,c2,h1r,h1i,h2r,h2i;
  double wr,wi,wpr,wpi,wtemp,theta;

  theta=3.141592653589793/(double) (n>>1);
  if (isign == 1) {
    c2 = -0.5;
    four1(data,n>>1,1);
  } else {
    c2=0.5;
    theta = -theta;
  }
  wtemp=sin(0.5*theta);
  wpr = -2.0*wtemp*wtemp;
  wpi=sin(theta);
  wr=1.0+wpr;
  wi=wpi;
  np3=n+3;
  for (i=2;i<=(n>>2);i++) {
    i4=1+(i3=np3-(i2=1+(i1=i+i-1)));
    h1r=c1*(data[i1]+data[i3]);
    h1i=c1*(data[i2]-data[i4]);
    h2r = -c2*(data[i2]+data[i4]);
    h2i=c2*(data[i1]-data[i3]);
    data[i1]=h1r+wr*h2r-wi*h2i;
    data[i2]=h1i+wr*h2i+wi*h2r;
    data[i3]=h1r-wr*h2r+wi*h2i;
```

```
    data[i4] = -h1i+wr*h2i+wi*h2r;
    wr=(wtemp=wr)*wpr-wi*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
  }
  if (isign == 1) {
    data[1] = (h1r=data[1])+data[2];
    data[2] = h1r-data[2];
  } else {
    data[1]=c1*((h1r=data[1])+data[2]);
    data[2]=c1*(h1r-data[2]);
    four1(data,n>>1,-1);
  }
}

int
VsUltrasoundTransducerSignalLengthCmd(ClientData cd, Tcl_Interp* in, int argc,
char* argv[])
{
  VsUltrasoundTransducer* lpf = (VsUltrasoundTransducer*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?signalLength?");
  if (argc > 1)
    {
      if (VsGetUnsignedLong(in, argv[1], &lpf->signalLength) != TCL_OK)
return TCL_ERROR;
      lpf->BuildFilter();
    }
  return VsReturnInt(in, lpf->signalLength);
}

VsUltrasoundTransducer::VsUltrasoundTransducer(Tcl_Interp* in, VsEntity* pr, const char* nm)
     :VsFilter(in,pr,nm),signalLength(1024)
{
  fftData.Alloc(sizeof(float)*signalLength);
  leftovers.Alloc(sizeof(float)*signalLength);
  payloadSize = 0;
  firstCall = 1;
  tmp = 1;
  CreateOptionCommand("signalLength",
      VsUltrasoundTransducerSignalLengthCmd, (ClientData)this, 0);
}


VsUltrasoundTransducer::~VsUltrasoundTransducer() {
}

void*
VsUltrasoundTransducer::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsFilter::ObjPtr(cl);
}

void
VsUltrasoundTransducer::BuildFilter()
{
  VsSampleData* frag = VsSampleData::DerivePtr(payload);
  fftData.Free();
  payloadSize = frag->Samples();
  size = (int) pow(2,(int)ceil((log(payloadSize+signalLength-1)/log(2))));
  fftData.Alloc(sizeof(float)*size);
  float* fft_ptr = (float *) fftData.Ptr();

  for (u_int n = 0; n < signalLength; n++) {
    *(fft_ptr+n) = ((-3.33E6*3.71E3)/(6.66E6-3.71E3))*
      exp(-3.71E3*n*((float)1/(float)frag->SamplesPerSecond())) +
      ((-6.66E6*3.33E6)/(3.71E3-6.66E6))*
      exp(-6.66E6*n*((float)1/(float)frag->SamplesPerSecond()));
  }
  for (u_int i = signalLength; i < size; i++)
    *(fft_ptr+i) = 0.0;
  realft(fft_ptr-1, size, 1);
}

Boolean
```

```
VsUltrasoundTransducer::WorkRequiredP(VsPayload *p) {
  VsSampleData* frag = VsSampleData::DerivePtr(p);
  return (frag != 0);
}

Boolean
VsUltrasoundTransducer::Work()
{
  VsSampleData* frag = VsSampleData::DerivePtr(payload);
  if (payloadSize != (u_int)frag->Samples()) {
    payloadSize = (u_int)frag->Samples();
    BuildFilter();
  }
  float src_data[size];
  u_short *src = (u_short *)(frag->Data().Ptr());
  for (u_int i = 0; i < (u_int)frag->Samples(); i++) {
    src_data[i] = (float) *(src+i);
  }
  float *fft_ptr = (float *) fftData.Ptr();
  for (u_int i = payloadSize; i < size; i++)
    src_data[i] = 0.0;
  realft(src_data - 1, size, 1);
  for (int i = 0; i < 2; i++)
    src_data[i] *= fft_ptr[i];
  for (u_int i = 2; i < size-1; i+=2) {
    float tempi, tempr;
    tempr = (src_data[i] * fft_ptr[i]) - (src_data[i+1]*fft_ptr[i+1]);
    tempi = (src_data[i+1]*fft_ptr[i]) + (src_data[i]*fft_ptr[i+1]);
    src_data[i] = tempr;
    src_data[i+1] = tempi;
  }
  realft(src_data-1, size, -1);
  if (firstCall == 1) {
    leftovers.Free();
    firstCall = 0;
  }
  else {
    float *leftover_ptr = (float *) leftovers.Ptr();
    for (u_int i = 0; i < (size-payloadSize); i++)
      src_data[i] += *(leftover_ptr+i);
    leftovers.Free();
  }
  leftovers.Alloc(sizeof(float)*(size-payloadSize));
  float *leftover_ptr = (float *) leftovers.Ptr();
  for (u_int i = payloadSize, j = 0; i < size; i++,j++)
    *(leftover_ptr+j) = src_data[i];

  u_short ushortBytes = sizeof(u_short);
  u_short ushortBits = 8 * ushortBytes;
  u_short maxushort = (((u_short) pow(2,ushortBits - 1) - 1) * 2) + 1;
  u_short zero = maxushort/2;
  float maxfloat = 5E16;

  for (u_int i = 0; i < (u_int)frag->Samples(); i++) {
    src_data[i] /= maxfloat;
    if (src_data[i] > 1)
      src_data[i] = 1;
    if (src_data[i] < -1)
      src_data[i] = -1;
    src_data[i] = zero + (src_data[i] * zero);
    src_data[i] = src_data[i] * maxushort;
    *(src+i) = (u_short) src_data[i];
  }
  tmp = 0;
  return VsFilter::Work();
}

VsEntity*
VsUltrasoundTransducer::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new VsUltrasoundTransducer(in, pr, nm);
}
```

```
VsSymbol* VsUltrasoundTransducer::classSymbol;

void
VsUltrasoundTransducer::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in, Creator, "VsUltrasoundTransducer", "VsFilter");
}
```

B.2.3 Tcl Code for Control Panel

```
VsUltrasoundTransducer instanceProc panel {w orient args} {
  apply Viewport $w \
    -height 200 \
    -allowVert true \
    $args

  Form $w.form

  Label $w.form.label \
    -label "Ultrasound Transducer" \
    -borderWidth 0
}
```

B.3 Ultrasound Receiver

B.3.1 Header File

```
#ifndef _VSULTRASOUNDRECEIVER_H_
#define _VSULTRASOUNDRECEIVER_H_

#ifdef __GNUG__
#pragma interface
#endif

#include <vs/vsEntity.h>
#include <vs/vsSampleData.h>
#include <vs/vsFilter.h>


class VsUltrasoundReceiver :public VsFilter {
  static VsEntity* Creator(Tcl_Interp*,VsEntity*,const char*);
  static VsSymbol* classSymbol;

  u_int initGain;
  u_int maxGain;
  u_char function;
  float delay;
  u_int slope;
  u_char tcg;

  friend int VsUltrasoundReceiverInitialGainCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundReceiverMaximumGainCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundReceiverFunctionCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundReceiverDelayCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundReceiverSlopeCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsUltrasoundReceiverTcgCmd(ClientData,Tcl_Interp*,int,char*[]);
  VsUltrasoundReceiver(const VsUltrasoundReceiver&);
  VsUltrasoundReceiver& operator=(const VsUltrasoundReceiver&);

protected:
  virtual Boolean WorkRequiredP(VsPayload* p);

public:
```

```
  int outputEncoding;
  VsUltrasoundReceiver(Tcl_Interp*, VsEntity*, const char*);
  virtual ~VsUltrasoundReceiver();
  virtual VsSymbol* ClassSymbol() const { return classSymbol; };
  virtual void* ObjPtr(const VsSymbol*);
  virtual Boolean Work();
  static VsUltrasoundReceiver* DerivePtr(VsObj*);
  static int Get(Tcl_Interp*, char*, VsUltrasoundReceiver**);
  static void InitInterp(Tcl_Interp*);
};

inline VsUltrasoundReceiver*
VsUltrasoundReceiver::DerivePtr(VsObj* o) {
  return (VsUltrasoundReceiver*)o->ObjPtr(classSymbol);
}

inline int
VsUltrasoundReceiver::Get(Tcl_Interp* in, char* nm, VsUltrasoundReceiver** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}

#endif /* _VSULTRASOUNDRECEIVER_H_ */
```

B.3.2 Main Code

```
#ifdef __GNUG__
#pragma implementation
#endif

extern "C" {
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
}

#include <vs/vsFilter.h>
#include <vs/vsSampleData.h>
#include <vr/vsUltrasoundReceiver.h>
#include <vs/vsOutputPort.h>
#include <vs/vsTcl.h>
#include <vs/vsTclClass.h>

#define SPEED_OF_SOUND pow(1.54,5)

int
VsUltrasoundReceiverInitialGainCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?initGain?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->initGain) != TCL_OK)
      return TCL_ERROR;
  }
  return VsReturnInt(in, src->initGain);
}


int
VsUltrasoundReceiverMaximumGainCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?maxGain?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->maxGain) != TCL_OK)
      return TCL_ERROR;
  }
```

```
  return VsReturnInt(in, src->maxGain);
}


int
VsUltrasoundReceiverFunctionCmd(ClientData cd, Tcl_Interp* in, int argc,
  char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?function?");
  if (argc > 1) {
    if (VsGetUnsignedChar(in, argv[1], &src->function) != TCL_OK)
      return TCL_ERROR;
  }
  return VsReturnInt(in, src->function);
}


int
VsUltrasoundReceiverDelayCmd(ClientData cd, Tcl_Interp* in, int argc,
  char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?delay?");
  if (argc > 1) {
    if (VsGetFloat(in, argv[1], &src->delay) != TCL_OK)
      return TCL_ERROR;
  }
  return VsReturnFloat(in, src->delay);
}


int
VsUltrasoundReceiverSlopeCmd(ClientData cd, Tcl_Interp* in, int argc,
   char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?slope?");
  if (argc > 1) {
    if (VsGetUnsignedInt(in, argv[1], &src->slope) != TCL_OK)
      return TCL_ERROR;
  }
  return VsReturnInt(in, src->slope);
}


int
VsUltrasoundReceiverTcgCmd(ClientData cd, Tcl_Interp* in, int argc,
  char* argv[])
{
  VsUltrasoundReceiver* src = (VsUltrasoundReceiver*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?tcg?");
  if (argc > 1) {
    if (VsGetUnsignedChar(in, argv[1], &src->tcg) != TCL_OK)
      return TCL_ERROR;
  }
  return VsReturnInt(in, src->tcg);
}


VsUltrasoundReceiver::VsUltrasoundReceiver(Tcl_Interp* in, VsEntity* pr, const char* nm)
     :VsFilter(in,pr,nm),initGain(40), maxGain(140),
      function(0), delay(1), slope (1), tcg(0)
{
  CreateOptionCommand("initGain",
      VsUltrasoundReceiverInitialGainCmd, (ClientData)this, 0);
  CreateOptionCommand("maxGain",
      VsUltrasoundReceiverMaximumGainCmd, (ClientData)this, 0);
  CreateOptionCommand("function",
      VsUltrasoundReceiverFunctionCmd, (ClientData)this,0);
  CreateOptionCommand("delay",
      VsUltrasoundReceiverDelayCmd, (ClientData)this,0);
```

```
  CreateOptionCommand("slope",
      VsUltrasoundReceiverSlopeCmd, (ClientData)this,0);
  CreateOptionCommand("tcg",
      VsUltrasoundReceiverTcgCmd, (ClientData)this,0);
}


VsUltrasoundReceiver::~VsUltrasoundReceiver() {
}

void*
VsUltrasoundReceiver::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsFilter::ObjPtr(cl);
}

Boolean
VsUltrasoundReceiver::WorkRequiredP(VsPayload *p) {
  VsSampleData* frag = VsSampleData::DerivePtr(p);
  return (frag != 0);
}

Boolean
VsUltrasoundReceiver::Work()
{
  VsSampleData* frag = VsSampleData::DerivePtr(payload);
  u_short* src_data = (u_short *)(frag->Data().Ptr());
  u_int samples = frag->Samples();
  frag->SamplesPerSecond() = 20000000;
  u_int sampleRate = frag->SamplesPerSecond();
  float sampleDelay_temp = ((float) delay / (SPEED_OF_SOUND*10)) * sampleRate;
  u_int sampleDelay = (u_int) sampleDelay_temp;
  float initAmp = pow(10,((float)initGain/20));
  float maxAmp =  pow(10,((float)maxGain/20));
  u_short ushortBytes = sizeof(u_short);
  u_short ushortBits = 8 * ushortBytes;
  u_short maxushort = (((u_short) pow(2,ushortBits - 1) - 1) * 2) + 1;
  float maxfloat = 6.5535E9;
  float logmaxfloat = 4.8164733E5;
  for (u_int i = 0; i < samples; i++) {
    float temp = (float) *src_data;
    if (i < sampleDelay || tcg == 0) {
      if (function == 0) {
if (initAmp > maxAmp)
  initAmp = maxAmp;
  int value = 32767+*src_data;
  if (value > 65535) value = 65535;
  if (value < 0) value = 0;
  *src_data = value;
} */
temp *= initAmp;
temp /= maxfloat;
if (temp < 1)
  {}
else
  temp = 1;
temp *= maxushort;
*src_data = (u_short) temp;
if (*src_data > 65535) *src_data = 65535;
src_data++;
      }
      else {
if (initAmp > maxAmp)
  initAmp = maxAmp;
if (*src_data == 0)
  {}
else {
  temp = log10(temp) * initAmp;
  temp /= logmaxfloat;
  if (temp < 1)
    {}
  else
    temp = 1;
```

```
  temp *= maxushort;
  *src_data = (u_short) temp;
  if (*src_data > 65535) *src_data = 65535;
}
src_data++;
      }
    }
    else {
      float slopeAmp = pow(10,((float)(((i - sampleDelay) *
       ((float)(1/(float)sampleRate)) *
       pow(10,3) * slope)/20)));
      if (function == 0) {
if ((initAmp*slopeAmp)>maxAmp) {
      int value = 32767+*src_data;
      if (value > 65535) value = 65535;
      if (value < 0) value = 0;
    *src_data = value;
  }
  temp *= maxAmp;
  temp /= maxfloat;
  if (temp < 1)
    {}
  else
    temp = 1;
  temp *= maxushort;
  *src_data = (u_short) temp;
  if (*src_data > 65535) *src_data = 65535;
  src_data++;
}
else {
  temp *= (initAmp*slopeAmp);
  temp /= maxfloat;
  if (temp < 1)
    {}
  else
    temp = 1;
  temp *= maxushort;
  *src_data = (u_short) temp;
  if (*src_data > 65535) *src_data = 65535;
  src_data++;
}
      }
      else {
if ((initAmp*slopeAmp)>maxAmp) {
  if (*src_data == 0)
    {}
  else {
    temp = log10(temp) * maxAmp;
    temp /= logmaxfloat;
    if (temp < 1)
      {}
    else
      temp = 1;
    temp *= maxushort;
    *src_data = (u_short) temp;
    if (*src_data > 65535) *src_data = 65535;
  }
  src_data++;
}
else {
  if (*src_data == 0)
    {}
  else {
    temp = log10(temp) * (initAmp * slopeAmp);
    temp /= logmaxfloat;
    if (temp < 1)
      {}
    else
      temp = 1;
    temp *= maxushort;
    *src_data = (u_short) temp;
    if (*src_data > 65535) *src_data = 65535;
```

```
  }
  src_data++;
}
      }
    }
  }
  return VsFilter::Work();
}

VsEntity*
VsUltrasoundReceiver::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new VsUltrasoundReceiver(in, pr, nm);
}

VsSymbol* VsUltrasoundReceiver::classSymbol;

void
VsUltrasoundReceiver::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in, Creator, "VsUltrasoundReceiver", "VsFilter");
}
```

B.3.3 Tcl Code for Control Panel

```
VsUltrasoundReceiver instanceProc panel {w orient args} {
  apply Viewport $w \
    -height 400 \
    -allowVert true \
    $args

  Form $w.form

  Label $w.form.label \
    -label "Ultrasound Receiver" \
    -borderWidth 0

  VsLabeledChoice $w.form.function \
    -label "Function" \
    -choices {{0 "Linear"} {1 "Log"}} \
    -value [$self function] \
    -callback "$self function" \
    -width 200 \
    -fromVert $w.form.label


  VsLabeledScrollbar $w.form.initGain \
    -label "Initial Gain (dB)" \
    -value [$self initGain] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 100" \
    -inverter "vsLinearInverter 0 100" \
    -callback "$self initGain" \
    -width 200 \
    -fromVert $w.form.function


  VsLabeledScrollbar $w.form.maxGain \
    -label "Maximum Gain (dB)" \
    -value [$self maxGain] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 120" \
    -inverter "vsLinearInverter 0 120" \
    -callback "$self maxGain" \
    -width 200 \
    -fromVert $w.form.initGain


  VsLabeledScrollbar $w.form.delay \
    -label "Delay (mm)" \
```

```
    -value [$self delay] \
    -continuous [true] \
    -converter "vsLinearConverter 0 5" \
    -inverter "vsLinearInverter 0 5" \
    -callback "$self delay" \
    -width 200 \
    -fromVert $w.form.maxGain

  VsLabeledScrollbar $w.form.slope \
    -label "Slope (dB/ms)" \
    -value [$self slope] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 20" \
    -inverter "vsLinearInverter 0 20" \
    -callback "$self slope" \
    -width 200 \
    -fromVert $w.form.delay

  VsLabeledChoice $w.form.tcg \
    -label "TCG" \
    -choices {{0 "Off"} {1 "On"}} \
    -value [$self tcg] \
    -callback "$self tcg" \
    -fromVert $w.form.slope

}
```

B.4 Ultrasound Demodulator

B.4.1 Header File

```
#ifndef _VSULTRASOUNDDEMODULATOR_H_
#define _VSULTRASOUNDDEMODULATOR_H_

#ifdef __GNUG__
#pragma interface
#endif

#include <vs/vsEntity.h>
#include <vs/vsAudioFragment.h>
#include <vs/vsFilter.h>

#define TWOPI 6.28318530717959

extern "C" {
  extern int VsUltrasoundDemodulatorChannelCmd(ClientData,Tcl_Interp*,int,char*[]);
}

class VsUltrasoundDemodulator :public VsFilter {
  static VsEntity* Creator(Tcl_Interp*,VsEntity*,const char*);
  static VsSymbol* classSymbol;
  VsMemBlock leftovers;
  u_short firstcall;
  int samplingFrequency;
  int frequency;
  float carrier;
  float q;


  VsUltrasoundDemodulator(const VsUltrasoundDemodulator&);
  VsUltrasoundDemodulator& operator=(const VsUltrasoundDemodulator&);
protected:
  virtual Boolean WorkRequiredP(VsPayload* p);
public:
  VsUltrasoundDemodulator(Tcl_Interp*, VsEntity*, const char*);
```

```
  virtual ~VsUltrasoundDemodulator();
  virtual VsSymbol* ClassSymbol() const { return classSymbol; };
  virtual void* ObjPtr(const VsSymbol*);
  virtual void Start(Boolean);
  virtual Boolean Work();
  static VsUltrasoundDemodulator* DerivePtr(VsObj*);
  static int Get(Tcl_Interp*, char*, VsUltrasoundDemodulator**);
  static void InitInterp(Tcl_Interp*);
};

inline VsUltrasoundDemodulator*
VsUltrasoundDemodulator::DerivePtr(VsObj* o) {
  return (VsUltrasoundDemodulator*)o->ObjPtr(classSymbol);
}

inline int
VsUltrasoundDemodulator::Get(Tcl_Interp* in, char* nm, VsUltrasoundDemodulator** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}

#endif /* _VSULTRASOUNDDEMODULATOR_H_ */
```

B.4.2 Main Code

```
#ifdef __GNUG__
#pragma implementation
#endif

extern "C" {
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
}

#include <vs/vsFilter.h>
#include <vs/vsAudioFragment.h>
#include <vs/vsSampleData.h>
#include <vr/vsUltrasoundDemodulator.h>
#include <vs/vsTclClass.h>


VsUltrasoundDemodulator::VsUltrasoundDemodulator(Tcl_Interp* in, VsEntity* pr, const char* nm)
     :VsFilter(in,pr,nm),samplingFrequency(20000000), frequency(10000000),
      carrier(TWOPI*(float)frequency/(float)samplingFrequency), q(3.9)
{
  leftovers.Alloc(sizeof(short)*8193);
  *(leftovers.Ptr()) = 0;
  firstcall = 1;
}

VsUltrasoundDemodulator::~VsUltrasoundDemodulator() {
}

void*
VsUltrasoundDemodulator::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsFilter::ObjPtr(cl);
}

void
VsUltrasoundDemodulator::Start(Boolean mode) {
  VsFilter::Start(mode);
}

Boolean
VsUltrasoundDemodulator::WorkRequiredP(VsPayload *p) {
  VsSampleData* frag = VsSampleData::DerivePtr(p);
  return (frag != 0 && frag->Encoding() == VsShortAudioSampleEncoding);
```

```
}

Boolean
VsUltrasoundDemodulator::Work() {
  VsSampleData* frag = VsSampleData::DerivePtr(payload);
  u_short* remaining = (u_short*)leftovers.Ptr();
  u_short limit = *remaining;
  samplingFrequency = frag->SamplesPerSecond();
  u_short len = frag->Samples();
  u_short* data = (u_short *)(frag->Data().Ptr());
  float alpha = (float)frequency / (2*q);
  float sps = 2*alpha;
  double x = ceil(samplingFrequency/sps);
  u_int slice = (u_int) x;
  u_int subtract_length = ((len + limit)%slice)*slice;
  size_t bar = frag->Data().Fore() + (sizeof(short)*limit)
    - (sizeof(short)*subtract_length);
  VsSampleData* output_frag =
    new VsSampleData(frag->StartingTime(), frag->Channel(),
     bar,(u_int)(ceil(sps)),VsShortAudioSampleEncoding,
     16, HOSTORDER, frag->Channels());
  u_short* outputData = (u_short *)(output_frag->Data().Ptr());
  float sum = 0.0;
  u_short numsamps = (u_short)bar/sizeof(u_short);
  u_short counter = numsamps;
  for (u_short n = 0; n < numsamps; n++)
    *(outputData+n) = 0;
  float carrier = (TWOPI*(float)frequency) /(float)samplingFrequency;

  u_int total = len+limit;
  if (total < slice) {
    delete payload;
    firstcall = 1;
  } else {
    if (firstcall == 1 || limit == 0) {
      while(len >= slice) {
float numerator = 0;
float denominator = 0;
for (u_int i=0; i < slice; i++) {
  numerator = numerator + (float)(*data) * (float)sin(carrier*i+
      M_PI/4);
  denominator = denominator + (float)(*data) * (float)cos(carrier*i+
  M_PI/4);
  data++;
}
float amp = (2.0/(float)slice) * (float)sqrt(pow(denominator,2) +
   pow(numerator,2));
if (amp > 65535.0) amp = 65535.0;
*outputData = (u_short) amp;
sum += (float)*outputData;

if ((*outputData)==0)
  numsamps--;
outputData++;
len = len - slice;
firstcall = 0;
      } }
    else {
      if (limit >= slice) {
remaining++;
while(limit >= slice) {
  float numerator = 0;
  float denominator = 0;
  for (u_int i=0; i < slice; i++) {
    numerator = numerator + (float)(*remaining) *
      (float)sin(carrier*i);
    denominator = denominator + (float)(*remaining) *
      (float)cos(carrier*i);
    remaining++;
  }
  float amp = (2.0/(float)slice) + (float)sqrt(pow(denominator,2) +
```

```
    pow(numerator,2));
  if (amp > 65535.0) amp = 65535.0;
  *outputData = (u_short) amp;
  outputData++;
  limit = limit - slice;
}}
      float numerator = 0;
      float denominator = 0;
      remaining++;
      for (u_short i = 0; i < limit; i++) {
numerator = numerator + (float)(*remaining) * (float)sin(carrier*i);
denominator = denominator + (float)(*remaining) *
  (float)cos(carrier*i);
remaining++;
      }
      for (u_short i = limit; i < slice; i++) {
numerator = numerator + (float)(*data) * (float)sin(carrier*i);
denominator = denominator + (float)(*data) * (float)cos(carrier*i);
data++;
      }
      float amp = (2.0/(float)slice) + (float)sqrt(pow(denominator,2) +
 pow(numerator,2));
      if (amp > 65535.0) amp = 65535.0;
      *outputData = (u_short) amp;
      outputData++;
      len = len - (slice-limit);
      while(len >= slice) {
float numerator = 0;
float denominator = 0;
for (u_int i=0; i < slice; i++) {
  numerator = numerator + (float)(*data) * (float)sin(carrier*i);
  denominator = denominator + (float)(*data) * (float)cos(carrier*i);
  data++;
}
float amp = (2.0/(float)slice) + (float)sqrt(pow(denominator,2) +
   pow(numerator,2));
if (amp > 65535.0) amp = 65535.0;
*outputData = (u_short) amp;
outputData++;
len = len - slice;
      }}
    u_short* leftover_data = (u_short*)leftovers.Ptr();
    *leftover_data = len;
    leftover_data++;
    for (u_short i = 0; i < len; i++) {
      *leftover_data = *data;
      data++;
      leftover_data++;
    }
    float average = sum / (float)numsamps;
    float threshold = 3.0*(float)average;
    u_short* temp = (u_short *)(output_frag->Data().Ptr());
    for (u_short i=0; i < counter; i++) {
      if (*(temp+i)<(u_short)threshold)
*(temp+i) = 0;
    }
    output_frag->ComputeDuration();
    delete payload;
    payload = output_frag;
  }
  return VsFilter::Work();
}

VsEntity*
VsUltrasoundDemodulator::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new VsUltrasoundDemodulator(in, pr, nm);
}

VsSymbol* VsUltrasoundDemodulator::classSymbol;

void
VsUltrasoundDemodulator::InitInterp(Tcl_Interp* in) {
```

```
  classSymbol = InitClass(in, Creator, "VsUltrasoundDemodulator", "VsFilter");
}
```

B.4.3 Tcl Code for Control Panel

```
VsUltrasoundDemodulator instanceProc panel {w orient args} {

apply Form $w \
  $args

Label $w.label \
  -label "Ultrasound Demodulator" \
  -borderWidth 0

}
```

B.5 Tcl Script for Simulation Environment

```
proc VsOscope {w m args} {
  set scale [keyarg -scale $args [vsDefault -scale]]
  set info [keyarg -info $args 0]
  set delay [keyarg -sinkDelay $args [keyarg -delay $args 0.5]]
  set bufferDepth \
    [keyarg -sinkBufferDepth $args [keyarg -bufferDepth $args $delay]]
  set margs [keyarg -margs $args]
  set args [keyargs {-scale -delay -bufferDepth -margs} $args exclude]

  global onColor
  global offColor

  apply Form $w \
    $args
  VsScreen $w.screen \
    -scale $scale \
    -resizable true

  if {[debug] & 32} {
    Command $w.report \
      -label "Report" \
      -callback "vs report" \
      -fromVert $w.screen \
      -fromHoriz $w.visualPanel
  }

  VsEntity $m
  $m set w $w
  $m proc sourceCallback {args} {
    set sourceEnd [keyarg -sourceEnd $args 0]
    if $sourceEnd then {$self stop}
  }
  $m proc sinkCallback {args} {
    set sinkStop [keyarg -sinkStop $args 0]
    if $sinkStop then { catch {vs destory}; exit }
  }

  apply VsUltrasoundClock $m.source \
    -callback "$m sourceCallback" \
    [keyargs {-sampleRate -prf -offset -duration
      -amplitude} $margs]
  set currentOutput $m.source.output

  VsRateMeter $m.meter \
    -payload "VsSampleData" \
```

```
  -input "bind $currentOutput"
set currentOutput $m.meter.output

VsUltrasoundTransducer $m.transducer \
  -input "bind $currentOutput"
set currentOutput $m.transducer.output

apply VsUltrasoundReceiver $m.receiver \
  -input "bind $currentOutput" \
  [keyargs {-initGain -maxGain -function -delay -slope -tcg} $margs]
set currentOutput $m.receiver.output

 VsUltrasoundDemodulator $m.demodulator \
   -input "bind $currentOutput"
 set currentOutput $m.demodulator.output

apply VsSignalSource $m.sigsource \
  -callback "$m sourceCallback" \
  [keyargs {-sampleRate -frequency -amplitude
    -dutyCycle -waveform -offset -payloadSize} $margs]
set secondOutput $m.sigsource.output

apply VsOscilloscope $m.oscilloscope \
  -input1 "bind $currentOutput" \
  -input2 "bind $secondOutput" \
  [keyargs {-reduce -channel1 -add -ground -triggerControl -triggerOffset -scaleValue
    -verticalOffset -horizontalOffset -invertColor
    -invertWave -timeScale -memory -specMode -specSweep
    -channel2 -control} $margs]
set currentOutput $m.oscilloscope.output

VsRateMeter $m.meter2 \
  -input "bind $currentOutput"
set currentOutput $m.meter2.output

apply VsVideoSink $m.sink \
  -scale $scale \
  -widget $w.screen \
  -callback "$m sinkCallback" \
  -input "bind $currentOutput" \
  $margs

VsLabeledScrollbar $w.vOffset \
  -label "Vertical Offset" \
  -value [$m.oscilloscope verticalOffset] \
  -continuous [true] \
  -converter "vsRoundingLinearConverter -100 100" \
  -inverter "vsLinearInverter -100 100" \
  -callback "$m.oscilloscope verticalOffset" \
  -width 155 \
  -fromVert $w.screen
VsLabeledScrollbar $w.hOffset \
  -label "Horizontal Offset" \
  -value [$m.oscilloscope horizontalOffset] \
  -continuous [true] \
  -converter "vsRoundingLinearConverter -320 320" \
  -inverter "vsLinearInverter -320 320" \
  -callback "$m.oscilloscope horizontalOffset" \
  -width 160 \
  -fromVert $w.screen \
  -fromHoriz $w.vOffset
Command $w.ch1 \
  -label "Ch 1" \
  -foreground $onColor \
  -callback "changeCh1 $m $w" \
  -fromHoriz $w.screen
Command $w.ch2 \
  -label "Ch 2" \
  -foreground $offColor \
  -callback "changeCh2 $m $w" \
  -fromHoriz $w.ch1
Command $w.add \
```

```
  -label "Add" \
  -foreground $offColor \
  -callback "changeAdd $m $w" \
  -fromHoriz $w.ch2
Command $w.color \
  -label "Color" \
  -callback "changeColor $m" \
  -fromHoriz $w.add
VsLabeledChoice $w.control \
  -label "Control Switch" \
  -choices {{0 "Ch 1"} {1 "Ch 2"}} \
  -value [$m.oscilloscope control] \
  -callback "changeChannel $w $m.oscilloscope" \
  -fromHoriz $w.screen \
  -fromVert $w.ch1
Command $w.invert \
  -label "Invert" \
  -foreground $offColor \
  -callback "changeInvert $m $w" \
  -fromHoriz $w.screen \
  -fromVert $w.control
Command $w.ground \
  -label "Ground Off" \
  -callback "changeGround $m $w" \
  -fromHoriz $w.invert \
  -fromVert $w.control
VsLabeledChoice $w.trigControl \
  -label "Trigger Control" \
  -choices {{0 "Free"} {1 "Peak"} {2 "Single"}} \
  -value [$m.oscilloscope triggerControl] \
  -callback "$m.oscilloscope triggerControl" \
  -fromVert $w.invert \
  -fromHoriz $w.screen
VsLabeledScrollbar $w.trigOffset \
  -label "Trigger Offset" \
  -value [$m.oscilloscope triggerOffset] \
  -continuous [true] \
  -converter "vsRoundingLinearConverter -160 160" \
  -inverter "vsLinearInverter -160 160" \
  -callback "$m.oscilloscope triggerOffset" \
  -width 150 \
  -fromVert $w.trigControl \
  -fromHoriz $w.screen
VsLabeledScrollbar $w.scaleValue \
  -label "Scale Factor" \
  -value [$m.oscilloscope scaleValue] \
  -continuous [true] \
  -converter "vsLinearConverter 0 10" \
  -inverter "vsLinearInverter 0 10" \
  -callback "$m.oscilloscope scaleValue" \
  -width 150 \
  -fromVert $w.trigOffset \
  -fromHoriz $w.screen
VsLabeledScrollbar $w.timeScale \
  -label "Time Scale" \
  -value [$m.oscilloscope timeScale] \
  -continuous [true] \
  -converter "vsRoundingLinearConverter 500 2500" \
  -inverter "vsLinearInverter 500 2500" \
  -callback "$m.oscilloscope timeScale" \
  -width 150 \
  -fromVert $w.scaleValue \
  -fromHoriz $w.screen
VsLabeledChoice $w.specMode \
  -label "Spectrum Analyzer Mode" \
  -choices {{0 "Magnitude"} {1 "Phase"}} \
  -value [$m.oscilloscope specMode] \
  -callback "$m.oscilloscope specMode" \
  -fromVert $w.timeScale
VsLabeledScrollbar $w.specSweep \
  -label "Frequency Sweep" \
  -value [$m.oscilloscope specSweep] \
```

```
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 2000" \
    -inverter "vsLinearInverter 0 2000" \
    -callback "$m.oscilloscope specSweep" \
    -width 166 \
    -fromVert $w.timeScale \
    -fromHoriz $w.specMode
  VsLabeledChoice $w.memory \
    -label "Memory" \
    -choices {{0 "Off"} {1 "Save"} {2 "Display"}} \
    -value [$m.oscilloscope memory] \
    -callback "$m.oscilloscope memory" \
    -fromVert $w.timeScale \
    -fromHoriz $w.specSweep
  VsLabeledScrollbar $w.frameRate \
    -label "Frame Rate" \
    -value [$m.oscilloscope frameRate] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 1 30" \
    -inverter "vsLinearInverter 1 30" \
    -callback "$m.oscilloscope frameRate" \
    -width 150 \
    -fromVert $w.memory \
    -fromHoriz $w.specSweep

  Command $w.agc \
    -label "Auto GC" \
    -foreground $onColor \
    -callback "changeAGC $m $w" \
    -fromVert $w.specMode

  VsLabeledScrollbar $w.reduce \
    -label "Reduction" \
    -value [$m.oscilloscope reduce] \
    -continuous [true] \
    -converter "vsRoundingLinearConverter 0 20" \
    -inverter "vsLinearInverter 0 20" \
    -callback "$m.oscilloscope reduce" \
    -width 200 \
    -fromVert $w.specMode \
    -fromHoriz $w.agc

  Command $w.controlPanel \
    -label "Control Panel" \
    -callback "VsPanelShell $w.controlPanel.shell -obj $m" \
    -fromVert $w.reduce
  Command $w.visualPanel \
    -label "Program" \
    -callback "VsVisualShell $w.visualPanel.shell -obj $m" \
    -fromVert $w.reduce \
    -fromHoriz $w.controlPanel


}

proc changeCh1 {m w} {
  global offColor
  global onColor
  if [$m.oscilloscope channel1] then {
    $m.oscilloscope channel1 0
    $w.ch1 setValues -foreground $offColor
  } else {
    $m.oscilloscope channel1 1
    $w.ch1 setValues -foreground $onColor
  }
}

proc changeCh2 {m w} {
  global offColor
  global onColor
  if [$m.oscilloscope channel2] then {
    $m.oscilloscope channel2 0
```

```
    $w.ch2 setValues -foreground $offColor
  } else {
    $m.oscilloscope channel2 1
    $w.ch2 setValues -foreground $onColor
  }
}

proc changeAdd {m w} {
  global offColor
  global onColor
  if [$m.oscilloscope add] then {
    $m.oscilloscope add 0
    $w.add setValues -foreground $offColor
  }  else {
    $m.oscilloscope add 1
    $w.add setValues -foreground $onColor
  }
}

proc changeColor {m} {
  if [$m.oscilloscope invertColor] {$m.oscilloscope invertColor 0} \
    else {$m.oscilloscope invertColor 1}
}

proc changeInvert {m w} {
  global offColor
  global onColor
  if [expr [$m.oscilloscope invertWave] == 1] then {
    $m.oscilloscope invertWave -1
    $w.invert setValues -foreground $onColor
  }  else {
    $m.oscilloscope invertWave 1
    $w.invert setValues -foreground $offColor
  }
}

proc changeGround {m w} {
  if [expr [$m.oscilloscope ground] == 0] then {
    $m.oscilloscope ground 1
    $w.ground setValues -label "Ground On"
  } elseif [expr [$m.oscilloscope ground] == 1] then {
    $m.oscilloscope ground 2
    $w.ground setValues -label "Ground Ref"
  } else {
    $m.oscilloscope ground 0
    $w.ground setValues -label "Ground Off"
  }
}

proc changeAGC {m w} {
  global offColor
  global onColor
  if [$m.oscilloscope gainControl] then {
    $m.oscilloscope gainControl 0
    $w.agc setValues -foreground $offColor
  } else {
    $m.oscilloscope gainControl 1
    $w.agc setValues -foreground $onColor
  }
}

proc changeChannel {wi module channel} {
  global offColor
  global onColor

  $module control $channel

  set gnd [$module ground]
  set trig [$module triggerControl]
  set inv [$module invertWave]
  set toff [$module triggerOffset]
  set sfact [$module scaleValue]
```

```
  set voff [$module verticalOffset]
  set hoff [$module horizontalOffset]

  if [expr $gnd == 0] then {$wi.ground setValues -label "Ground Off"} \
    elseif [expr $gnd == 1] then {$wi.ground setValues -label "Ground On"} \
    else {$wi.ground setValues -label "Ground Ref"}

  if [expr $inv == 1] then {$wi.invert setValues -foreground $offColor} \
    else {$wi.invert setValues -foreground $onColor}

  $wi.trigControl.$trig setValues state true

  $wi.trigOffset.sb setValues -topOfThumb [expr ($toff + 160) / 320.0]
  [$wi.trigOffset.value getSource] setValues -string $toff

  $wi.scaleValue.sb setValues -topOfThumb [expr $sfact / 10.0]
  [$wi.scaleValue.value getSource] setValues -string $sfact

  $wi.vOffset.sb setValues -topOfThumb [expr ($voff + 32767) / 65535.0]
  [$wi.vOffset.value getSource] setValues -string $voff

  $wi.hOffset.sb setValues -topOfThumb [expr ($hoff + 320) / 640.0]
  [$wi.hOffset.value getSource] setValues -string $hoff

}


proc VsOscopeShell {w m args} {
  set cmdArgs [keyargs {-scale -margs} $args]
  set args [keyargs {-scale -margs} $args exclude]

  if {[info commands $m] != ""} then {$m destroy}
  apply VsShell $w \
    -title VsOscope \
    -cmd VsOscope \
    -args [concat $m $cmdArgs] \
    -allowShellResize true \
    $args
}

proc main {} {
  global argv name class errorInfo offColor onColor
  set name [lindex $argv 0]
  set class VsOscope

  set offColor grey
  set onColor white


  if [catch {
    xt appInitialize app_context $class argv {
      {*.Viewport.Form.*.bottom: ChainTop}
      {*.Viewport.Form.*.top: ChainTop}
      {*.Viewport.Form.*.left: ChainLeft}
      {*.Viewport.Form.*.right: ChainLeft}
      {*.Scrollbar.Translations: #override \n\
<BtnDown>: StartScroll(Continuous) MoveThumb() NotifyThumb() \n\
  <BtnMotion>: MoveThumb() NotifyThumb() \n\
  <BtnUp>: EndScroll()}
      {*.font: fixed}
    }
    if [catch {
      set args [lrange $argv 1 end]
      set margs [keyargs {-timeout -scale} $args exclude]
      set args [keyargs {-scale -depth -visual} $args]

      vs appInitialize app_context vs
      $name display display
      $name setValues -allowShellResize true
      apply VsOscopeShell $name.oscope vs.oscope \
-realize "vs start" \
-dismiss "catch {vs destroy}; exit" \
```

```
-margs $margs \
$args
    } msg] {
      VsErrorShell $name.fatal -summary $msg -detail $errorInfo -dismiss exit
    }
    while {[catch {app_context mainLoop} msg]} {
      VsErrorShell $name.err -summary $msg -detail $errorInfo
    }
  }] then {
    puts stderr $errorInfo
    catch {vs destroy}
    exit 1
  }
}

main
```

Bibliography

[1] Acuson Corporation. Acuson-History of Ultrasound. http://www.acuson.com/6.onlineresource/6_3/6_3.html.

[2] Advanced Micro Devices. *MACH 1,2,3, and 4 Family Data Book.* Sunnyvale, CA: 1995.

[3] Hatem Atta. *Ophthalmic Ultrasound: A Practical Guide.* New York, NY: Churchill Livingstone, 1996.

[4] Vanu G. Bose, Andrew G. Chiu, and David L. Tennenhouse. Virtual Sample Processing: Extending the Reach of Multimedia. To appear in *Multimedia Tools and Applications*, Volume 5, 1997.

[5] Douglas Christensen. *Ultrasonic Bioinstrumentation.* New York: John Wiley & Sons, Inc., 1988.

[6] Electronic Design. Advanced Ultrasound Technology Provides Twice the Information in Half the Time. Volume 44, n 19, September 16, 1996: 38.

[7] Electronic Times. Realistic 3D Images from Ultrasound. August 22, 1996: 12.

[8] Harris Semiconductor. RFP8N20L: N-Channel Logic-Level FET. Harris Semiconductor, August 1991.

[9] Paul Horowitz and Winfield Hill. *The Art of Electronics.* New York, NY: Cambridge University Press, 1989.

[10] Innovative Imaging Inc. *Operator's and Maintenance Manual for I^3System-ABD.* Sacramento, CA: 1995.

[11] Michael Ismert, MIT Software Devices and Systems Group. GuPPI Overview. http://www.sds.lcs.mit.edu/SpectrumWare/guppi.html

[12] D. Jennings, A. Flint, B.C.H. Turton, L.D.M. Nokes. *Imaging Technology.* London: Edward Arnold, 1995.

[13] Mustafa Karaman, Ertugrul Kolagasioglu, and Abdullah Atalar. A VLSI Receive Beamformer for Digital Ultrasound Imaging. *IEEE*, September 1992.

[14] Denise Kung. Prototype Software Environment for Digital Ultrasound Review. MIT Master's Thesis, 1994.

[15] Christopher J. Lindblad. A Programming System for the Dynamic Manipulation of Temporally Sensitive Data. *MIT Laboratory for Computer Science Technical Report 637*, August 1994.

[16] Christopher J. Lindblad, David J. Wetherall, William F. Stasior, Joel F. Adam, Henry Houh, Mike Ismert, David R. Bacher, Brent Phillipss, David L. Tennenhouse. ViewStation Applications: Implications for Network Traffic. *IEEE Journal on Selected Areas in Communication* Volume 13, No. 5, June 1995.

[17] Matrox Corporation. PC Technology Builds Better Ultrasound. http://www.matrox.com/imgweb/ultrasnd.htm

[18] Matrox Corporation. Ultrasound Imaging on PC Platform. http://www.matrox.com/imgweb.prulta.htm

[19] National Semiconductor. DM74LS123: Dual Retriggerable Monostable. Santa Clara, CA: 1989.

[20] Northeastern University. Ultrasound Laboratory. http://www.cer.neu.edu/ultrasound.html.

[21] Numerical Recipes in C online Version. Cambridge University Press. http://cfatab.harvard.edu/nr/bookc.html.

[22] Alan V. Oppenheim and Alan S. Willsky with Ian T. Young. *Signals and Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[23] Behrouz Rasoulian. Design of A-Mode Scanning in Ultrasound. Master's Thesis, University of Texas El Paso 1984.

[24] Carol Rumack, Stephanie Wilson, and J. William Charboneau. *Diagnostic Ultrasound, Volume 1.* St. Louis: Mosby Year Book, 1991.

[25] Seattle Times. Siemens Unveils Powerful New Ultrasound Technology. November 26, 1996: F1.

[26] Sonomed Corporation. A-Scan Data Sheets. Lake Success, NY.

[27] Sonomed Corporation. B-Scan Data Sheets. Lake Success, NY.

[28] David Tennenhouse and Vanu Bose. The SpectrumWare approach to wireless signal processing. *Wireless Networks*, 2(1996): 1-12.

[29] The Toronto Globe and Mail. ALI Puts New Picture on Ultrasound. August 28, 1996.

[30] University of Washington Imaging Computing Systems Lab. UW and Siemens Develop Diagnostic Breakthrough with new Programmable Ultrasound Imaging Technology. http://icsl.ee.washington.edu/

[31] Sun Yao Wong. IBM PC/AT Based Diagnostic Ultrasound Imaging System. Master's Thesis, University of Washington.