Carotid Collar: A Device for Auscultory Detection of Carotid Artery Stenosis

by

Jason Ayres Gift

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 25, 2003

Certified by . . . . . .
John Guttag
Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Carotid Collar: A Device for Auscultory Detection of Carotid Artery Stenosis

by

Jason Ayres Gift

Submitted to the Department of Electrical Engineering and Computer Science
on August 25, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The carotid collar is a hardware device designed to aid in detecting carotid artery stenosis, a disease that increases the risk of stroke. This device consists of three electronic stethoscopes that record heart and carotid artery sounds and an electrocardiograph. A software application, `ccrec`, was written to make the device easy to use by displaying real-time waveforms and storing the recorded signals in files for later analysis. The results of some preliminary tests of the device's ability to make accurate recordings, including the performance of the software and a test of the frequency response of the stethoscope sensors, are presented. The results suggest that this inexpensive device has considerable promise for rapid screening for carotid artery stenosis.

Thesis Supervisor: John Guttag
Title: Professor

Acknowledgments

I would especially like to thank John Guttag; without his guidance, keen editing eye, encouragement and support, this thesis would not exist. Completing this project has been a very fulfilling and rewarding process, and is a fitting end to my academic career at MIT.

While thinking about this thesis Dorothy Curtis, Michel Goraczko, and the rest of the ORNet group have been great to bounce ideas off of, and I want to thank them for their help.

I would also like to thank Dr. Lee Shwamm of Massachusetts General Hospital for bringing this field of research to our attention and giving us a tour of the hospital, allowing us to see first-hand what current stenosis exams involve.

Dr. Nat Sims has given us wonderful support, and I'd like to thank him for the contacts, knowledge, and sensors he has provided.

I would like to thank the Center for the Integration of Medicine and Innovative Technologies (CIMIT) for providing financial support.

John Ankcorn was my first UROP supervisor in the Networks and Mobile Systems group at LCS, and I would like to thank him for bringing me into the group and getting me started on the projects that led to working on this one.

I would like to thank my mother and father for doing a great job of raising me: getting me interested in science at an early age and always providing the best tools and opportunities for education.

And finally, to Alice, who has sacrificed my company for the last year, whose telephone calls, letters, encouragement, and love have kept me going through the late nights and long weeks to finish this thesis, I dedicate this thesis, and I send my thanks and endless love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Goals

Carotid artery stenosis is a medical condition which afflicts many people throughout the world. Stenosis refers to the narrowing of arteries, caused either by aging tissue or, more commonly, the deposition of plaque (called atheroma and consists of fatty material, smooth muscle cells, and connective tissue) on the artery walls, also causing clogging[12]. This clogging reduces blood flow and causes turbulence in the blood flow through the artery. However, there are many people who have severe stenosis, but have no associated problems (this is known as asymptomatic stenosis). The real problem is when a piece of plaque breaks off and enters the blood stream. This plaque may become lodged in a smaller artery, quickly blocking blood flow and causing a stroke [13]. Early detection of plaque which may break off is critical in preventing this kind of stroke.

Strokes are the third leading cause of death in the United States and the leading cause of adult disability[18]. A stroke occurs when there is oxygen deprivation in part of the brain[13]. Oxygen deprivation can cause many different types of damage to the brain, affecting motor skills, cognition, language ability, and others[13].

Physicians will often screen patients who are at risk of stroke using a stethoscope to listen to the carotid arteries. Auscultation, as the act of listening with a stethoscope is called, is not a perfect diagnostic tool. If the patient has stenosis, the physician

may hear a whooshing sound (a bruit) caused by turbulent blood flow just past the stenosed area of the vessel. However, at least one study suggests that there may not be an audible bruit in stenosis as severe as 70-99%[6].

Normally, if the physician suspects there might be serious carotid artery stenosis, he or she will recommend that the patient get a doppler ultrasound exam done. This is the current gold standard, or best available method, for diagnosis of carotid artery stenosis. The long term goal of this project is not to replace the doppler ultrasound as the gold standard, but rather to provide a better screening tool. We think it is possible to use electronic stethoscopes to listen to the carotid arteries and screen for possible stenosis.

Another study suggests that bruits become more prevalent with increasing stenosis up to maximum, after which they become less likely because the greater stenosis causes lower blood flow, and the turbulence is consequently less[7]. So even though the bruit is there, it is too quiet to be heard by a human. Electronic stethoscopes can be more sensitive than the human ear, so it may be possible to hear bruits that a human cannot.

The first step towards a system that sensitively and accurately detects carotid artery stenosis is the design and implementation of a device to start collecting data that can be analyzed to develop an algorithm for detection. This thesis covers the creation of the hardware and software constituting such a device, and shows that it is reasonable to expect that it could be used to detect carotid artery stenosis.

The following four important criteria are the design goals that the device must accomplish:

1. Safety of the examinee

   The device should not pose any health risks to the patient. The patient should be protected electrically from the circuitry and physically from the sensors and components (wires should not be able to strangle the examinee, nor should there be any sharp edges or points).

2. Accurately record stenosis sounds

The device should make every attempt possible to make a recording of sounds related to stenosis without distorting them or filtering out any of the relevant sound. However, it should attempt to filter out or otherwise reduce noise in the signal.

3. Easy to set up and use

   It should take a short time and limited effort to set up the device (attach the sensors to the patient, start the software). The software interface should be easy to use: collecting data should be a simple task as well as quickly reviewing data just collected to make sure the quality is adequate.

4. Portability and robustness

   The device should be portable enough to move it from room to room in a hospital or around a primary care physician's office without damaging it. It should hold up to normal wear caused by frequent use.

1.2 Stenosis is Difficult to Detect

There are many factors that make stenosis a difficult disease to detect by auscultation. There are both physical and physiological reasons why it is hard to hear a bruit, or why it may not even exist.

The carotid arteries supply blood to the brain, neck, and head. On each side of the neck, the carotid artery consists of three parts: the common or main carotid, the external carotid, and the internal carotid (*Figure 1-1*) [8]. The main carotid comes up from the base of the neck and splits into the external and internal carotids[8]. The external carotid continues upwards, near the surface of the skin, and branches several more times to supply blood to the face and other external parts of the head[8]. The internal carotid artery remains near the surface for a short distance, and then bends inwards, deep into the neck, supplying blood to the front of the brain, the eyes, nose, and forehead[8].

When stenosis occurs in the external or main carotid and a bruit exists, then

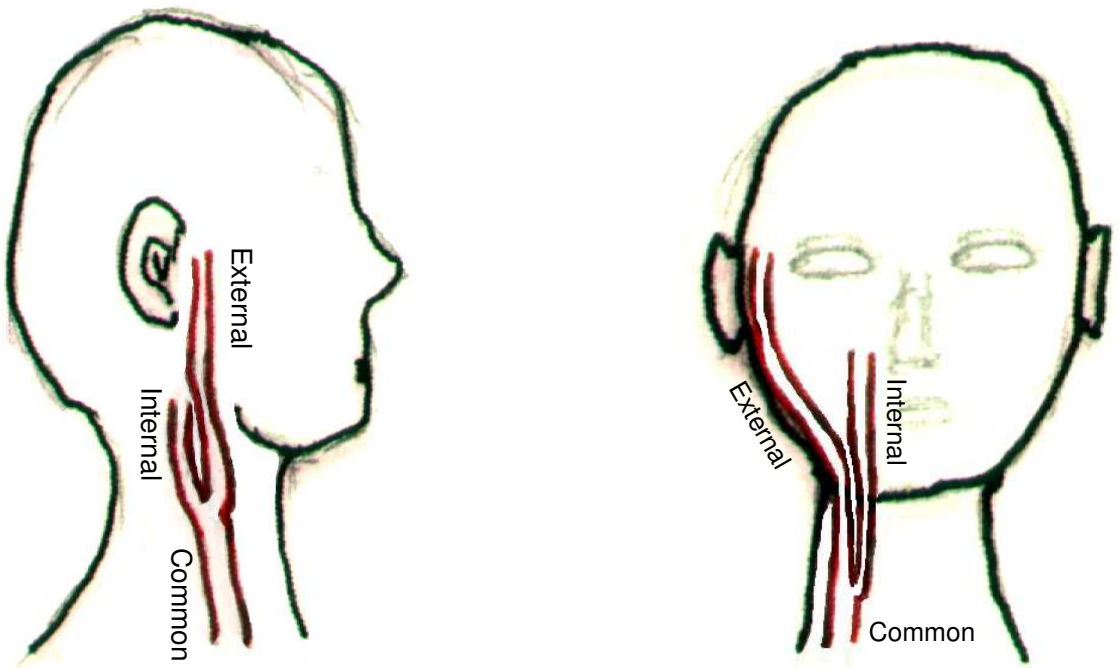


Figure 1-1: The three branches of the carotid artery, from the side (a) and front (b). Sketched using Figures 283 and 289 from [8] as a reference.

it is easier to hear the bruit because the source of the sound is close to the skin. However, if there is stenosis in the internal carotid, but in neither the external nor main carotids, then a bruit is easy to miss. The problem is that the sound of normal blood flow in the external carotid is loud when listening with a stethoscope, due to the close proximity to the skin, but the bruit in the internal carotid is relatively quiet because it is behind a lot of flesh. The external carotid sounds may even completely drown out the internal carotid sounds.

One possible solution to this dilemma is to use multiple stethoscopes on each side of the neck and post-process the signals using Independent Components Analysis (ICA). ICA is a statistical signal processing technique that uses multiple simultaneous recordings to try to separate mixed signals that are independent of each other. Although the sounds coming from the three carotids are not strictly independent, they are different and each occur at a slightly different time. If ICA is going to be used, then we need to collect as many different signals as there are components, or three per side of the neck.

The issue is further complicated because both heart murmurs and stenosis that occurs between the heart and the carotids may cause sounds that propagate through the arteries up into the neck[14]. In this case, these sounds should be heard through both the left and right main carotids almost equally. However, rather than rely on measuring the difference in the sounds between the carotids, it would be easier to additionally record sounds from near the heart, where a murmur would be loudest and a carotid bruit would not be audible[14].

Because it takes time for blood to flow through arteries, the sounds heard in the neck region are slightly delayed from those heard near the heart. The length of the delay conveys information about the speed of the blood flow. In order to tell when the heart beat occurs and have a reference from which to measure delays, an electro-cardiogram (EKG) should also be recorded.

EKG would also be useful when separating the four sounds, called S1, S2, S3, and S4, which occur during each heartbeat. Because there are two contractions (the left and right ventricles) and two valve closings associated with each heartbeat, there are also four pressure waves, and therefore four sounds, that happen with every beat. The first two sounds, S1 and S2, which occur during the higher pressure of systole, are likely to be more useful in detecting bruits than the last two sounds, which are during the lower pressure of diastole[14]. The higher pressure means that the blood will be flowing faster through the vessels and any turbulent blood flow caused by stenoses should be larger and louder than during diastole.

1.3 Approach

Because of the complicated nature of this disease and the various other problems that can make screening difficult, there are now seven stethoscope sensors needed and one EKG. Rather than trying to solve the whole problem at once, the first version of the device will have only four sensors: three stethoscopes and EKG. Using these four sensors, it should be possible to detect stenosis in the external/main carotids and distinguish it from a stenosis closer to the heart.

If this is successful, then further research into separating the sounds from the three branches of the carotids can take place. Since this will require the addition of more or different sensors, the design of this device must be modular, allowing sensors to be added without re-writing the recording software or modifying existing sensor hardware.

There are already electronic stethoscopes in production by several companies. They use a piezo-electric sensor, have a built-in amplifier, often have a built-in low and high-pass filters, and a headphones. One example is the Master Elite Stethoscope, made by Welch-Allyn with a sensor made by Meditron. Since it is designed to be a replacement for a traditional stethoscope, its packaging is shaped to be comfortably held, and is consequently bulky. The packaging would make it difficult to arrange several of these stethoscopes around a patient and hold them in place. Instead of using a commercial stethoscope, a new, specialized device, using simple electronics and commercial sensors would be more useful.

If it is possible to make such a device, and the associated algorithm to analyze the data collected by the device, then it could be an easy-to-use, consistent, more sensitive alternative to traditional auscultation screening for carotid artery stenosis. It could also reduce the number of false positive diagnoses sent to have an unnecessary, expensive doppler ultrasound exam. This thesis covers the design and implementation of the device and software used to make recordings of the carotid arteries that are suitable for automated analysis.

1.4 Contributions of this Work

The major contribution of this work is expected to be to the field of carotid artery stenosis research. By creating a device that can be used to record bruits and other related signals, in addition to being expandable, this thesis provides the basis from which further research into carotid artery stenosis can be performed.

More generally, this thesis shows that it does not require expensive, proprietary hardware and software to record medical signals. This is not the first time that a

sound card has been used as a digital acquisition device, but it provides another example of how it can be done. This thesis will hopefully encourage more research into medical signal processing by showing that it is possible to do signal acquisition with inexpensive hardware.

1.5 Thesis Organization

The requirements and design of the hardware will be discussed in *Chapter 2*, followed by the software in *Chapter 3*. The preliminary results from testing will be explained in *Chapter 4*. Additional applications of this device, future improvements, and an evaluation of how well the device meets the design goals is in *Chapter 5*.

Chapter 2

Hardware

The hardware was designed to meet the four goals set down in *Chapter 1*: safety of the examinee, accurate recording of signals, easy set-up and use, and portability. In addition, the hardware should be modular so that it is simple to add more or different devices in the future if they are needed.

Keeping these five goals in mind, the rest of this chapter describes the hardware in detail, explaining how it works and showing how it meets each of these goals. The general architecture of the device is described first, followed by the details of the stethoscope, EKG, and power circuitry. The chapter concludes with a description of the mechanical hardware.

2.1 Architecture

A block diagram of the basic architecture is shown in *Figure 2-1*. Each sensor has a circuit that amplifies the signal and does some noise reduction. The amplifier gets its power from a USB port on a hub, which gets its power from the USB port on the computer. The output from the amplifier is connected to a sound card (in this figure a USB sound card, although it could also be a sound card internal to the computer). The signal data, after being digitized by the sound card, is passed to the computer through the USB bus.

There is an assumption made in this chapter, which is that the sound card or

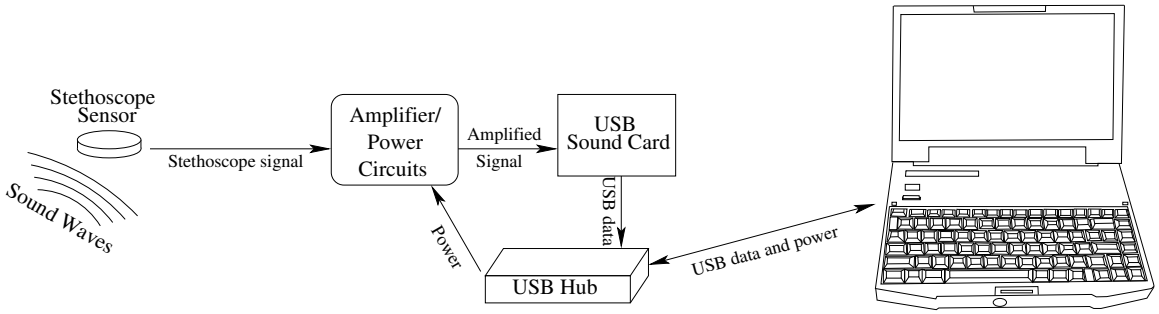


Figure 2-1: Block diagram of the device

other audio device into which these sensors will be plugged will have a low-pass filter before their sampling circuit that is appropriate for that sound card's sampling rate. The Nyquist sampling theorem states that in order to avoid noise caused by aliasing, a signal must be sampled at a rate that is at least twice that of the highest frequency information in the signal. For example, interference from an FM radio broadcast (at about 100Mhz) may cause aliased noise if the signal is only sampled at 44.1 kHz. However, if a 22.05 kHz low-pass filter were used, the signal could be sampled at 44.1 kHz without inducing aliasing.

So, rather than making an assumption about the sampling rate of the sound card, a wider range of audio devices can be supported by assuming those devices have the low-pass filter on their inputs. This assumption is reasonable since most sound cards are designed to have unamplified, unfiltered microphones plugged directly into their inputs.

2.2 Stethoscope Circuit and Sensor

There are some materials that generate an electric potential when physically deformed. This property is called the "piezoelectric effect[9]." Because many of these materials produce an electric potential that is roughly proportional to their deformation, they can be used to make devices called pressure transducers which convert changing pressures into an electric signal. Since sound propagates as pressure waves, it is therefore possible to use these piezoelectric pressure transducers to convert sound

to an electrical signal.

There are two different stethoscope sensors used in this project, one made by Meditron, and the other custom-made by Jim Kassal. Each is constructed with two piezoelectric "films" (thin, flexible piezoelectric devices) on top of each other. These films are then wrapped around a cylindrical gel perpendicular to the skin (Meditron sensor), or put over something which allows them to physically couple with the body, with the films parallel to the skin (Kassal sensor).

Each sensor has its advantages and disadvantages. The Meditron sensor is inexpensive and the small size is both a benefit and a detriment. It can be placed more carefully over a specific area of the body, possibly getting less signal from unimportant sound sources, but at the same time it has to be placed carefully in order to record the desired sound. The Kassal sensor, on the other hand, has a much larger surface area, making placement less critical, but the price is significantly higher.

In making these stethoscope sensors, the manufacturers are careful to match the physical impedance of the sensor to that of a human body. Just as an impedance mis-match in electronics can cause attenuation or reflections of signals, a physical impedance mis-match can cause attenuation or reflections of pressure waves. So, it is important that the material used to couple the film with the body has properties similar to human skin, since that will provide the best possible signal.

Generally speaking, piezoelectric devices act like voltage sources, whose voltage is dependent on the degree of deformation. However, these voltages for quiet sounds such as a heartbeat are often only on the order of $1mV$. Additionally, the devices have relatively high internal resistance. This means that the voltage will be reduced as more current is drawn from them, so they are actually non-ideal voltage sources. Therefore, their output must be buffered (protected against being changed by measurement) and amplified before the signal can be used by other circuits.

Furthermore, they are prone to electric noise. This problem is overcome by using two films together. Piezoelectric devices have polarity and will generate a positive potential if deformed in one direction, but a negative potential if deformed in the opposite direction. This property can be used to an advantage by placing two films

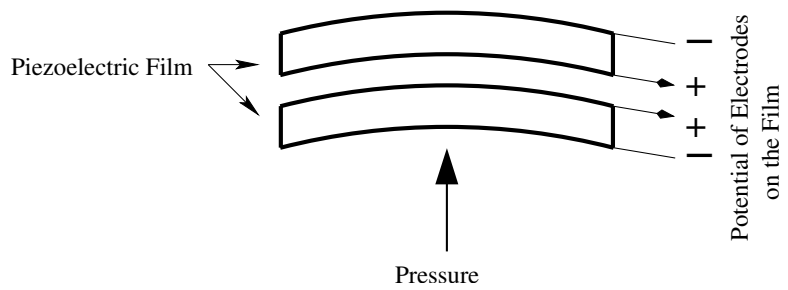


Figure 2-2: The piezoelectric films are placed back-to-back to generate opposite potentials when exposed to pressure.

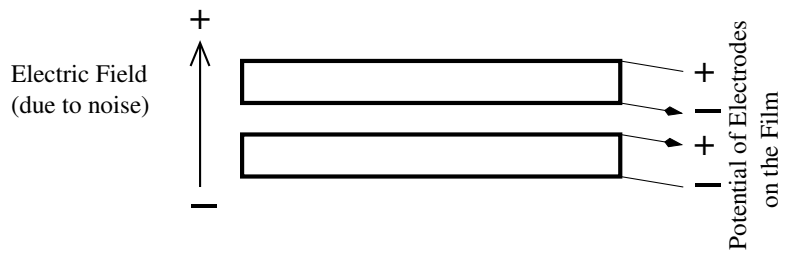


Figure 2-3: The piezoelectric films generate same potentials when exposed to electrical noise.

back-to-back. If this is done, a deformation will cause a negative potential on one of them but a positive potential on the other (*Figure 2-2*). However, the polarity of any noise (such as 60Hz noise) will depend on the orientation of the sensor rather than the direction of the deformation, so the noise will have the same polarity on both films (*Figure 2-3*). If the noise is close to the same amplitude on both films, subtracting the two signals from the films will reduce the noise while boosting the signal due to the deformation of the sensor. This is called "common mode rejection."

The circuit between the stethoscope sensor and the audio device serves three purposes: to buffer the audio output signal by providing a higher-current voltage source, to decrease noise through common mode rejection and filtering, and to amplify the signal. The circuit used consists of three operational amplifiers (*Figure 2-4*).

The operational amplifiers (op-amps) are useful because the input terminals require very little current, while the output can supply much more current. This allows an op-amp to act as a "buffer" between a low-current device, and satisfies the buffering

purpose of the stethoscope circuit.

The op-amps in this circuit are arranged into two stages. The first stage (the two op-amps on the left in *Figure 2-4*) buffers the output of the stethoscope (denoted by V_s^+ and V_s^-) and implements a high-pass filter. In addition to providing buffering, this stage also implements part of the noise reduction of the stethoscope circuit.

Because we are interested in collecting audio data with this device, it is important that we can reject two kinds of noise. The first can be observed as a constant voltage on the inputs to the differential amplifier circuitry. Sources of this noise include static charges on the piezoelectric films and the constant pressure with which the sensor is pressed against the patient. This DC (zero frequency) signal can be reduced by high-pass filtering the output of the stethoscope sensor. However, in order not to distort the real signal, the filter must have a very low cut-off frequency. In a prior carotid bruit study, the "break frequency" of the bruits (the frequency of maximum energy) was found in the range of 70 to 640 Hz[5], so it is important to be able to capture frequencies in at least this range. The filter should also not distort the signal by changing the phase of the various frequency components. So while it does not need to be a general linear-phase filter, it should at least be linear-phase in the range of frequencies of interest.

The second type of noise comes from electric and electromagnetic interference. Common sources of this noise are radio broadcasts: both intentional broadcasts, such as wireless network cards or music stations, and unintentional, such as microwave ovens. However, most of these are in frequencies that are many orders of magnitude higher than the audio data of interest, so the real problem lies with the power lines which run through walls and the 60 Hz electric fields they generate.

The second stage (the single op-amp on the right in *Figure 2-4*), subtracts the signal generated by the first film from the signal of the second film, thereby meeting the second purpose (noise reduction) of the stethoscope circuit. The following analysis of the stethoscope circuit will show in detail that it meets these stated goals: buffers and amplifies the input, and reduces noise.

For this analysis, the impedance method will be used (resistors have an impedance

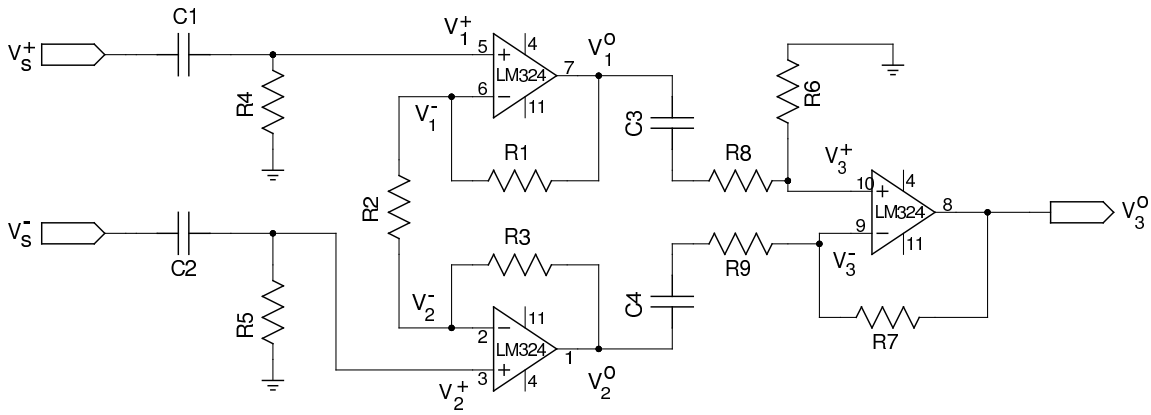


Figure 2-4: Stethoscope circuit schematic

of R, capacitors have an impedance of $1/Cs$, where $s = j\omega$, and ω is frequency). Furthermore a standard op-amp model will be used, which states that as long as there is feedback from the output to the negative input, the two inputs are kept at the same potential. From the op-amp model, we obtain *Eqs. 2.1, 2.2, 2.3*.

$$V_1^+ = V_1^- \tag{2.1}$$

$$V_2^+ = V_2^- \tag{2.2}$$

$$V_3^+ = V_3^- \tag{2.3}$$

Applying the impedance model to the impedance bridges formed by C1/R4 and C2/R5, we arrive at *Eqs. 2.4* and *2.5*:

$$V_1^+ = \left(\frac{R_4}{R_4 + \frac{1}{C_1 s}}\right) V_s^+ \tag{2.4}$$

$$V_2^+ = \left(\frac{R_5}{R_5 + \frac{1}{C_2 s}}\right) V_s^- \tag{2.5}$$

Doing the same for the bridge formed by R1, R2, and R3, we get the following definitions for V_1^- and V_2^-:

$$V_1^- = \left(\frac{R_2 + R_3}{R_1 + R_2 + R_3}\right)(V_1^O - V_2^O) + V_2^O \tag{2.6}$$

$$V_2^- = \left(\frac{R_3}{R_1 + R_2 + R_3}\right)(V_1^O - V_2^O) + V_2^O \tag{2.7}$$

Finally, the last two node equations can be found by using the bridges formed by C3/R8/R6 and C4/R9/R7:

$$V_3^+ = \left(\frac{R_6}{R_6 + R_8 + \frac{1}{C_3 s}}\right) V_1^O \tag{2.8}$$

$$V_3^- = \left(\frac{R_7}{R_7 + R_9 + \frac{1}{C_4 s}}\right)(V_2^O - V_3^O) \tag{2.9}$$

There are now nine equations with nine unknowns (the node voltages at the inputs and outputs of the op-amps), so it is possible to solve for the unknowns. First, *Eqs. 2.4* and *2.6* are combined with the op-amp model (*Eq. 2.1*) to arrive at:

$$V_s^+ \left(\frac{R_4}{R_4 + \frac{1}{Cs}}\right) = \left(\frac{R_2 + R_1}{2R_1 + R_2}\right)(V_1^O - V_2^O) + V_2^O \tag{2.10}$$

The same is done with *Eqs. 2.5, 2.7,* and *2.2* to get the following relationship:

$$V_s^+ \left(\frac{R_4}{R_4 + \frac{1}{Cs}}\right) = \left(\frac{R_1}{2R_1 + R_2}\right)(V_1^O - V_2^O) + V_2^O \tag{2.11}$$

And again, the same thing is done with the third op-amp (*Eqs. 2.8, 2.9,* and *2.3*):

$$V_1^O \left(\frac{R_1}{2R_1 + \frac{1}{Cs}}\right) = \left(\frac{R_1}{2R_1 + \frac{1}{Cs}}\right)(V_2^O - V_3^O) \tag{2.12}$$

Simplifying *Eq. 2.12*, it can be seen that the output of the differential amplifier circuit is the difference between the outputs of the first stage amplifiers:

$$V_1^O - V_2^O = -V_3^O \tag{2.13}$$

Next, we combine *Eqs. 2.10* and *2.11*, getting rid of a V_2^O term in each:

$$V_s^+\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right)-\left(\frac{R_2+R_1}{2R_1+R_2}\right)(V_1^O-V_2^O)=V_s^-\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right)-\left(\frac{R_1}{2R_1+R_2}\right)(V_1^O-V_2^O) \tag{2.14}$$

Because of *Eq. 2.13*, we can substitute for $(V_1^O - V_2^O)$ in *Eq. 2.14*:

$$V_s^+\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right)+V_3^O\left(\frac{R_2+R_1}{2R_1+R_2}\right)=V_s^-\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right)+V_3^O\left(\frac{R_1}{2R_1+R_2}\right) \tag{2.15}$$

Rearranging the terms by bringing the V_s^- term to the left-hand side, we arrive at:

$$(V_s^+ - V_s^-)\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right)=V_3^O\left(\frac{-R_2}{2R_1+R_2}\right) \tag{2.16}$$

And finally, by division, V_3^O (the output of the differential amplifier circuit) is left by itself:

$$V_3^O=-(V_s^+ - V_s^-)\left(\frac{2R_1+R_2}{R_2}\right)\left(\frac{R_4}{R_4+\frac{1}{Cs}}\right) \tag{2.17}$$

Substituting in the actual values of the capacitors and resistors, we get *Eq. 2.18*:

$$V_3^O=-(V_s^+ - V_s^-)5.205\left(\frac{2.2\times10^6}{2.2\times10^6+\frac{1}{0.22\times10^{-6}s}}\right) \tag{2.18}$$

From *Eq. 2.18*, it can be seen that the output of the stethoscope circuit is proportional to the difference of the inputs. Since $V_s^+ \approx -V_s^-$ the first term is approximately equal to $2V_s^+$, so the output signal will be proportional to about 10 times the amount the sensor is deformed by a pressure wave. There is a trade-off between a large gain, which allows smaller signals to be recorded but causes clipping (distortion) if there are larger signals, and a small gain, which prevents clipping from happening, but causes small signals to be too low to be captured by a sound card. This particular gain was chosen by experimentation with different gains. With both the Meditron sensor and the sensor made by Jim Kassal it is possible at this gain to capture heart and artery

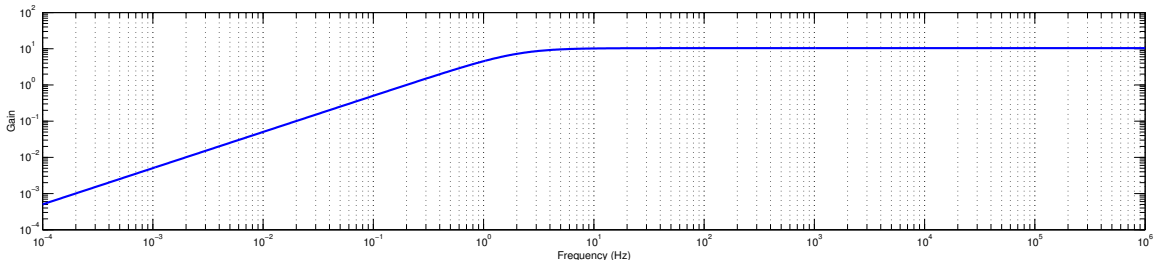


Figure 2-5: Theoretical Frequency Response of the Stethoscope Circuit (magnitude)

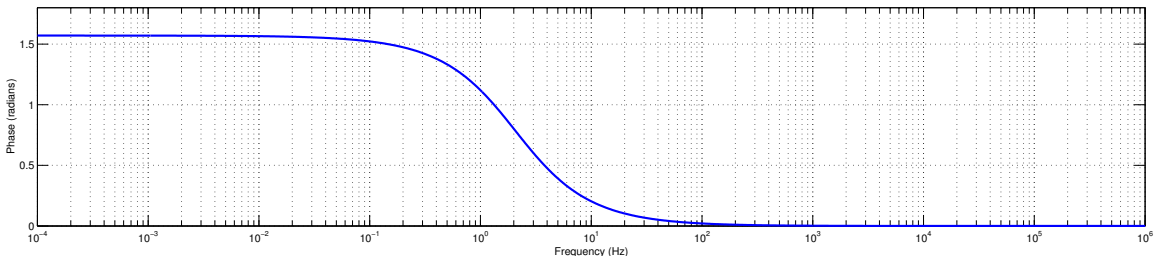


Figure 2-6: Theoretical Frequency Response of the Stethoscope Circuit (phase)

sounds without causing clipping when the person wearing the device swallows, coughs lightly, or talks very quietly. The swallowing or coughing sounds are not of interest, but if they cause clipping they will also distort the heart and artery sounds.

Also from *Eq. 2.18*, as the frequency goes to 0 Hz ($s \rightarrow 0$), the denominator grows infinitely large and the gain goes to zero. The differential amplifier blocks DC noise while still passing the higher-frequency audio information. A plot of the differential amplifier gain versus the frequency of the input can be found in *Figure 2-5*. The gain is almost constant for frequencies higher than 10 Hz, and it is still greater than one for frequencies as low as 1 Hz. However, it drops off quickly and severely attenuates frequencies below 0.1 Hz.

The phase response of the differential amplifier (*Figure 2-6*) also remains constant (and nearly zero) at frequencies greater than 100 Hz. It is still nearly zero at frequencies greater than 10 Hz, but becomes large very quickly for frequencies below 10Hz. However, since little useful audio information related to bruits is expected to be at such low frequencies, this phase distortion is not a problem.

Since the differential amplifier circuit will accomplish its goals, the only other

requirement to consider is its safety. However, this is a simple matter, since there is no part of the circuitry that intentionally comes into contact with the person wearing this device. Even if accidental contact is made with any part of the circuit, the maximum potentials present are $\pm 5V$, so they are not dangerous. Furthermore, the ground of this circuit is floating since it is powered off of batteries, so there is no increased risk of shock from external power sources.

2.3 EKG Circuit

Just as it is important to design the stethoscope circuit to be able to accurately record an audio signal, it is important to create an EKG circuit that can be used to accurately record an EKG signal. As with the stethoscope circuit, the input signal is very small and can not provide much current. So the same three conditions must be met: the amplifier circuit must buffer the signal, it must pass all frequencies in which the EKG signal has useful information and reject noise, and it must amplify the signal.

For EKG, the frequencies of interest lie between 0.005 Hz and 100 Hz [10], a much narrower band than for an audio signal. The basic circuit was modeled on one which appeared in the June, 2000 issue of *Scientific American*[3]. However, several key changes were made. A different instrumentation amplifier was chosen: the AD627, instead of the discontinued AD624. Also, since the input impedance is higher on the AD627, series resistors were no longer needed on the inputs to the instrumentation amplifier. Instead, a parallel resistor to ground was added, which, in parallel with the natural capacitance of the skin, filters out some of the DC noise. The low-pass filter on the output of the amplifier was removed. This is because the original intent of the circuit in *Scientific American* was for use with another device, such as an oscilloscope, that might not have had a low-pass filter. Instead, the output of this circuit will go to a sound card, which is assumed to have a built-in low-pass filter which is designed specifically for the sampling circuitry of that sound card. Finally, a high-pass filter was added to the output because the presence of extremely low frequency EKG signal

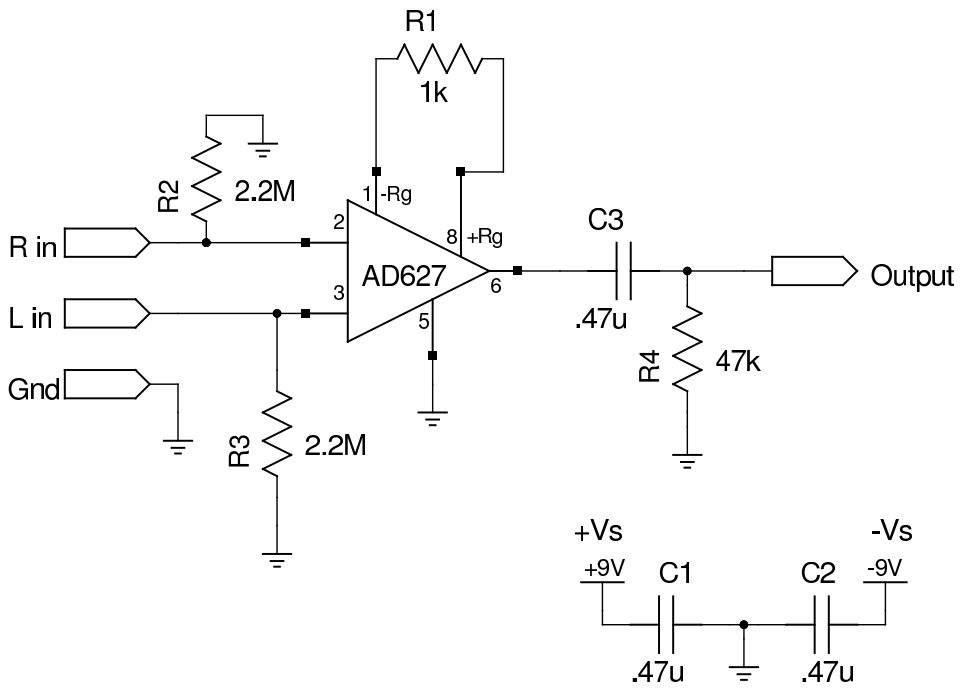


Figure 2-7: EKG amplifier circuit schematic

makes detection of the QRS complex (for heart beat timing) difficult[10].

The result is the circuit in *Figure 2-7*. For the AD627, the gain is set by a single resistor across pins 1 and 8[1]. With a $1k\Omega$ resistor, the gain is approximately 200, according to *Table 1* in the datasheet[1]. This is sufficient enough gain to make an EKG signal, which is approximately one millivolt peak-to-peak[10] almost a volt, which is easily recorded by a sound card, meeting the amplification condition. In addition, the input current is less than $10nA$, while the output can supply $25mA$, meeting the buffering condition.

A similar trick to the two-film noise solution of the stethoscope sensor can be used with EKG. Two "leads" are used (a lead refers to one potential measured between two electrodes on the skin), one formed by measuring the potential from an electrode on the left side of the heart to a common ground electrode below the heart, and another formed from the electrode to the right side of the heart and the common ground electrode. The EKG signal, due to the electrical signals flowing through the nerves in the heart, has opposite polarity, and roughly equal magnitude on either side of the

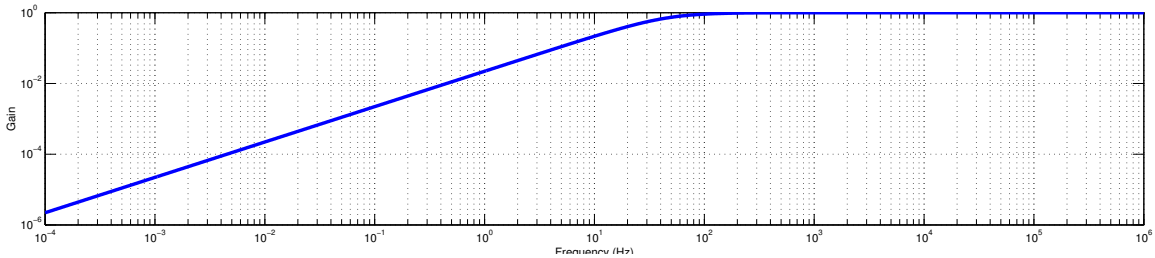


Figure 2-8: Frequency response of the EKG amplifier's output high-pass filter

heart. As long as the fields associated with noise from 60 Hz power lines and radio interference are not lined up with the line formed by the left and right electrodes, the noise signal will appear approximately the same on both leads, and so subtracting the two leads from each other will mostly eliminate the noise in the signal. This is a practical concern, since any wall usually contains many power lines and so there are many 60Hz electric fields. However, during testing of the EKG circuit the orientation of the person wearing the EKG had to be very close to the orientation of the field in order for the interference to be large. Since a changing the orientation by only a couple of degrees greatly reduced the interference, this alignment issue is not a large problem.

The frequency response of the high-pass filter after the output of the instrumentation amplifier is shown in *Figure 2-8*. This filter does not significantly attenuate frequencies as low as 3 Hz. However, frequencies below 0.1 Hz are attenuated greatly. Normally these are useful in diagnosing certain heart conditions, but the purpose of this EKG is to do time correlation by using the QRS wave. Since EKG signals are often passed through a 3 Hz high-pass filter when only considering the QRS wave [10], this filter is reasonable.

When the gain of the amplifier is set to about 200, it is nearly constant over frequencies less than a kilohertz (figure TPC 20, in [1]), and does not drop off significantly until frequencies over 10 kHz, so the amplifier itself behaves well within the bounds of the EKG signal. Furthermore, the common mode rejection ratio (CMRR) of this amplifier, which measures its ability to attenuate noise that appears with the same polarity at both terminals, is greater than 100 dB for frequencies below 100

Hz[1]. Because of this, it will easily reject 60 Hz noise, which is the largest component of radio interference. This and the high-pass filters satisfy the second condition on the operation of the EKG amplifier.

This leaves only the question of safety to be considered. Because this amplifier has a direct electrical connection to the body, safety is a higher concern than with the stethoscope circuit. However, the input bias current of this amplifier, which is the current it will sink or source at its inputs, has a maximum of $10nA$, which is well below the maximum for human safety (look up citation). Furthermore, the input impedance of the instrumentation amplifier is $20G\Omega$, so the body is very well isolated from the rest of the circuit by the instrumentation amplifier.

There are two other paths for current to flow in the connections to the patient. These are the parallel resistors to ground on the inputs to the instrumentation amplifier, and the third electrode, which is a direct connection to ground. Because the resistance through the parallel resistors is very large ($> 1M\Omega$), if the patient were exposed to a large potential compared to the circuit's ground, it would flow predominantly through the third electrode, so that path is the real concern.

However, since this circuit is powered from the USB bus, the largest potential the patient may be accidentally exposed to because of the circuit is from the power supply for the instrumentation amplifier, which has potentials of $\pm 5V$, which is a safe level for humans.

Furthermore, as long as the laptop is not plugged into wall power, the circuit's ground is floating relative to earth ground. Even if the person wearing the EKG electrodes were to be exposed to a large potential relative to earth ground (such as the hot wire from a wall outlet), there would be no discharge through the EKG circuit. If it is desired to have the laptop plugged in during an examination, then additional protection circuitry would need to be added to ensure isolation of the patient from wall power. This circuitry already exists in other medical devices and could be adapted to this circuit.

2.4 Power Conversion Circuit

Power consumption is a concern any time the availability is limited. It is also useful to limit power usage for safety reasons. The less power that is available to flow into a circuit, the lower the risk that it might unintentionally flow into the patient in the case of equipment failure or short circuits. Additionally, if power consumption is low enough, everything can be run off of batteries, making it more portable and removing a connection to wall power, increasing safety. Furthermore, the amplifiers must be powered by a dual-rail supply ($\pm 5V$) because the voltage output from the sensors may have negative potential.

The result is a power conversion circuit which uses power from the USB bus on the host computer. The USB bus has an easy external connector which is standardized across all computer platforms, but it supplies only $+5V$, so the power conversion circuit must be able to generate $-5V$. The selection of a particular converter depends on the power requirements of the amplifiers being powered by it, so an analysis of the power availability and requirements will follow.

The power is taken from the USB bus, but not directly from the port on the computer. A USB port on a computer can supply up to $500mA$[4]. However, because a laptop might only have one USB port, the power is taken from a USB hub so that other USB devices can be used at the same time. A bus-powered hub (one which draws its power from the USB connector on the host, rather than a separate power supply), supplies a maximum of $100mA$ on each port[4]. So at least $100mA$ is available. Since USB hubs usually have four ports, possibly as much as $300mA$ is available if only one normal USB device (such as a sound card) is plugged into the hub.

In addition to being chosen for their operating properties, as explained in the previous sections, the operational and instrumentation amplifiers were chosen for their low power consumption. The AD627 instrumentation amplifier has a maximum quiescent supply current of $85\mu A$, and an output source current of $25\mu A$[1]. The LM324 (operational amplifier used for the stethoscopes) has a maximum quiescent

supply current of $3mA$, with an output source current of $40mA$ per channel[16]. It also has an input bias current of $250nA$[16].

Although the output source currents are large, this is what is available to drive the next stage and not really representative of the actual output current. These amplifiers are driving the input of an audio device, which usually have buffered, low-current inputs, so the actual output current would be much smaller than the maximum output current. Since the amplifiers may be connected to different sound cards depending on the computer being used, it is useful to have a reasonable approximation for the maximum output current. An analog to digital converter (ADC), such as the AD7654, is a 16-bit ADC which is capable of two-channel, 44100kHz sampling (comparable capabilities to a sound card). It has an input current of $45\mu A$[2]. As another example, the UDA1342, a fully-featured, single-chip audio device, has an input resistance of $10k\Omega$. With the maximum voltage output from the amplifier being less than $5V$, this would lead to a maximum output current of $5V/10k\Omega = 0.5mA$. Therefore, although the exact output current will depend on the audio device to which the amplifiers are connected, a reasonable assumption for a maximum output current would be on the order of $1mA$.

Using this information the maximum supply current for the stethoscope circuit would be the sum of its quiescent supply current, the input current on the inputs of its output stage (and only the output stage, since the input current to the first stage is supplied by the sensor), and the output current of the output stage, which totals $4.0mA$. For the EKG amplifier, the maximum supply current is the sum of the quiescent supply current and the output current, or $1.1mA$.

Based on these maximums and ease-of-use considerations the MAX889R, an inverting charge pump, was chosen. It only requires three external capacitors and two external resistors, making it a small and simple device to use. Furthermore, because it could provide up to $200mA$ of output current[11], it can handle the full current capabilities of a single USB hub port.

The circuit used was based on the "Typical Operating Circuit" contained in the datasheet, and values for the capacitors were taken from the recommended values in

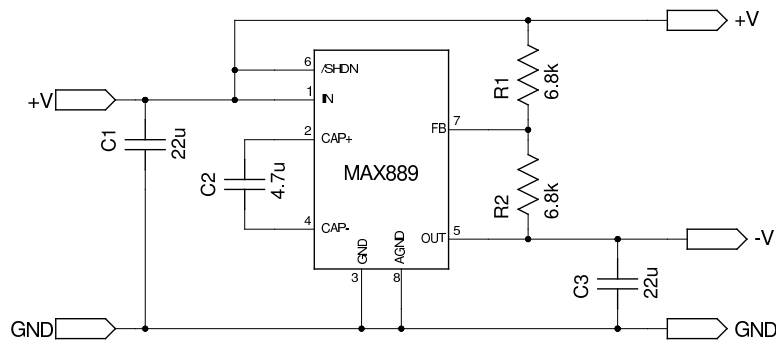


Figure 2-9: Power inverter circuit schematic

Table 1. There is a capacitor (C1) on the input voltage, that helps to filter noise out of the power supply (*Figure 2-9*). Capacitor C2 is the flying capacitor that is used to store charge by the charge pump to generate a negative voltage on the output. Capacitor C3 is an output filter that reduces the ripple on the output voltage. This ripple is caused by the switching operation of the charge pump.

Resistors R1 and R2 control how quickly the flying capacitor is charged, regulating the output voltage (the faster C2 is charged, the more negative the output voltage). The datasheet recommends choosing an R1 and R2 such that these three equations are satisfied[11]:

$$V_{OUT} = -V_{REF}(R_2/R_1) \tag{2.19}$$

$$R_1 \leq V_{REF}/30\mu A \tag{2.20}$$

$$R_2 < -V_{REF}/30\mu A \tag{2.21}$$

V_{REF} is the voltage, relative to ground, measured on R1's positive side (top connection in this schematic), which is equivalent to the positive supply voltage in this circuit. According to *Eq. 2.19*, in order for $V_{OUT} = -V^+$, R1 and R2 must have the same value. Furthermore, R1 and R2 must each be less than $5V/30\mu A = 167k\Omega$, according to *Eqs. 2.20,2.21*.

Since the quality of the voltage regulation increases with larger current flowing through R1 and R2[11], it is desirable to have the resistances as small as possible. However, these two resistors also form a connection from the positive supply to the negative supply, so they will always have a current flowing through them. So the tradeoff is between the quality of the voltage regulation and increased quiescent power consumption. A value of $6.8k\Omega$ was chosen for these resistors, since it is significantly smaller than required, and the current flowing through them will be only $0.7mA$, which is several orders of magnitude smaller than the available power output from the charge pump.

This conversion circuit can provide the necessary $-5V$ supply from the USB hub for the amplifier circuits. Since the positive supply of the USB hub is already $+5V$, it is merely passed through to provide the positive supply for the amplifier circuits. Drawing power from the USB hub, the voltages in these circuits have been limited to a potential of $\pm 5V$, which is small enough not to pose a risk to a person using the devices. Furthermore, everything can run off of the battery in a laptop, eliminating the need for wall power, making everything both more portable and safe.

2.5 Physical Enclosures and Mechanical Considerations

If the devices are to be portable and easy to use, then reliability is also crucial. The physical packaging of the devices is therefore important, as it protects them from damage and makes them easy to handle.

The two stethoscopes which are intended to go over the carotid arteries are attached to plastic boxes (*Figure 2-10*). In the each sensor's box there is the amplifier circuit for that sensor. The amplifiers are placed as close to the sensors as possible to minimize the length of wire the signal travels down before being amplified. This helps reduce radio noise compared to the signal strength, since long wires act as good antennas. There is a cable which goes from the right sensor and carries power to the

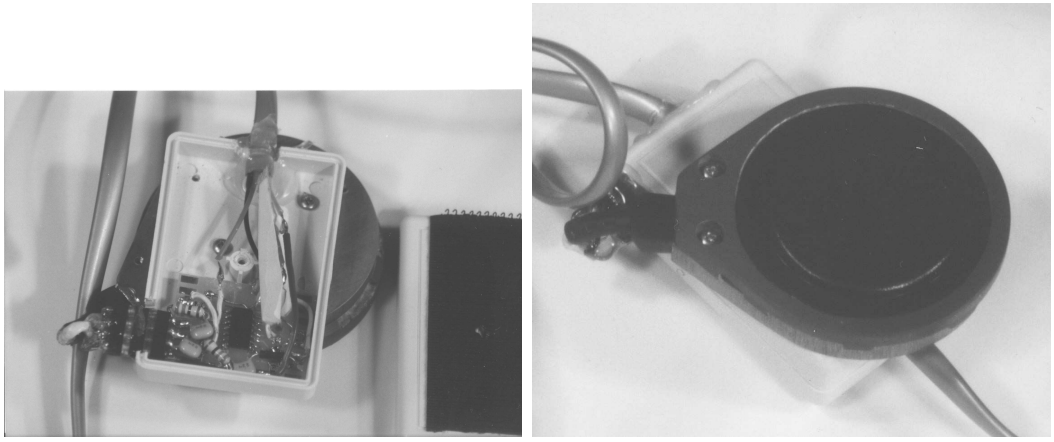


Figure 2-10: Inside (a) and outside (b) of a carotid artery stethoscope box

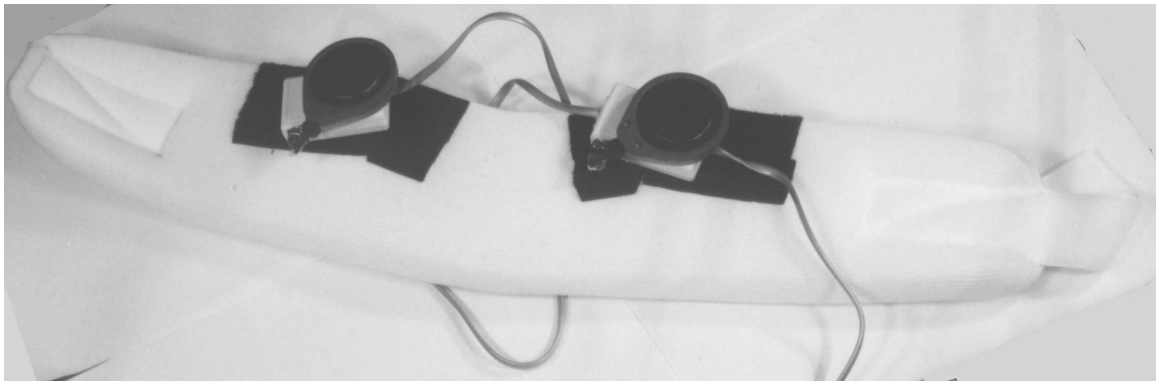


Figure 2-11: The carotid artery stethoscopes attached to the cervical collar

left sensor and the amplified signal back from the right sensor to the left sensor. The right sensor's box has the amplifier for the right side sensor and the power conversion circuit. There is a cable that goes from this box back to the USB hub and the sound card, carrying the power for the converter circuit one way and the signals from both amplifiers the other way.

The stethoscopes can be positioned on a cervical collar (a foam pad which can be fastened around the neck, originally designed for immobilizing it to aid in recovery from injuries) depending on the size and shape of the neck of the patient. This is made simpler by putting velcro on both the sensor boxes and the cervical collar (*Figure 2-11*). The cervical collar can then be fastened around the patient's neck to hold the stethoscopes in place (*Figure 2-12*). This way, there is no noise caused by movement

Figure 2-12: The carotid collar being worn

of the stethoscopes by the examiner; the collar also helps block out external room noise.

The EKG circuit is contained in another plastic box (*Figure 2-13*). Variations in electrode positioning make it necessary to have long leads, which causes difficulty putting the amplifier close to the electrodes. However, the leads are as short as possible. A cable, which carries power and the EKG signal, goes from this box to another box that contains the power conversion circuit that provides power for both the EKG amplifier and the chest stethoscope.

The chest stethoscope is contained within a plastic box with its amplifier (*Figure 2-14*). It is attached to a cable that carries power and the stethoscope signal back to the common power supply box shared with the EKG circuit. This stethoscope can either be held to the patient's chest by the examiner or by an elastic chest strap.

Wherever possible, all the sensors and components are contained within plastic boxes to protect them from damage. Any wires that leave the boxes or connectors on the edge of boxes have hot glue reinforcing their connection to the box by filling in the space to the edge of the holes in the side of the box (*Figure 2-15*). This hot glue acts as a strain relief, so if the wires get pulled or twisted they exert force on the boxes instead of the components they are connected to.

Other than the very ends of the connectors and the EKG electrodes, there are also no exposed wires. Anywhere there is a connection that does not occur in a plastic

Figure 2-13: Inside (a) and outside (b) of the EKG box

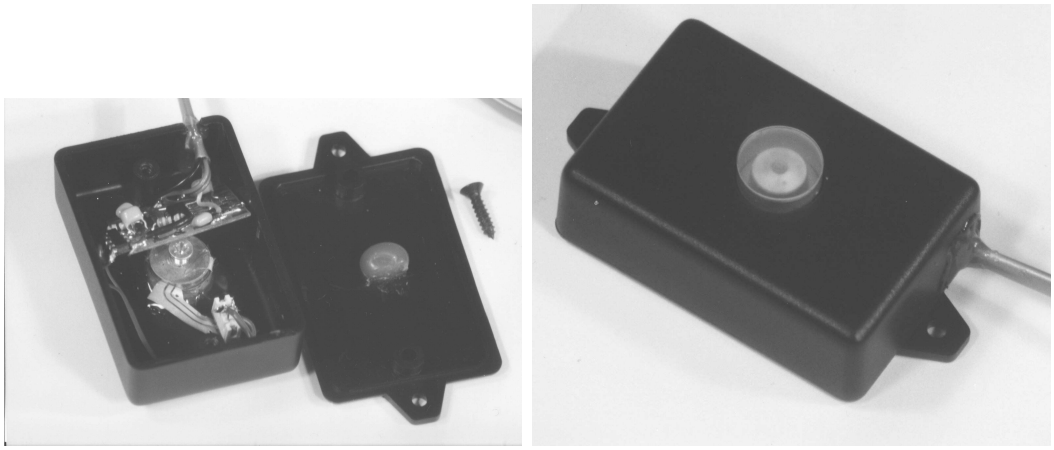


Figure 2-14: Inside (a) and outside (b) of the chest stethoscope box

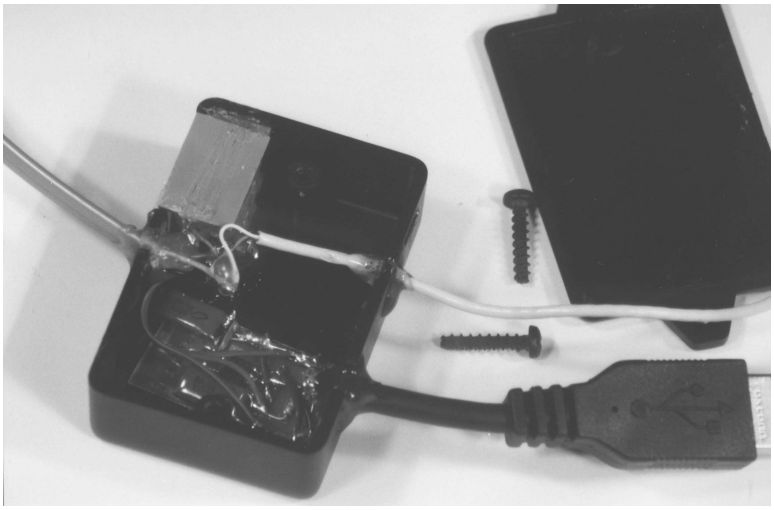


Figure 2-15: The inside of the power converter box which supplies the chest stethoscope and EKG, showing hot glue being used to reinforce outside connections.

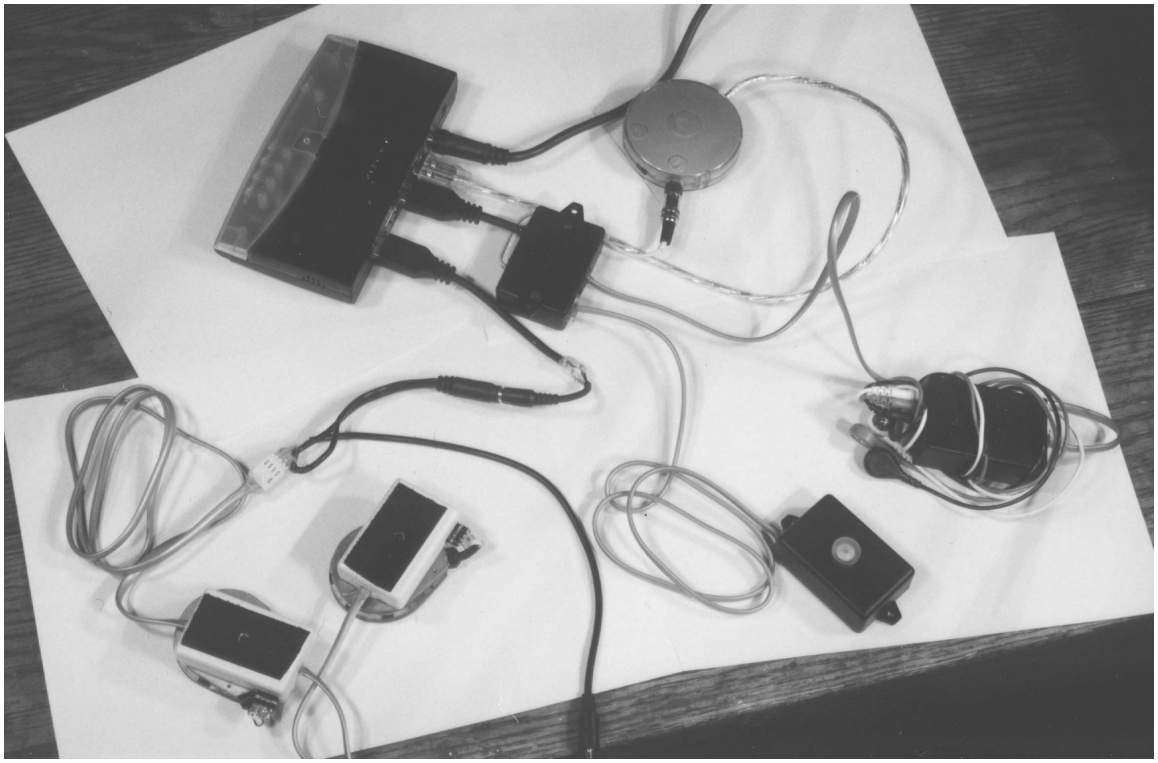


Figure 2-16: All of the sensors plugged into the USB hub as they would be for an examination.

box, it is coated with hot glue to insulate it. A picture of all of the hardware while connected is in *Figure 2-16*.

Chapter 3

Software

The software is what allows a clinician to use the hardware to make recordings of the signals related to stenosis. This software should make data collection easy and accurate. The user interface should be easy to use and capable of real-time data display.

The Carotid Collar Recorder (`ccrec`) is a program that allows the user to make recordings of the audio and EKG signals coming from the carotid collar and other sensors. Its operation is based around a simple data-handling loop, with a graphical user interface (GUI) used to control data flow. At the GUI, the program executes in one of three modes: playing, recording, and stopped. Internally, there is a fourth mode, playing live data, which is a subset of the playing mode. The software uses the GTK widget toolkit and XWindows, and executes on a system running Linux.

The following two sections describe the operation of the program from two viewpoints: from the user's point of view, and internally.

3.1 Graphical User Interface

A screen capture of the user interface in operation is shown in *Figure 3-1*. It was taken during playing of "live" data (data read from the sound card instead of a previous recording).

A "widget" is the basic building block of a GUI, referring to one area of the screen

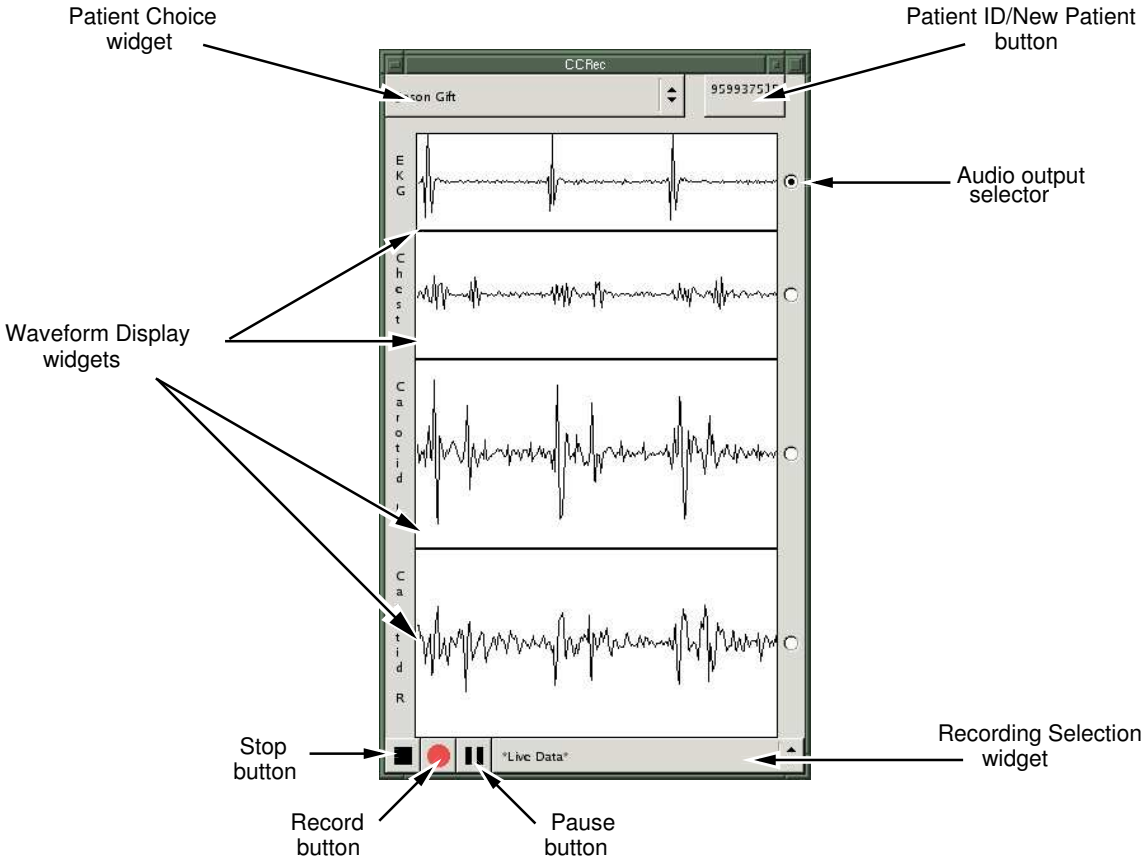


Figure 3-1: A screen capture of the user interface, showing simultaneous recording of three channels of audio data and EKG.

and how the user can interact with it. This program's widgets can be broken down into three categories: data display, data selection, and program control. The data display is done in the waveform display widgets. There is one of these widgets for each available sound card channel. Although it is an inefficient use of screen space, the waveform display widgets are stacked vertically. This allows the user to easily compare what is happening in all of the signals at the same moment in time.

While the the program is stopped or playing live data, the labels next to the waveform display widgets can be changed by clicking on them. When a recording is made, data from each channel is stored in a separate file, and the text in these labels is used as part of the filename. Consequently, since they describe the data, they cannot be changed during playback of a previous recording or during a recording.

Data selection is done through two drop-down list widgets, one at the top of the window, and the other at the bottom of the window. The patients are identified by name in the drop-down list at the top, and ID, which appears next to the patient drop-down list. The data are listed by their date and time when they were recorded, and only the recordings for the currently selected patient are listed.

There are four buttons that control the program's operation. Only the buttons that have a use in the current mode show up at the bottom of the window, to the left of the data selection widget. The first three buttons (`stop`, `play`, `record`) control which mode the program is operating in, and the fourth button, `pause`, temporarily stops the updates of the waveform display widgets.

Since the operation of the program is different depending on which mode it is in, the rest of this section is split into three sections corresponding to the operating modes.

3.1.1 Mode: Stopped

When the program is started, this is the mode it enters. While the program is in the stopped mode, there are no signals displayed in the waveform widgets. All of the data selection widgets are available, but only the `play` button is visible. The user can select historical or live data, and then push the `play` button to start viewing it,

causing the program to enter the playing mode.

3.1.2 Mode: Playing

The waveform display widgets are updated in real-time during the playing mode. The data may come from one of two sources: a previously made recording stored in files or live data.

When the data is from a previous recording, the labels on the waveform widgets are updated to reflect the labels given to the data at the time of the recording, and cannot be changed. While live data is being played, the labels can be changed to reflect which sensor each channel is capturing. With both live and pre-recorded data, the signal that is selected for audio output (by the radio button next to the waveform widget) is output to both channels of one of the sound cards, allowing the user to listen to it while watching all of the signals on the display.

The waveform display widgets are intended to give the user a quick view of the signals coming from the sensors to ensure proper operation and provide visual clues about what they are listening to. Because of limited screen space and performance issues due to the real-time display, they do not perfectly represent the signals, and as a result are meant to be used only as a diagnostic aid, and not directly for diagnosis.

In the playing mode, there are always two buttons available to the user: `stop` and `pause`. Pressing `pause` will cause the waveform widgets to hold whatever they were displaying at the time. If `pause` is pressed again, the widgets will resume their updates. During playback of recorded data it will resume from where it was paused. However live data will drop any updates that would have happened while being paused in order to remain being live.

There is one additional button, `record`, which is only available during playing of live data. When this button is pressed, the user is prompted for the type of recording, which can be selected from a list of standard options, or a custom type can be entered (*Figure 3-2*). After selection, the recording mode is entered.

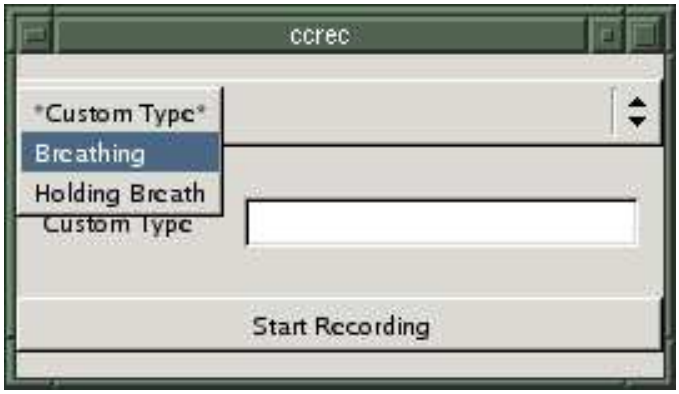


Figure 3-2: A screen capture of the recording type selection dialog box, presented when a new recording is being started.

```
0(/dev/dsp1)=L:''Carotid L'',R:''Carotid R''
1(/dev/dsp0)=L:''EKG'',R:''Chest''
```

Figure 3-3: Example contents of a channel description file

3.1.3 Mode: Recording

During recording, the program displays live data captured from the sound card in the waveform display widgets. It also saves each signal in a file on the hard disk. When the recording mode is entered, the waveform displays are cleared so that it is obvious to the user when the recording was started.

`Stop` and `pause` are the only two buttons that are visible. `Pause` has the same behavior in record mode as it does in playing mode while viewing live data. The `stop` button will cause the recording to end and the program to return to stop mode, allowing the user to listen to the recently made recording, or start another recording.

3.2 Internal Operation

3.2.1 Data Storage

A single "recording" consists of several files: one channel description file, and one data file for each signal (channel on a sound card). The different types of files are identified by their extension, which is ".info" for the channel description file, and the

```
patientdb/
├─ John Doe/
│  ├─ 2003−12−01_00:30:14.EKG
│  ├─ 2003−12−01_00:30:14.Carotid L
│  ├─ 2003−12−01_00:30:14.Carotid R
│  ├─ 2003−12−01_00:30:14.Chest
│  ├─ 2003−12−01_00:30:14.info
│  ├─ 2003−05−04_06:45:17.EKG
│  ├─ 2003−05−04_06:45:17.Carotid L
│  ├─ 2003−05−04_06:45:17.Carotid R
│  ├─ 2003−05−04_06:45:17.Chest
│  └─ 2003−05−04_06:45:17.info
└─ Jane Smith/
   ├─ 2003−10−13_18:18:42.EKG
   ├─ 2003−10−13_18:18:42.Carotid L
   ├─ 2003−10−13_18:18:42.Carotid R
   ├─ 2003−10−13_18:18:42.Chest
   └─ 2003−10−13_18:18:42.info
```

Figure 3-4: Directory structure of the patient database

label for the signal for the data files (such as ".EKG," ".Carotid L"). The file name (without extension) is a date code that consists of the year, month, day, hour, minute, and second when the recording was started.

The channel description file contains a one-line description for each sound card (*Figure 3-3*). The line starts with the name of the sound card, followed by the label for the left channel and the label for the right channel. The purpose of this file is to keep a record of which channels came from which sound cards, in case this information is important during analysis.

The data files are in WAVE format [15] (often indicated in other contexts with a ".wav" extension), which includes both the audio data and a header that describes the sample rate and sample size of the audio stream. Since this file format is a common standard, its use makes it easier to import the data into other programs for signal analysis (such as Matlab).

Recordings are stored in a simple file-based database. Each patient has their own directory in which their recordings are stored. All of the patient directories are kept in

```
''/dev/dsp0'',1,''EKG'',''Chest''
''/dev/dsp1'',1,''Carotid L'',''Carotid R''
```

Figure 3-5: Example contents of the configuration file

a directory called "patientdb." An example database, in *Figure 3-4*, has two patients: John Doe has two recordings, and Jane Smith has one. Each recording was done with two sound cards, producing four data files.

3.2.2 Initialization

When the program is first started, there are several steps of initialization that must be completed before it is ready to start interacting with the user and audio devices.

The first step is to read from the configuration file (`ccrec.conf`). A two-device configuration might look like *Figure 3-5*. Each line contains a description of a sound card to read from. The first field is device filename associated with the sound card. The second field is a boolean flag that indicates if the device needs to be initialized, the next two flags are the default labels used for the left and right channels, respectively.

This information is cached in an array of structures (`audio_device`) so that it can be looked up quickly later in the program. A total count of the number of sound cards is also kept for future reference.

Once the total number of sound cards is known, the variables that buffer audio data, the pointers to the widgets that display data, the file descriptors for the data files, and several other bookkeeping variables for each sound card channel can be allocated and initialized (which happens in `main.c:alloc_vars`). Since they're initialized based on the entries in the configuration file, it is possible to specify new sound cards and default labels without recompiling the program.

The sounds cards are initialized next (`init.c:opensound`). The device file is first opened for both reading and writing (i.e., both recording and playing). If the configuration file specifies the device should be initialized as a sound card, `sound_init` is called for the device, which uses I/O control calls to reset the device, set the sample rate, set stereo recording (two channels), set 16 bits per sample, and retrieve the

hardware buffer size of the audio device. `opensound` then allocates a buffer the same size as the hardware buffer and returns. At this point, the audio devices are ready for playback and recording.

Continuing with the initialization, if an optional visual scaling argument is specified on the command line, the scaling values are stored for use by the waveform display redrawing function. The widget toolkit's initialization function is called to set up the initial widget environment.

Next, the user interface is allocated and started (`gui.c:init_gui`). This creates all of the widgets, including the waveform display widgets based on the configuration file entries, sets an initial size for the main application window, and allocates a secondary buffer for the waveform display widgets.

Finally, the initial state of the program is set to the stopped mode, unpaused, and the patient and recording choice boxes are filled in (`main.c:sampleloop`), and the main loop is called (`main.c:sampleloop`).

3.2.3 Control Flow and Data Handling

The heart of the program is a loop that ferries data between several inputs and outputs. The inputs are the sound cards and data files. The outputs are data files, the display, and the sound cards. Which inputs and outputs are used depends on the current operating mode.

Control flow is managed through callbacks associated with the widgets in the GUI. Twice during an iteration of the loop the GUI is polled to see if there are any pending user events (button presses, etc). If they are, the callback function is called and changes the internal state of the program, then control returns to the main loop. The GUI is polled twice per iteration. It is not polled more frequently, otherwise there would be less time to move the data around, and it is not polled more slowly so that it does not respond too slowly to user input.

Data read from a sound card has the channels interleaved. The first sample read from the audio device is the left channel, and the next sample read is from the right channel, then the left, and so forth. To provide an implementation consistent with

the user interface, the audio data is separated into two streams immediately after being read.

Because sound cards sample their inputs at regular intervals but programs may not read the samples from the device regularly, they have internal buffers. The sound cards will start to fill up the buffer, and when a program reads data from the device, it will give it as many samples as are in the buffer. However, these buffers tend to be small, and if the program does not read from the buffer frequently enough the sound card will start to overwrite the oldest values, causing samples to be dropped. Therefore it is important to keep processing between reads from a sound card to a minimum so that samples are not dropped.

As a result, the block size used by the program to pass data around internally must be chosen carefully. Since different sound cards tend to have differently sized internal buffers, the devices are queried for their buffer size (`ccrec_audio.c:opensound`). However, it is not sufficient to merely use different block sizes for the different devices, since a device with a small buffer may drop samples by the time the data from a device with a larger buffer is processed. Therefore, the size of the smallest buffer is used for the program's block size.

When the user presses the play button, `cb.c:ccrec_play_cb` is called and the program enters playback mode. This function takes the current recording name from the recording selection widget and opens all of the files associated with it (if it is not live data). Since the recordings are always in the same format, it simply skips the WAVE header. It then clears the waveform display widgets and returns to the main loop.

In the main loop, if live data was selected, it is read in blocks from the sound card, and interleaved channels are separated. If pre-recorded data was selected, it is read in blocks from files. The signal which was chosen by the user to be output to the sound card is interleaved with itself (the sound card also expects data written to it to be two-channel data). After reading the data, it is averaged and drawn on the waveform display widgets. When the end of the file is reached, `ccrec_stop_cb` is called, returning the program to stopped mode.

If the user presses the record button while looking at the live data, then then the callback `cb.c:ccrec_rec_cb` is called. This function displays the recording type dialog box. `cb.c:ccrec_start_rec_cb` is called when the user has made their type selection, and the program enters recording mode. This function then creates and opens a data file for each audio device and writes the channel description file for the new recording. For each data file it creates, it also writes an initial WAVE file header. The waveform display widgets are cleared, the new recording is added to the recording selection widget, and the program returns to the main loop, now in recording mode.

When the program is in its recording mode, its data flow is the same as playing live data, with one additional step. The separated data is also written to files on the hard disk. When it does the writing, it keeps track of the total number of samples written to each file in the variable `total_len`.

The program leaves recording mode when the user presses the stop button and `cb.c:ccrec_stop_cb` is called. The final size of the data file in the WAVE header is filled in using `total_len` (wave.c:write_wave_header_size), and the data files are closed.

Because the WAVE header is versatile, it can actually become very complicated. However, since the audio data is always being stored as raw samples (i.e. no compression), there are only three fields which matter: the size of the file, the sampling rate, and the number of bytes per second. For this reason, the `write_wave_header_init` and `write_wave_header_size` functions defined in `wave.c` and `wave.h` are very simplistic, treating most of the header as a constant.

During the stopped mode, there is no data being moved around. The GUI is polled continuously in the main loop until the user causes the program to enter a different mode.

3.2.4 Updating the GUI

Unlike the recorded data, the data on the waveform displays is not a perfect representation of the samples coming from the audio devices. To do this for a single audio device would require more than 88000 refreshes per second (44100 samples per

second on each of two channels). This is certainly beyond the capabilities of current off-the-shelf hardware. Instead, samples are averaged over windows (the default is 512 samples, dropping the number of refreshes per second to $88000/512 \approx 172$), and the display is only updated once for every 16 averaging windows (further reducing the refresh rate to about 11 times per second).

There are two problems with this method of reducing display work. The first is that the averaging causes the displayed data to have a different shape from the actual data, since averaging removes high frequency content. However, the averaging allows a larger time window to be displayed at once on the screen (the waveform display shows up to 256 averaged sample points, so if the averaging window is 512 samples, the waveform display will show almost three seconds of audio data). The second problem is that only updating once every 16 averaging windows causes the scrolling of the waveform across the screen appear slightly jerky. However, the drastic reduction in the number of display updates achieved by this infrequent redrawing allows the program to handle more channels simultaneously without dropping samples.

The waveform display widget is double-buffered, so the new waveform is first drawn into a buffer in memory, then the whole buffer is copied over to the actual screen. If the waveforms were drawn directly to the screen each time, the screen would appear to flicker because of the delay between drawing the white background and the new waveform. There are two functions in `gui.c` that change what is displayed in the waveform display widget.

`ccrec_clear` draws a new background, consisting of a white rectangle the size of the widget and a black box around the edge, into the buffer and copies it over to the on-screen widget without adding a waveform. It also truncates the internal buffer of averaged sample points for the device associated with that widget.

`ccrec_redraw` also starts by drawing a new background, but it then adds the waveform which is passed to it. The waveform is passed to it as an array of two-byte numbers. These are scaled by the visual-scaling factor (which defaults to 8, or is set during initialization on the command line), and then mapped to pixel locations inside the widget. Lines are then drawn into the buffer connecting these points. Finally,

the buffer is copied over to the on-screen widget.

The use of the scaling factor allows waveforms to be "amplified" visually without changing how they are stored. This is important if the widget is being used to display the sound of blood in the carotid artery, for example, since that sound is very quiet. Since the recording volume cannot be turned up without inducing clipping when the patient swallows or coughs, the visual display of the data must be scaled separately.

Chapter 4

Preliminary Results

This chapter describes some preliminary results. This will include an initial look at some data from a normal carotid and some performance measurements.

4.1 Stethoscope Sensor Frequency Response

A simple experiment was done to test the frequency response of the stethoscope sensors. The stethoscope was placed on top of a speaker. The speaker was driven with a frequency that started low and progressively got higher in steps of 60Hz, between 0 and 12000 Hz. At each frequency, a recording was made with the stethoscope. A Fourier transform was taken of a section of this recording, and the sum of the energy in a 60Hz wide band centered around the driving frequency was noted. This was repeated with both a Meditron sensor and a Kassal sensor.

This experiment does not yield a perfect measurement of the frequency response of the sensors and amplifiers. The output power of the speaker and also the transmission of the sound energy from the speaker to the sensor will vary depending on the frequency. The sensor itself will have a characteristic frequency response which will also affect the results, so these frequency responses cannot be compared to the theoretical frequency responses in *Chapter 2* (since those responses were only for the circuits, not the sensors). Furthermore, because computers are not silent while running and the speakers were physically attached to the computer, it is difficult to conduct the

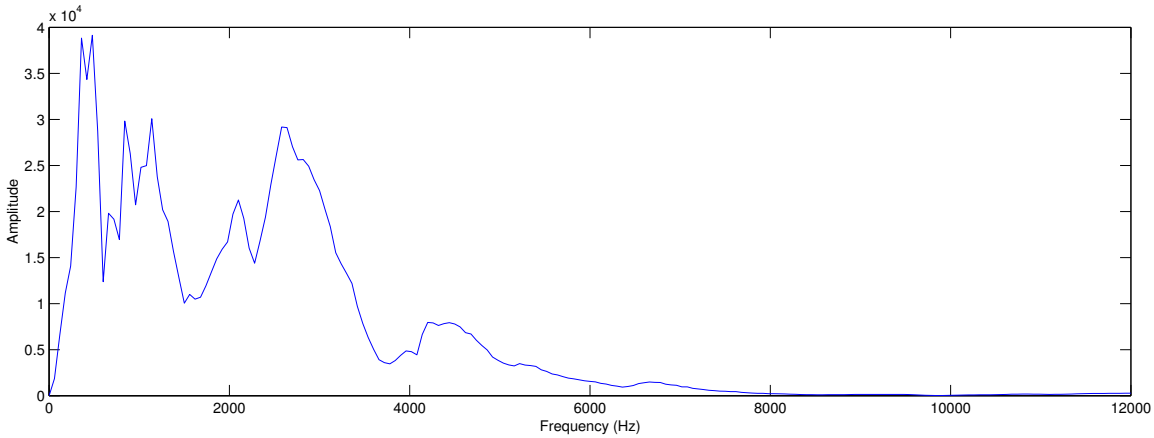


Figure 4-1: Measured frequency response of the stethoscope amplifier circuit and sensor, used with the Meditron sensor

experiment in an entirely silent room to prevent external noise from influencing the results. However, the experiment does indicate whether or not the sensor has a response at each of the tested frequencies. Additionally, since the speakers, circuit, and method was the same for both sensors, the two sensors can be compared to each other.

The frequency response for the Meditron sensor is plotted in *Figure 4-1*. The response for the Kassal sensor is in *Figure 4-2*. As can be seen from the graphs, both sensors respond up to frequencies of at least 6000Hz. In an older carotid bruit study, the "break frequency" of the bruits (the frequency of maximum energy) was found in the range of 70 to 640 Hz[5]. Therefore, since both sensors along with the circuit have relatively large responses up to 4000Hz, they should be sufficient to capture bruit sounds. It is also interesting to note that the Meditron sensor has a second peak in its response, between 2000 and 4000Hz, compared to the Kassal sensor. Because of this and the generally smoother shape of the Kassal sensor's frequency response, it is probably a better sensor, and is the sensor used for the subtler sounds of the carotids in the carotid collar. While both sensors could be used to capture bruits, data recorded by one should not be compared directly with data recorded by the other without correcting for the difference in the frequency responses.

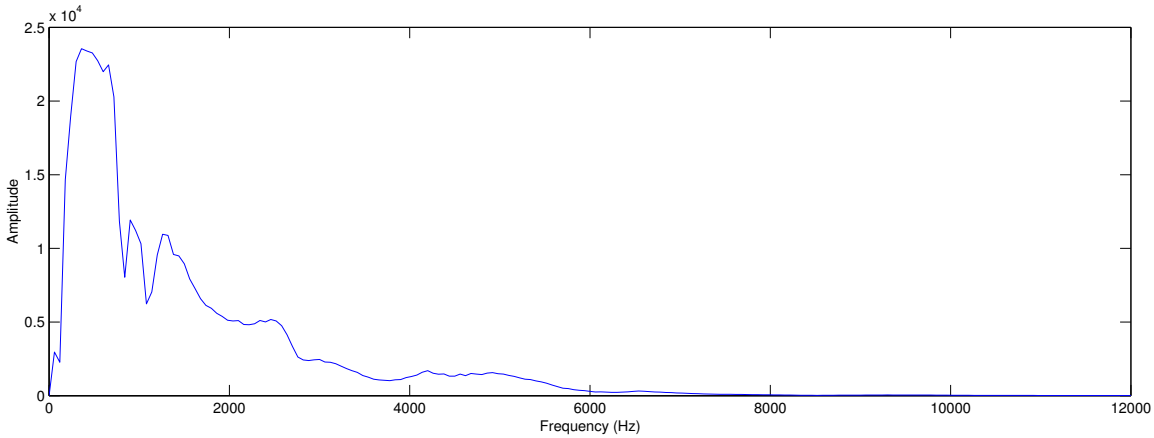


Figure 4-2: Measured frequency response of the stethoscope amplifier circuit and sensor, used with the Kassal sensor

4.2 Sample Recordings

Some sample recordings and frequency analyses are presented here. The recordings were of the author, on August 22, 2003. All four sensors (three stethoscopes and one EKG) were used: the labels of "Left Carotid Artery" and "Right Carotid Artery" correspond to the stethoscopes over the left and right carotid arteries, and the label "Chest" is the stethoscope over the chest.

4.2.1 EKG and noise filtering

A section of the EKG recording appears in *Figure 4-3*. It clearly shows the R waves (the tall, positive spikes), in addition to the Q and S waves (the shorter, negative spikes before and after the R wave). The 60Hz noise exists (the fuzzy-looking, low-amplitude noise), but is small relative to the QRS complex. The noise does not obscure the location of the beat in the EKG, but it is still important to consider ways to remove 60Hz noise, since it is a common problem with any signal that is recorded and may be worse depending on the environment.

In order to try to reduce the 60Hz noise, two techniques were used. The first was a simple notch filter that filtered out a narrow band around 60Hz. The notch filter had a very narrow stop band (about 58Hz to 63Hz) because the 60Hz noise occurs in the

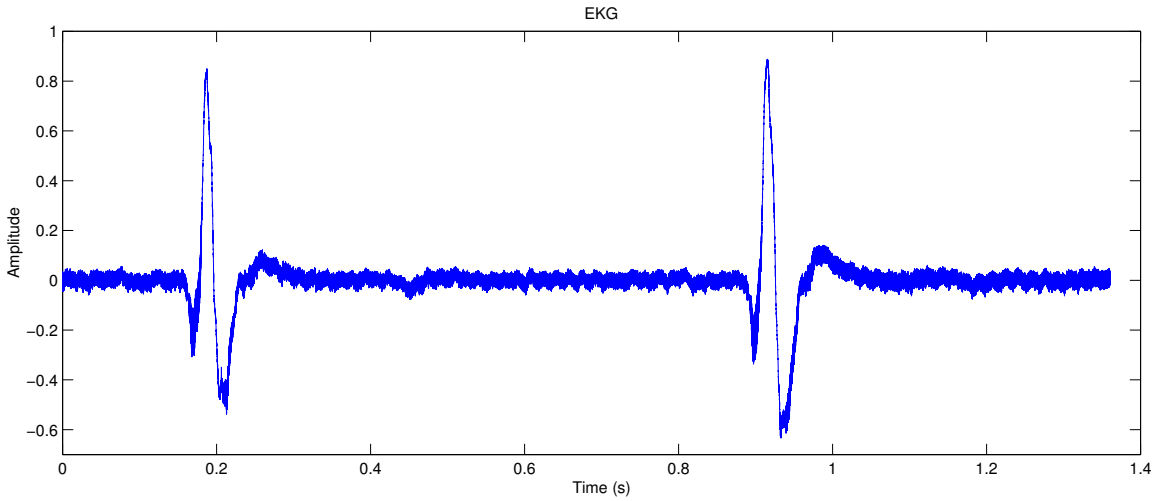


Figure 4-3: EKG recording

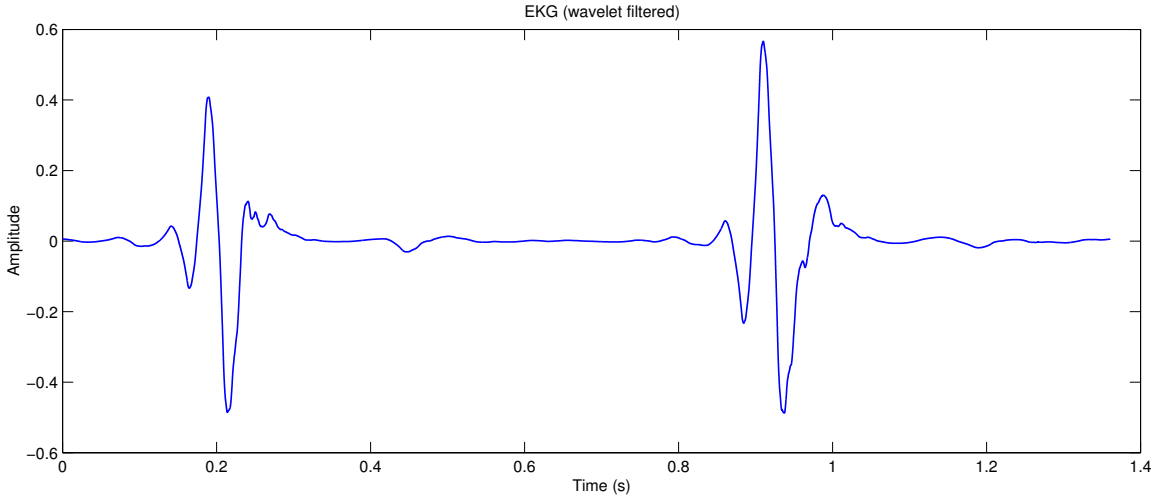


Figure 4-4: EKG recording, after being filtered using wavelet analysis

```
[wc,l] = wavedec(y3Short,10,'db4');
ra10 = ( wrcoef('a',wc,l,'db4',10) );
```

Figure 4-5: Matlab code used to filter the EKG signal

middle of the frequency range considered important for EKG. As a result, the filter was highly non-linear in phase. This non-linear phase distorted both the amplitude and the timing of the QRS complex. This is not beneficial when using the EKG for a time basis.

The wavelet-based filter did not distort the timing, but it did change the amplitude. The positive peaks were reduced, while the negative peaks remain at approximately the same amplitude. This is probably an effect of the particular wavelet used, and suggests that a different wavelet should be used in future filtering.

The other technique, using wavelet analysis, caused a small amount of amplitude distortion, but left the timing the same. Wavelet analysis is similar to Fourier analysis, but instead of decomposing a signal into infinitely long sine and cosine waves of various frequencies, it is decomposed into "wavelets," which are arbitrary, short waveforms. As a result of the finite time duration of the wavelets the morphology, or shape of the signal, is preserved[17]. A tenth order wavelet decomposition was done using the "db4" wavelet defined in Matlab's wavelet toolkit (*Figure 4-5*)[17]. The signal was then reconstructed by using approximation of the 10th level wavelet[17], and the result is shown in *Figure 4-4*.

4.2.2 Carotid Artery

Two short time sequences from the left carotid artery are shown in *Figures 4-6* and *4-7*. The large pulses in both are the beginning of the pressure waves from the heartbeat. These two recordings show the difference between breathing and holding breath. The sounds caused by breathing tend to be at many different high frequencies because of how air flows through the neck, bronchial tube, and the smaller passageways in the lungs.

An FFT (fast Fourier transform) was taken of this recording during the S1 of two

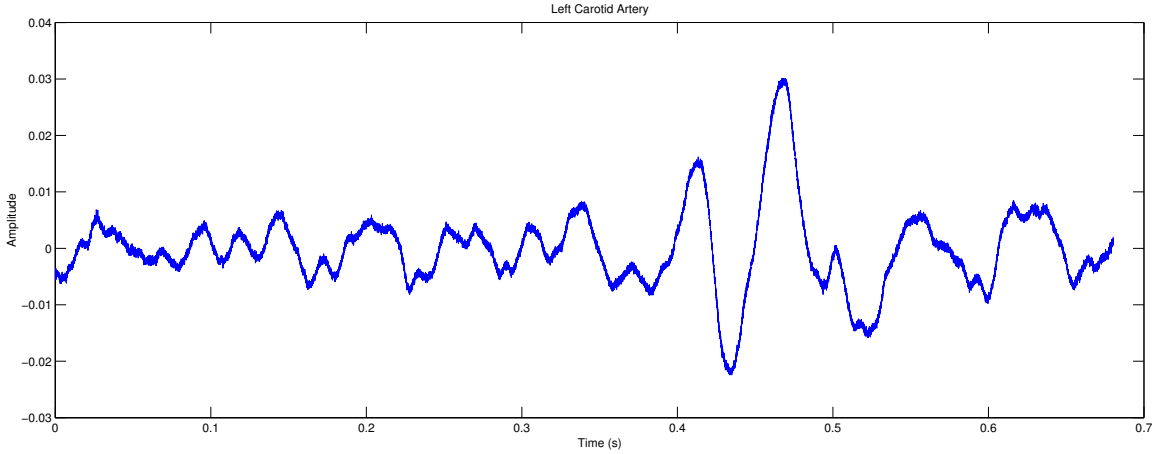


Figure 4-6: Left Carotid Artery, holding breath

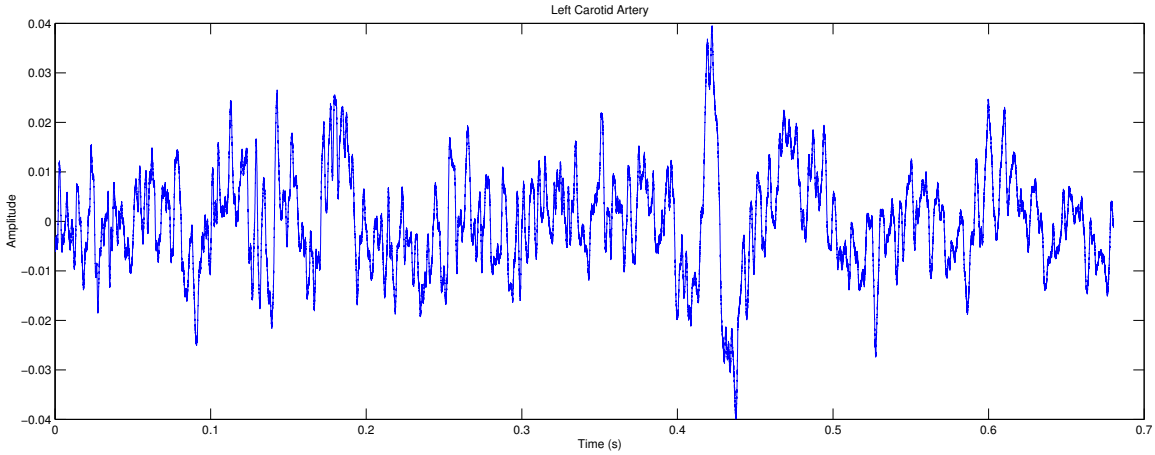


Figure 4-7: Left Carotid Artery, while breathing

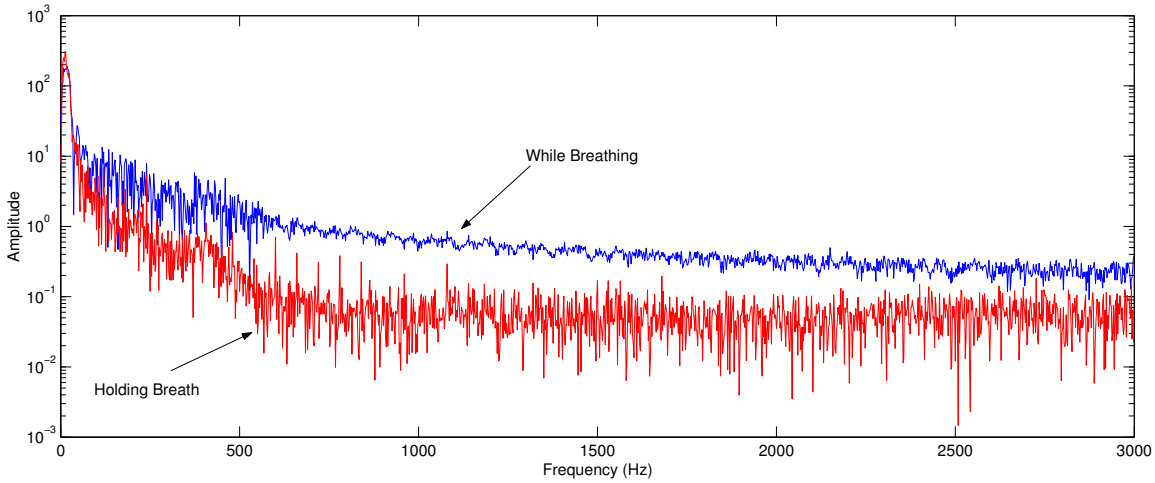


Figure 4-8: FFT of a normal during S1, while breathing and while holding breath

different heartbeats: during a breath hold and while breathing. As expected, the one during breathing has a larger high-frequency content, caused by the breathing sounds.

4.2.3 Four signals: Carotid Arteries and Chest stethoscopes, and EKG

In *Figure 4-9* all four signals are shown, arranged vertically in the same fashion as the real-time display of the software. The signals were all filtered using a wavelet method similar to the one used for the EKG. The same 10th order decomposition was done using Matlab. Because the left and right carotid sensors had different amounts of noise, they were filtered with different wavelet filters. The left carotid was reconstructed using an approximation of the 6th level wavelet, the right carotid with the 8th level wavelet. The EKG was done with a 9th level wavelet this time, and the chest stethoscope was reconstructed using the 6th level wavelet. Compared to the unfiltered signals (*Figure 4-10*), it is much easier to see the correlation between the signals without the noise.

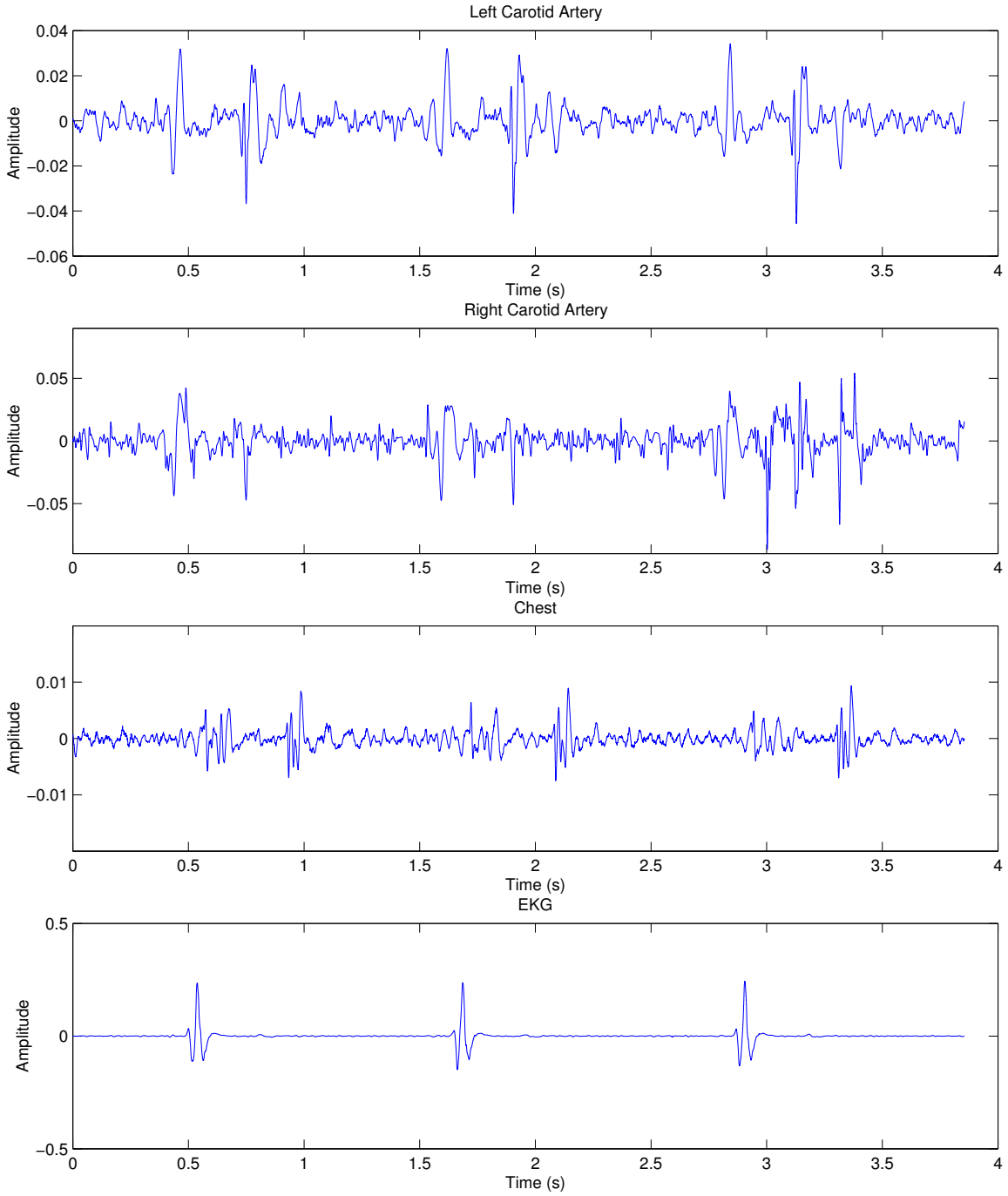


Figure 4-9: All four signals, stacked vertically to emphasize their relationships and filtered to remove some noise

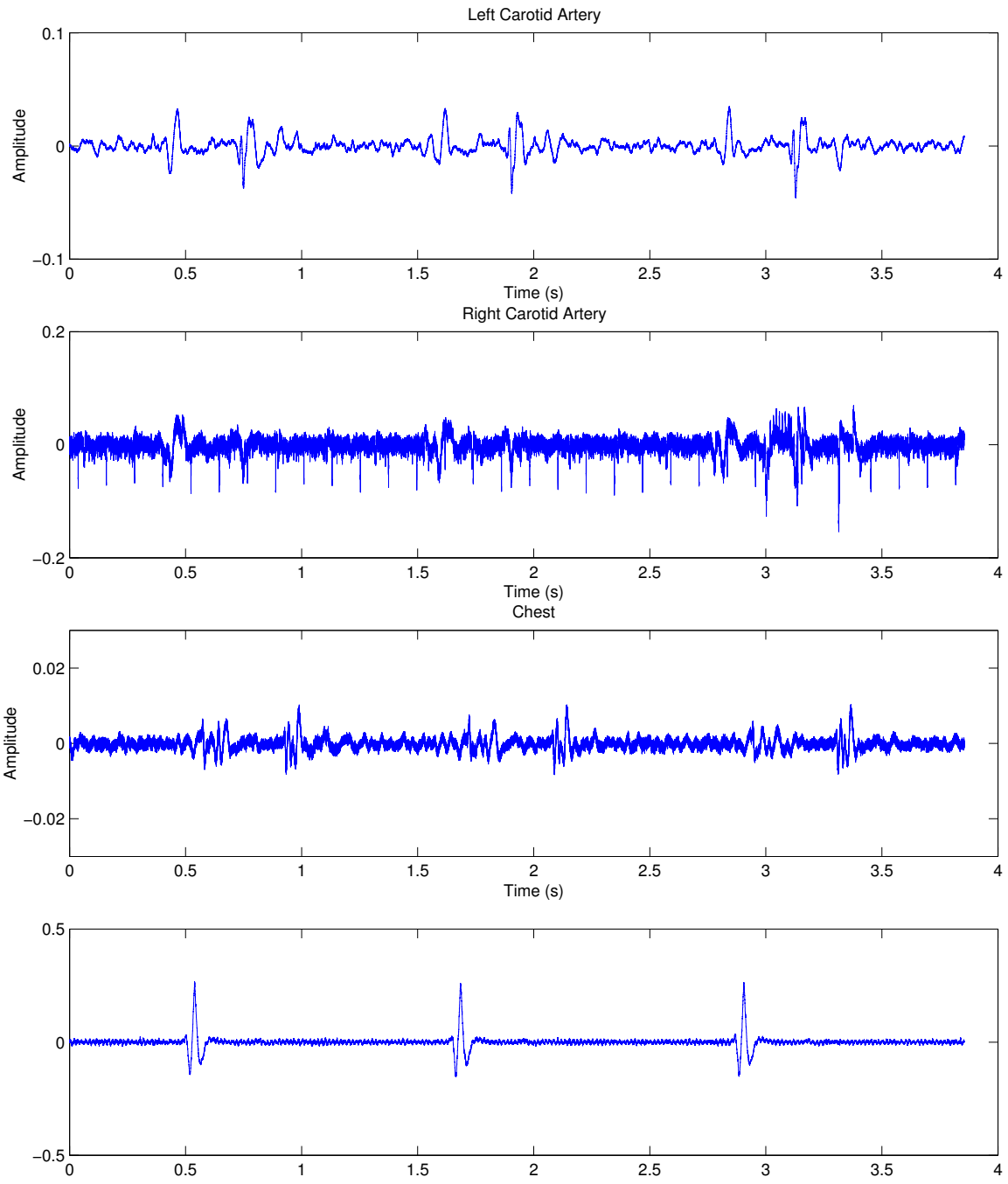


Figure 4-10: All four signals, unfiltered

4.3 Software Performance

Several delays were measured in the main loop to form an estimate of the performance of the software. The system time (measured in milliseconds) at the start and end of the loop, as well as before the display update, was recorded. Because the time for a display update was close to the smallest unit of time measured, a second experiment was done to form a better estimate of the display refresh time. About 70 repeated display updates were done in succession, and the start and end times were measured. The average refresh time was calculated from these measurements.

The time to complete one entire iteration of the loop, which includes waiting for data to become available, reading, processing, and displaying that data was measured as $23ms$.

The refresh time (the time it takes to redraw a waveform display widget) of $2.08ms$ for a single channel of a single sound card was measured on an IBM Thinkpad A21p (PIII 850Mhz laptop with an ATI Rage Mobility M3 video card). The time spent waiting for the data was $10ms$. After the waiting time and displaying time is subtracted, the time spent reading and processing the data was $8.84ms$.

If the buffer for each sound card is 4096 bytes, this means that at a sample rate of 44100 samples per second there is about $2048/44100 = 92.9ms$ during which reading, processing, and displaying can be done. If these three tasks take longer than $92.9ms$, then samples will be dropped. Since these three tasks take about $13ms$ to complete for two channels of data, the maximum number of channels which can be recorded from simultaneously is $(92.9ms/13ms) * 2 = 14$ channels.

Fourteen channels would require seven sounds cards, and a throughput of roughly 10Mb per second to transfer the data. The USB bus has a maximum throughput of 12Mb per second under revision 1.1[4], which is greater than the throughput needed for the data alone. However, this does not account for other data packet information that needs to be sent down the USB bus (which depends on the exact USB sound cards used), so the actual number of sound cards may be limited by the USB bus.

If a faster computer is used, the reading, processing, and display times will go

down, while the maximum time between reads will remain constant. Because a faster computer would support more devices, if USB sound cards are used then they will have to be USB 2.0 compatible, which has a higher throughput to handle the additional devices.

Chapter 5

Evaluation

We set out to create a carotid collar device that could aid in the detection of carotid artery stenosis. There were four goals set down in the introduction, and the strengths, weaknesses of the carotid collar, along with suggested solutions for those weaknesses, will be discussed as they relate to each goal. This will be followed by a description of how the device is intended to be used to make a recording. Other applications of the device or parts of the device are also included.

5.1 Goals

5.1.1 Safety of the Examinee

Steps were taken to make sure that the device does not pose a threat to the patient. Except for the EKG circuit, there are no exposed connections. Furthermore, all of the devices are low-powered and run off of the USB bus's power, so as long as the computer is running from a battery, there is no risk of the examinee being accidentally exposed to the 120V power. The EKG circuit was carefully designed so that the patient is isolated from the circuit as much as possible, since making direct electrical contact with the patient is unavoidable. There are no wires that can get wrapped all the way around the patient's neck, since the two stethoscopes on the cervical collar are connected together and only a single wire travels back to the computer. The cervical

collar itself is soft and should not pose a choking hazard.

If there this device needs to be used with a desktop computer or a laptop while plugged into 120V power, then more circuitry will need to be added to insure that the patient remains isolated from the 120V power. This circuitry was not considered in this project. However, there are other medical devices (such as the Welch-Allyn stethoscope) that have been approved for use while plugged into a higher voltage supply, so it is possible to achieve sufficient isolation.

5.1.2 Accurately Record Stenosis Sounds

As explained in *Chapter 2* and demonstrated in *Chapter 4*, the stethoscope sensors and amplifiers should be able to accurately record the sounds of a bruit. The stethoscope takes several measures to reduce noise: placing the amplifier as close as possible to the sensor, using high-pass filters to get rid of DC bias, and taking advantage of the properties of the sensor to do noise cancellation. Care was taken in the design to ensure that the stethoscope has a frequency response that does not greatly attenuate any of the frequencies believed to be important in detecting a bruit. However, data needs to be collected and analyzed to determine how well the device actually records bruits.

It is likely that, during the course of exploring the data collected from the initial study there will be other hardware improvements that are necessary. For example, it might be discovered that three sensors are needed on each side of the neck, and they need to be relatively close to each other. In that case, a small sensor, such as the one made by Meditron, would be more appropriate.

It would probably be advantageous to put all three sensors inside the same enclosure, so that only one conglomerate sensor need be placed on each side of the neck, simplifying setup. Because the relative positions of the three sensors would be known, it should be possible to do better filtering or possibly even three-dimensional location of the source of the sound within the body. However, if the sensors are fixed in position relative to each other, they can no longer be adjusted for different neck sizes or abnormal anatomy.

Although the circuits were designed to reduce noise, there is still noise present. As *Figure 4-10* shows, there is significantly more noise in the right carotid artery signal. The right stethoscope sensor's box is the one where the power conversion circuit is, and it is possible that it is emitting radio noise.

Furthermore, because the wire carrying the signal from the left carotid stethoscope goes through the right carotid stethoscope's box, the lengths of the wires are different, so they could pick up different amounts and frequencies of noise (the wires act like antennas, and the frequencies of noise they will pick up depend on their length). The power converter circuit could be moved to another box, placed closer to the computer and farther from the sensors, with a wire going to each sensor, solving both of these issues. However, since it is possible to remove much of the noise in software, these changes are probably not required in order to accurately record bruits.

Because we do not yet know which signals are relevant to stenosis detection, the effect of this filtering is not yet known. It may be that the presence of 60Hz noise is irrelevant, or that wavelet filtering introduces distortions which hinder, rather than help detection. The type and degree of filtering used will depend on future research.

Beyond the acoustic sensors, the EKG circuit is also adequate to pick off the QRS waves, and even the T wave if the noise is small enough. It does attenuate very low frequencies which are sometimes useful for diagnosis of heart conditions. However, this EKG circuit is not intended to replace a highly accurate, multi-lead EKG, it is only intended to be used for timing information.

5.1.3 Easy set up and use

Set-up involves plugging the devices in and attaching the sensors to the patient. We do not know how accurately the sensors need to be placed. If they have to be placed carefully, this step of the setup may not be easy.

Other than the sensor attachment, setup is simple and straightforward. Recording can be started as soon as the software is running and the correct patient is selected. There is no calibration process to go through. The software provides a simple interface that allows the examiner to quickly make recordings, review old recordings, and check

device operation in real time.

5.1.4 Physical Portability and Robustness

All of the circuits are enclosed in boxes, and the sensors attached to those boxes. Wire connections are insulated with hot glue to prevent damage to the circuits in case the wires are pulled or twisted. All of the sensors, the USB hub, and all the related wires can easily fit into a small bag or box, so it is easy to move the device from place to place. While it is expected that it will hold up during reasonable usage, the various stresses it will experience will not be known until it is actually used. If necessary, custom boxes could be made to enclose the circuits. This would provide better protection and make the device more physically compact.

5.2 Intended Method of Use

This section describes how to set up and use the carotid collar to do an examination. The first step is to reconnect the hardware if it was not left connected between examinations. The USB sound cards would be plugged into the USB hub and the hub into the computer. The sensor circuits would be connected to the USB hub for power and their outputs to the sound cards.

Using the software, the patient would either be selected (if the patient already exists in the database), or new patient button would be pressed and his or her name and ID entered. The examiner would then place the carotid stethoscopes on the cervical collar where he or she thinks the best signal can be recorded, and fasten the collar around the patient's neck.

Then, in the software, the examiner would select "Live Data" and press the play button to check that there is adequate signal coming from the carotid stethoscopes. If it is inadequate, the examiner can ask the patient to talk or cough to make sure the carotid stethoscopes are working. If the signal while the patient is quiet is not sufficient or does not appear to be the carotids, then the collar can be removed and the sensors repositioned.

Also, the labels on the waveforms need to be checked. The examiner should tap on the right and left sensors to make sure they are correctly labeled. Additionally, he or she can do other quick checks to determine which branch of the carotids is the loudest being captured. One such test is to tap on the temples of the patient, which should be heard in the external branche but not the others.

After the carotid sensors are properly positioned, the EKG and Chest stethoscopes can be attached. These should also be checked to ensure that an adequate signal is coming from them. The QRS complex should be clearly visible on the EKG waveform display, and the chest stethoscope should have a clear heart beat that is synchronized to the EKG.

After all of the sensors are positioned, the examiner would press the record button and select the type of recording (such as "holding breathing"), then ask the patient to hold their breath, and press the start record button. About 30 seconds later (or until the patient stops holding his or her breath), he or she would push the stop button. Then the examiner would instruct the patient to do the next maneuver and record that as well.

5.3 Testing and Algorithm Development

Now that the carotid collar has been built and some initial testing done, the next step in the development of automated carotid stenosis detection is to use it in a medical environment and collect data. This will test the device's ability to survive the wear of clinical use, whether or not it accurately records bruits, and if it actually is easy to set up and use.

Because it is not known at this time exactly what data needs to be collected in order to create an automated algorithm, initial data collection will occur in a small pilot study. Approximately ten patients, with a variety of different stenoses, will be asked if they want to participate after having a doppler ultrasound exam.

After the standard doppler exam is completed, an extra thirty seconds of doppler ultrasound will be recorded over the carotid artery. The diagnosis will also be noted

and all personal patient identification (such as names) will be removed for privacy.

The extra doppler ultrasound recordings will be followed by a few minutes of recordings of the stethoscopes and EKG using the carotid collar. Several different positions and maneuvers will be recorded for short periods (about thirty seconds each). These include having the patient laying supine, while the patient is breathing, while the patient is holding his or her breath in, while the patient holds his or her breath out, and so forth.

This data and the diagnosis will be used to begin design of an algorithm to detect stenosis. It is possible that more or different data will be needed while working on the algorithm. In that case, because of the modular design, additional sensors and amplifier circuits can added to the device and another small study will be done to collect more data, allowing algorithm design to continue. This iterative design can be repeated until a robust algorithm has been created.

Once the automated detection algorithm is finished, it will not require an expert to use the device. This would make it a useful screening device that can be used at a primary care physician's office.

5.4 Other Applications

The device described in this thesis was designed to aid in detecting carotid artery stenosis. However, because of its modular design and simplicity there are also applications in other areas as well.

Another medical area, called dysphagia, is the study of swallowing and related disorders. The neck stethoscopes and cervical collar are already well suited to being used to record swallowing sounds. It might be necessary to decrease the amplification of the stethoscope sensors, however, since swallowing is louder than bruits.

Similarly, the cervical collar and stethoscopes can be used with very little, if any, modifications in any number of other neck-related medical research. Coughing, vocalization, and breathing are all possibilities.

A different sensor with other physical coupling properties can also be used with

Table 5.1: Stethoscope amplifier parts cost (priced at Digikey.com)

Part	Cost	Quantity	Total
Surfacemount protoboard	2.04	1	2.04
Plastic box	2.80	1	2.80
$82k\Omega$ resistor	0.144	6	0.864
$39k\Omega$ resistor	0.144	1	0.144
$2.2M\Omega$ resistor	0.054	2	0.108
$0.22\mu A$ capacitor	0.48	4	1.92
LM324 op-amp	0.46	1	0.46
		Total:	$8.34

Table 5.2: Power conversion circuit parts cost (priced at Digikey.com)

Part	Cost	Quantity	Total
Surfacemount protoboard	1.60	1	1.60
Plastic box	2.80	1	2.80
$6.8k\Omega$ resistor	0.095	2	0.190
$22\mu A$ capacitor	0.46	2	0.92
$4.7\mu A$ capacitor	0.46	1	0.46
MAX889 charge pump (priced at shop.maxim-ic.com)	3.90	1	3.90
		Total:	$9.87

this amplifier to make recordings. This would not be limited to medical applications, but could include mechanical engineering, general acoustic signal processing research, or any number of other fields where recording quiet sounds using a sensor with physical contact would be useful.

5.5 Summary

Beyond the device and supporting software, this thesis describes a design methodology classified by its sparse use of custom hardware. Other than the sensors and amplification circuits, standard computer equipment was used, such as a sound card as an analog to digital converter. By designing the device to work with normal, off-the-shelf equipment the hardware can be tailored to the needs of the research. If a more accurate analog to digital converter is needed, either a better sound card or a dedicated data acquisition card could be substituted. If very processor-intensive algorithms are required, then a faster computer can be used.

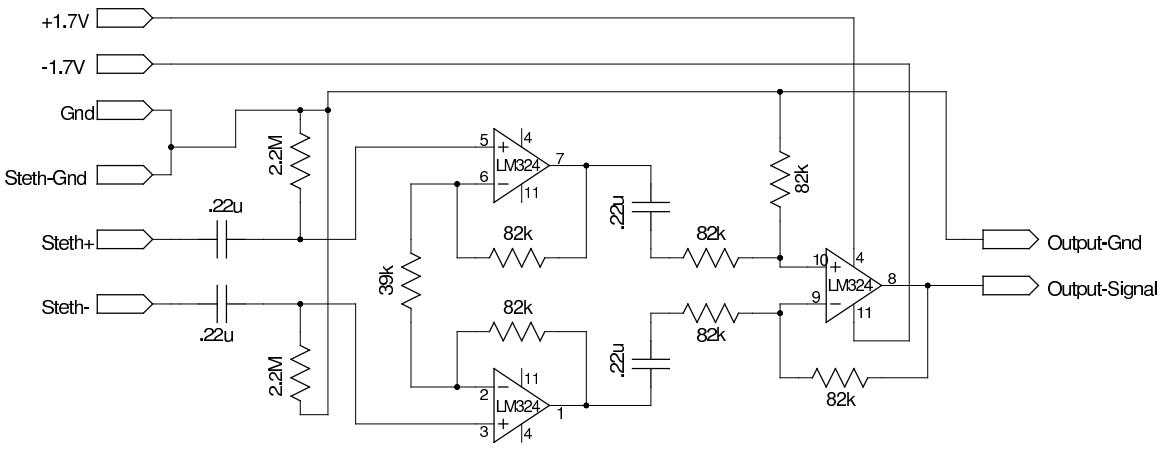
The cost of the parts used in the stethoscope amplifier (not including the wire or the sensor itself) and the power conversion circuit is shown in *Table 5.1* and *Table 5.2*, respectively. The total for a single stethoscope would be less than $20, plus the cost of the sensor. Because piezoelectric film sensors are widely available with different sizes and properties and over a wide price range, using this circuit can be an easy and cheap alternative to buying a pre-built electronic stethoscope.

This thesis has described a device that has been designed to further research into the detection of carotid artery stenosis, while keeping the possibilities for future development and research as open as possible.

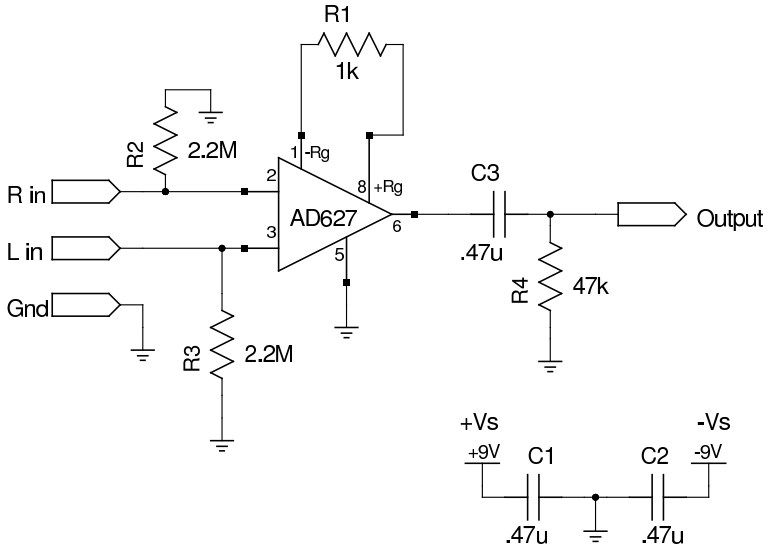
Appendix A

Schematics

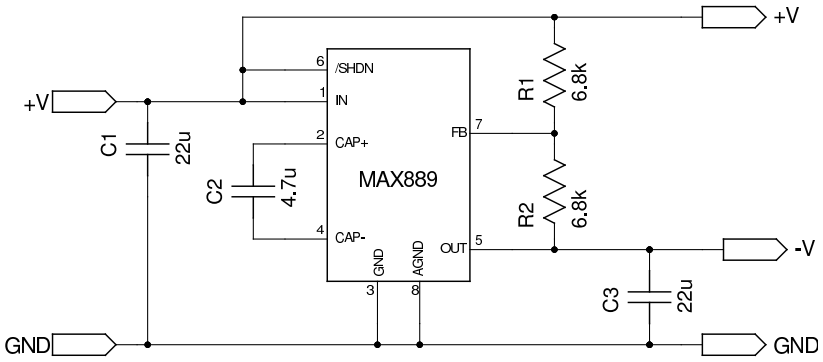
A.1 Stethoscope Amplifier Circuit



A.2 EKG Amplifier Circuit



A.3 Power Conversion Circuit



Appendix B

Code

B.1 cb.h

```
void recording_name_cb(GtkWidget *widget, char *recording_name_tmp);
void patient_name_cb(GtkWidget *widget, char *patient_name_tmp);
void ccrec_rec_cb(GtkWidget *widget, gpointer *arg);
void ccrec_play_cb(GtkWidget *widget, gpointer *arg);
void ccrec_stop_cb(GtkWidget *widget, gpointer *arg);
void ccrec_create_patient_cb(GtkWidget *widget, gpointer *arg);
void ccrec_start_new_patient_cb(GtkWidget *widget, gpointer *arg);
void rec_type_cb(GtkWidget *widget, char *arg);
void ccrec_start_rec_cb(GtkWidget *widget, gpointer *arg);
void ccrec_pause_cb(GtkWidget *widget, gpointer *arg);
void ccrec_label_change_cb(GtkWidget *widget, int dev);
void ccrec_lc_done_cb(GtkWidget *widget, gpointer arg);
void ccrec_audioselect_cb(GtkWidget *widget, int as);
```

B.2 cb.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>

#include <gtk/gtk.h>
#include <gpe/init.h>

#include "ccrec_audio.h"
#include "main.h"
#include "cb.h"
#include "gui.h"
#include "wave.h"
```

```
char *rec_type;
int label_dev;
int label_chan;


/* patient_name_cb() is called when somebody changes which patient is selected
   in the patient name widget. when this happens, we need to re-fill the
   recording names widget. */
void patient_name_cb(GtkWidget *widget, char *arg)
{
  FILE *id;
  DIR *patient_dir;
  GtkWidget *menu, *item;
  struct stat stat_buf;
  struct dirent *patient_dirent;
  char filename[512];
  char filename_x[512];
  char filename_save[512];
  char tmp;
  int dev;
  int len,i;

  /* allocate a new menu for the recording names widget */
  i=0;
  menu = gtk_menu_new();

  /* cache the patient name into the temp var */
  strncpy(patient_name_tmp,arg,MAX_PATIENT_NAME);

  /* attempt to open the patient's directory in the patient database */
  strncpy(filename,"patientdb/",11);
  strncat(filename,patient_name_tmp,MAX_PATIENT_NAME);
  if((patient_dir=opendir(filename))==NULL) {
    printf("Error: Unable to open patient database for %s\n",patient_name_tmp);
    exit(1);
  }

  /* now attempt to open the ID file in the patient database */
  strncat(filename,".id",4);
  if((id=fopen(filename,"r"))==NULL) {
    printf("Error: Unable to open patient ID file for %s\n",patient_name_tmp);
    gtk_button_set_label(GTK_BUTTON(patient_id),"000000000");
  } else {
    /* opened, now read the ID and fill in the ID label */
    if((len=fread(filename,1,512,id))==0) {
      printf("Error: Unable to read patient ID file for %s\n",patient_name_tmp);
      gtk_button_set_label(GTK_BUTTON(patient_id),"000000000");
    } else {
      filename[len]='\0';
      gtk_button_set_label(GTK_BUTTON(patient_id),filename);
    }
    fclose(id);
```

```
  }

  /* add the special "recording" which is the live data */
  item = gtk_menu_item_new_with_label("*Live Data*");
  g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(recording_name_cb),
                    (gpointer) strdup("*Live Data*"));
  gtk_widget_show(item);
  gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
  /* start with the live data selected */                                    80
  recording_name_cb(NULL, "*Live Data*");
  /* Note: each recording may have multiple files, so we need to make sure
     to ignore the extra files with the same date and time codes */
  filename_save[0]='\0';
  do {
    /* get an entry from the directory */
    patient_dirent=readdir(patient_dir);
    /* if no more entries, we're done */
    if(patient_dirent==NULL)
      break;                                                               90
    /* assemble a full pathname */
    strncpy(filename,"patientdb/",11);
    strncat(filename,patient_name_tmp,MAX_PATIENT_NAME);
    strncat(filename,"/",2);
    strncat(filename,patient_dirent->d_name,strlen(patient_dirent->d_name));
    /* try to get info on the file */
    if((stat(filename,&stat_buf))!=0) {
      printf("Error: Unable to get dir entry for %s.\n",filename);
      continue;
    }                                                                      100
    /* if it's a regular file, then we'll assume it's a recording */
    if(S_ISREG(stat_buf.st_mode)) {
      /* strip off the .0, etc so we just have the date and time */
      sscanf(patient_dirent->d_name,"%[^.]%1c%d",filename_x,&tmp,&dev);
      /* if the last file we dealt with had the same date and time code,
         then just move on, otherwise, it's a new recording, so save the
         date and time code */
      if(!strncmp(filename_save,filename_x,512))
        continue;
      else                                                                 110
        strncpy(filename_save,filename_x,511);
      /* i stores how many recordings there are */
      i++;
      /* add the recording to the menu */
      item = gtk_menu_item_new_with_label(filename_x);
      g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(recording_name_cb),
                        (gpointer) strdup(filename_x));
      gtk_widget_show(item);
      gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
    }                                                                      120
  } while(patient_dirent!=NULL);
  closedir(patient_dir);
  /* give the widget its new menu */
  gtk_option_menu_set_menu(GTK_OPTION_MENU(recording_name),menu);
}
```

```
/* ccrec_expose_event() is called when one of the drawable widgets is exposed */
gint ccrec_expose_event(GtkWidget *widget, GdkEventExpose *event, GdkPixmap *drawable_b)
{
  GdkDrawable *drawable;                                                          130
  GdkGC *tmp_gc;
  guint height,width;

  g_return_val_if_fail(widget!=NULL,TRUE);
  g_return_val_if_fail(GTK_IS_DRAWING_AREA(widget),TRUE);

  if(mode == STOPPED) {
    drawable = widget->window;
    /* get the dimensions of the drawing area */
    width = widget->allocation.width;                                             140
    height = widget->allocation.height;
    /* drawing is done, so copy into the actual on-screen widget */
    tmp_gc = widget->style->white_gc;
    gdk_draw_pixmap(drawable, tmp_gc, drawable_b, 0, 0, 0, 0, width, height);
  }


  return TRUE;
}
                                                                                  150
/* ccrec_audioselect_cb() is called when the user clicks on one of the
   radio buttons to select the channel to play back through the sound
   card */
void ccrec_audioselect_cb(GtkWidget *widget, int as)
{
  if(gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget)))
    audioselection=as;
}

/* ccrec_label_change_cb() is called when the user clicks on one of                160
   the labels to change its text */
void ccrec_label_change_cb(GtkWidget *widget, int dev)
{
  /* don't allow label changes while recording or playing previously
     recorded data */
  if(mode == RECORDING || mode == PLAYING)
    return;
  label_dev=dev;
  label_chan = l_label[dev] == widget ? 0 : 1;
  /* display the label change dialog */                                           170
  gtk_widget_show_all(lc_window);
}

/* ccrec_lc_done_cb is called when the user clicks on the "set label"
   button in the label changing dialog */
void ccrec_lc_done_cb(GtkWidget *widget, gpointer arg)
{
  char *new_label;
  char tmp[256];
```

```
  int i;

  gtk_widget_hide_all(lc_window);

  new_label=strdup(gtk_entry_get_text(GTK_ENTRY(lc_entry)));

  if(label_chan == 0) {
    /* set the label name in the audio device structure */
    strncpy(audio_device[label_dev].name_l,new_label,255);
    /* we want it vertical, not horizontal: add \n's */
    for(i=0;audio_device[label_dev].name_l[i]!='\0'&&i<254;i++) {
      tmp[2*i]=audio_device[label_dev].name_l[i];
      tmp[2*i+1]='\n';
    }
    tmp[2*i]='\0';
    /* set new label */
    gtk_button_set_label(GTK_BUTTON(l_label[label_dev]),tmp);
  } else {
    /* set the label name in the audio device structure */
    strncpy(audio_device[label_dev].name_r,new_label,255);
    /* we want it vertical, not horizontal: add \n's */
    for(i=0;audio_device[label_dev].name_r[i]!='\0'&&i<254;i++) {
      tmp[2*i]=audio_device[label_dev].name_r[i];
      tmp[2*i+1]='\n';
    }
    tmp[2*i]='\0';
    /* set new label */
    gtk_button_set_label(GTK_BUTTON(r_label[label_dev]),tmp);
  }
}

/* recording_name_cb() merely caches the name of the current recording */
void recording_name_cb(GtkWidget *widget, char *arg)
{
  /* if a previous cache exists, free it */
  if(recording_name_tmp!=NULL)
    free(recording_name_tmp);
  /* copy the current recording name */
  recording_name_tmp=(char *)strdup(arg);
}

/* ccrec_pause_cb() is called when the pause button is pressed */
void ccrec_pause_cb(GtkWidget *widget, gpointer *arg)
{
  int dev;
  /* toggle the pause state */
  paused = !paused;

  /* if we're coming out of pause and are not in the playing mode, then clear
     the screen, since some display updates were dropped */
  if(!paused && mode != PLAYING)
    for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
      ccrec_clear(draw_l[dev],draw_l_b[dev],dev);
      ccrec_clear(draw_r[dev],draw_r_b[dev],dev);
```

```
    }
}

/* ccrec_stop_cb() is called in two ways: when someone clicks the stop
   button, and to reset state after a played file finishes */
void ccrec_stop_cb(GtkWidget *widget, gpointer *arg)
{                                                                                    240
  int dev;
  /* clear the displays */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    ccrec_clear(draw_l[dev],draw_l_b[dev],dev);
    ccrec_clear(draw_r[dev],draw_r_b[dev],dev);
  }

  /* if we just stopped recording, fill in the rest of the
     wave header in the file */
  if(mode == RECORDING) {                                                            250
    for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
      write_wave_header_size(datafd_l[dev],total_len_l[dev]);
      write_wave_header_size(datafd_r[dev],total_len_r[dev]);
    }
  }

  gtk_label_set_text(GTK_LABEL(rec_type_label),"");

  printf("end.\n");
                                                                                     260
  /* reset state */
  mode=STOPPED;
  paused=0;

  gtk_widget_hide(button_stop);
  gtk_widget_hide(button_rec);
  gtk_widget_hide(button_pause);
  gtk_widget_show(button_play);

  /* close the data files */                                                         270
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    if(datafd_l[dev]!=NULL) {
      fclose(datafd_l[dev]);
      datafd_l[dev]=NULL;
    }
    if(datafd_r[dev]!=NULL) {
      fclose(datafd_r[dev]);
      datafd_r[dev]=NULL;
    }
  }                                                                                  280
}

/* ccrec_rec_cb() is called when the user presses the record button */
void ccrec_rec_cb(GtkWidget *widget, gpointer *arg)
{
  /* display the new patient dialog */
  gtk_widget_show_all(rec_type_window);
```

```
}

/* rec_type_cb() is called when the user changes the recording type in                    290
   the recording type selection dialog */
void rec_type_cb(GtkWidget *widget, char *arg)
{
  if(rec_type!=NULL)
    free(rec_type);
  rec_type=strdup(arg);
}

/* ccrec_start_rec_cb() is called when the user presses the start
   recording button in the recording type selection dialog */                              300
void ccrec_start_rec_cb(GtkWidget *widget, gpointer *arg)
{
  int dev;
  char filename[256];
  char filename_x[256];
  GtkWidget *menu;
  GtkWidget *item;
  char stringbuf[128];
  time_t *timet;
  struct tm *timetm;                                                                          310
  struct stat stat_buf;
  FILE *infofd;

  if(!strcmp("*Custom Type*",rec_type)) {
    if(rec_type!=NULL)
      free(rec_type);
    rec_type=strdup(gtk_entry_get_text(GTK_ENTRY(rec_type_cust_entry)));
  }

  gtk_label_set_text(GTK_LABEL(rec_type_label),rec_type);                                     320

  /* get rid of the recording type dialog */
  gtk_widget_hide_all(rec_type_window);


  /* if we were playing, stop because we are going to record */
  if(mode==PLAYING)
    ccrec_stop_cb(NULL,NULL);

  /* assemble the path to the patient directory in the database */                            330
  strncpy(filename,"patientdb/",11);
  strncat(filename,patient_name_tmp,MAX_PATIENT_NAME);
  /* if it doesn't exist, make it */
  if((stat(filename,&stat_buf))!=0) {
    if((mkdir(filename, 0777))!=0) {
      printf("error! cannot make patient directory!\n");
      return;
    }
  } else {
    /* if it already exists, but isn't a directory, we can't continue */                      340
    if(!S_ISDIR(stat_buf.st_mode)) {
```

```
    printf("error! not a directory!\n");
    return;
  }
}

timet=(time_t *)malloc(sizeof(time_t));

/* get the current date and time */
time(timet);                                                          350
timetm = localtime(timet);
/* make a filename based on that date and time:
   year-month-day_24hour:min:sec */
strftime(stringbuf,117,"/%F_%T",timetm);
strncat(filename,stringbuf,117);

/* cycle through all the audio devices... */
for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
  /* if the data FDs were open for some reason, close them */
  if(datafd_l[dev]!=NULL) {                                           360
    write_wave_header_size(datafd_l[dev],total_len_l[dev]);
    fclose(datafd_l[dev]);
    datafd_l[dev]=NULL;
  }
  if(datafd_r[dev]!=NULL) {
    write_wave_header_size(datafd_r[dev],total_len_r[dev]);
    fclose(datafd_r[dev]);
    datafd_r[dev]=NULL;
  }
                                                                      370
  /* open up a data file for each channel of this audio device */
  snprintf(filename_x,255,"%s.%s",filename,audio_device[dev].name_l);
  printf("opening %s...\n",filename_x);
  if((datafd_l[dev] = fopen(filename_x,"w+"))==NULL) {
    printf("error! cannot open file for writing!\n");
    return;
  }

  snprintf(filename_x,255,"%s.%s",filename,audio_device[dev].name_r);
  printf("opening %s...\n",filename_x);                               380
  if((datafd_r[dev] = fopen(filename_x,"w+"))==NULL) {
    printf("error! cannot open file for writing!\n");
    return;
  }

  /* write a wave header (except for file size) */
  write_wave_header_init(datafd_l[dev],44100);
  write_wave_header_init(datafd_r[dev],44100);
}
                                                                      390
/* open an info file (used to store channel, device, and recording
   names for later use) */
strncat(filename,".info",6);
if((infofd=fopen(filename,"w"))==NULL) {
  printf("error! cannot open info file for writing!\n");
```

```
    return;
  }
  /* print the device and channel names to the file, one line per
     audio device, finish the file with the type of recording */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++)                                    400
    fprintf(infofd,"(%s)=L:\"%s\",R:\"%s\"\n",audio_device[dev].devname,
            audio_device[dev].name_l,audio_device[dev].name_r);
  fprintf(infofd,"rec_type:%s\n",rec_type);
  fclose(infofd);

  /* we've made it far enough to enter the recording state */
  mode=RECORDING;
  paused=0;

  gtk_widget_hide(button_play);                                             410
  gtk_widget_hide(button_rec);

  /* clear the displays so the user knows we've entered the recording state */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    ccrec_clear(draw_l[dev],draw_l_b[dev],dev);
    ccrec_clear(draw_r[dev],draw_r_b[dev],dev);
  }

  /* add this recording to the recording names menu */
  menu=gtk_option_menu_get_menu(GTK_OPTION_MENU(recording_name));           420
  item = gtk_menu_item_new_with_label((char *)strdup(stringbuf+1));
  g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(recording_name_cb),
                    (gpointer) strdup(stringbuf+1));
  gtk_widget_show(item);
  gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
  gtk_option_menu_set_menu(GTK_OPTION_MENU(recording_name),menu);

  free(timet);
}
                                                                            430
/* ccrec_play_cb() is called when the user presses the play button */
void ccrec_play_cb(GtkWidget *widget, gpointer *arg)
{
  FILE *infofd;
  int dev;
  int i;
  char filename[256];
  char filename_x[256];
  char tmp[256];
  char tmp_devname[256],tmp_left[256],tmp_right[256];                       440

  /* if we were recording, stop since we are going to start playing */
  if(mode==RECORDING)
    ccrec_stop_cb(NULL,NULL);

  /* if there isn't a valid recording currently selected, we can't play it */
  if(recording_name_tmp==NULL) {
    printf("NULL recording name.\n");
    return;
```

```
}

gtk_widget_show(button_pause);
gtk_widget_hide(button_play);
gtk_widget_show(button_stop);

/* if it's the special live data "recording", change modes */
if(!strcmp(recording_name_tmp,"*Live Data*")) {
  mode = PLAYING_LIVE;
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* set channel labels in UI */
    for(i=0;audio_device[dev].name_l[i]!='\0'&&i<254;i++) {
      tmp[2*i]=audio_device[dev].name_l[i];
      tmp[2*i+1]='\n';
    }
    tmp[2*i]='\0';
    gtk_button_set_label(GTK_BUTTON(l_label[dev]),tmp);
    for(i=0;audio_device[dev].name_r[i]!='\0'&&i<254;i++) {
      tmp[2*i]=audio_device[dev].name_r[i];
      tmp[2*i+1]='\n';
    }
    tmp[2*i]='\0';
    gtk_button_set_label(GTK_BUTTON(r_label[dev]),tmp);
  }
  gtk_widget_show(button_rec);
  return;
}

/* if we're playing an old recording, then no need for the record button */
gtk_widget_hide(button_rec);

/* assemble the full path to the recording */
strncpy(filename,"patientdb/",11);
strncat(filename,patient_name_tmp,MAX_PATIENT_NAME);
strncat(filename,"/",1);
strncat(filename,recording_name_tmp,117);

snprintf(filename_x,255,"%s.info",filename);
printf("opening info file. . .\n");
if((infofd = fopen(filename_x,"r"))==NULL) {
  printf("error! cannot open info file for reading!\n");
  return;
}
while(!feof(infofd)) {
  if(fscanf(infofd,"(%[^)])=L:\"%[^\"]\",R:\"%[^\"]\"\n",tmp_devname,
            tmp_left,tmp_right)==3) {
    /*printf("|%s|%s|%s|\n",tmp_devname,tmp_left,tmp_right);*/
    /* cycle through the audio devices */
    for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
      if(!strcmp(audio_device[dev].devname,tmp_devname)) {
        /* if a data file is still open for some reason, close it */
        if(datafd_l[dev]!=NULL) {
          write_wave_header_size(datafd_l[dev],total_len_l[dev]);
          fclose(datafd_l[dev]);
```

```
        datafd_l[dev]=NULL;
      }
      if(datafd_r[dev]!=NULL) {
        write_wave_header_size(datafd_r[dev],total_len_r[dev]);
        fclose(datafd_r[dev]);
        datafd_r[dev]=NULL;
      }

      /* try to open the files for this audio device */
      snprintf(filename_x,255,"%s.%s",filename,tmp_left);
      printf("opening %s...\n",filename_x);
      if((datafd_l[dev] = fopen(filename_x,"r"))==NULL) {
        printf("error! cannot open file for reading!\n");
        return;
      }

      snprintf(filename_x,255,"%s.%s",filename,tmp_right);
      printf("opening %s...\n",filename_x);
      if((datafd_r[dev] = fopen(filename_x,"r"))==NULL) {
        printf("error! cannot open file for reading!\n");
        return;
      }

      /* get past the wave header (which we don't care about) */
      fread(&tmp,1,sizeof(struct wave_header_single_chunk_pcm),datafd_l[dev]);
      fread(&tmp,1,sizeof(struct wave_header_single_chunk_pcm),datafd_r[dev]);

      /* set channel labels in UI */
      for(i=0;tmp_left!='\0'&&i<254;i++) {
        tmp[2*i]=tmp_left[i];
        tmp[2*i+1]='\n';
      }
      tmp[2*i]='\0';
      gtk_button_set_label(GTK_BUTTON(l_label[dev]),tmp);
      for(i=0;tmp_right[i]!='\0'&&i<254;i++) {
        tmp[2*i]=tmp_right[i];
        tmp[2*i+1]='\n';
      }
      tmp[2*i]='\0';
      gtk_button_set_label(GTK_BUTTON(r_label[dev]),tmp);
    }
  }
}
else {
  if(fscanf(infofd,"rec_type:%[^\n]",tmp)==1) {
    printf("%s\n",tmp);
    gtk_label_set_text(GTK_LABEL(rec_type_label),tmp);
  }
  break;
}
}


/* if we've made it this far, enter the playing state */
```

```
  mode=PLAYING;
  paused=0;

  /* clear the display so the user knows we've entered the playing state */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    ccrec_clear(draw_l[dev],draw_l_b[dev],dev);
    ccrec_clear(draw_r[dev],draw_r_b[dev],dev);
  }

}

/* ccrec_create_patient_cb() is called when the user presses the “create new patient”
   button from the new patient dialog */
void ccrec_create_patient_cb(GtkWidget *widget, gpointer *arg)
{
  GtkWidget *menu;
  GtkWidget *item;
  char filename[512];
  char *tmp_patient_name,*tmp_patient_id;
  FILE *id;
  struct stat stat_buf;
  int len;

  /* grab the patient name and ID from the text entry boxes in the dialog */
  tmp_patient_name=(char *)strdup(gtk_entry_get_text(GTK_ENTRY(patient_name_entry)));
  tmp_patient_id=(char *)strdup(gtk_entry_get_text(GTK_ENTRY(patient_id_entry)));

  /* check to see if there's a directory for the patient in the database */
  strncpy(filename,"patientdb/",11);
  strncat(filename,tmp_patient_name,MAX_PATIENT_NAME);
  if((stat(filename,&stat_buf))!=0) {
    /* if there isn't, make one */
    if((mkdir(filename, 0777))!=0) {
      printf("error! cannot make patient directory!\n");
      return;
    }
  } else {
    if(!S_ISDIR(stat_buf.st_mode)) {
      printf("error! not a directory!\n");
      return;
    }
  }

  /* open the ID file for the patient */
  strncat(filename,".id",4);
  if((id=fopen(filename,"w"))==NULL) {
    printf("Error: Unable to open patient ID file for %s\n",tmp_patient_name);
    return;
  } else {
    /* write the patient's ID to the ID file */
    if((len=fwrite(tmp_patient_id,1,strlen(tmp_patient_id),id))==0) {
      printf("Error: Unable to write patient ID file for %s\n",tmp_patient_name);
      return;
    }
```

```
    fclose(id);
  }

  /* get rid of the new patient dialog */
  gtk_widget_hide_all(patient_window);

  /* add the patient to the patient name widget */
  menu=gtk_option_menu_get_menu(GTK_OPTION_MENU(patient_name));
  item = gtk_menu_item_new_with_label(tmp_patient_name);
  g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(patient_name_cb),
                    (gpointer) tmp_patient_name);
  gtk_widget_show(item);
  gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
  gtk_option_menu_set_menu(GTK_OPTION_MENU(patient_name),menu);
}

/* ccrec_start_new_patient_cb() is called when the user presses the button
   which is displaying the patient ID. */
void ccrec_start_new_patient_cb(GtkWidget *widget, gpointer *arg)
{
  /* display the new patient dialog */
  gtk_widget_show_all(patient_window);
}
```

B.3 ccrec_audio.h

```
#include <linux/soundcard.h>

struct ad_struct
{
  char devname[256];
  int init;
  char name_l[256];
  char name_r[256];
};


extern int NUM_AUDIO_DEVICES;

#ifndef _CCREC_AUDIO
/* we're being included from a file that isn't ccrec_audio.c
   so just add reference audio_device */
extern struct ad_struct *audio_device;
#else /*_CCREC_AUDIO*/
/* we're in ccrec_audio.c, so declare the actual variable */
struct ad_struct *audio_device;
#endif /*_CCREC_AUDIO*/


/* for the dsp ioctl() calls */
#define SOUND_16BIT AFMT_S16_LE
```

```
#define SOUND_STEREO 1
#define SOUND_MONO 0

/* AVG_AMOUNT is how many samples to average for one display point,
   AVG_POW is log_2(AVG_AMOUNT) */
#define AVG_POW   9
#define AVG_AMOUNT (1<<AVG_POW)

/* these are the lists of audio device and data file FDs */
FILE **dsp;
FILE **datafd_r;
FILE **datafd_l;

int avg_samples(short *buf, int pow);
int sound_init(FILE *dsp_dev, int format, int stereo, int rate);
```

B.4 ccrec_audio.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
#define _CCREC_AUDIO
#include "ccrec_audio.h"

/*************************************************/
/* avg_samples(buf, pow) takes a buffer of 16 bit
   samples, one channel, and averages 2^pow
   samples                                      */
int avg_samples(short *buf, int pow)
{
  int i=0;
  int lim=1<<pow;
  long total=0;
  while(i<lim)
    total+=buf[i++];
  total=total>>(pow);
  return (int)total;
}


/*************************************************/
/* sound_init()
   initializes the sound card device (dsp_dev)
   for recording from or playing to.            */
int sound_init(FILE *dsp_dev, int format, int stereo, int rate)
{
  int tmp,rc,bufsize;

  /*Reset sound driver*/
  if(ioctl(fileno(dsp_dev), SNDCTL_DSP_RESET, 0) < 0)
    {
```

```
    printf("Error: Unable to reset sound driver.\n");
    return 1;
  }

/*Set sample format*/
tmp = format;
rc = ioctl(fileno(dsp_dev), SNDCTL_DSP_SETFMT, &tmp);
if(rc<0 || tmp != format)
  {
    printf("Error: Unable to get requested format.\n");
    return 1;
  }

/*Set number of channels*/
tmp = stereo; /* Stereo = 1, mono = 0 */
rc = ioctl(fileno(dsp_dev), SNDCTL_DSP_STEREO, &tmp);
if(rc<0 || tmp != stereo)
  {
    printf("Error: Unable to get requested channels.\n");
    return 1;
  }

/*Set sampling rate*/
tmp = rate;
rc = ioctl(fileno(dsp_dev), SNDCTL_DSP_SPEED, &tmp);
if(rc<0 || tmp != rate)
  {
    printf("Warning: Unable to get requested sampling rate, defaulting to %d.\n"
           , tmp);
  }

/*Get size of audio buffer*/
tmp = 0;
ioctl(fileno(dsp_dev), SNDCTL_DSP_GETBLKSIZE, &tmp);
if(tmp < 4 || tmp > 65536)
  {
    printf("Error: Unable to get valid audio buffer size.\n");
    return 1;
  }
bufsize=tmp;

/*Re-sync DSP*/
if(ioctl(fileno(dsp_dev), SNDCTL_DSP_SYNC,NULL)<0)
  {
    printf("Error: Unable to sync DSP.\n");
    return 1;
  }

/*Change to un-buffered I/O*/
/*setvbuf(dsp_dev, NULL, _IONBF, sizeof(char) * bufsize);*/
return bufsize;
}
```

B.5 gui.h

```
void init_gui(GtkWidget *top, GtkWidget *patient_top, GtkWidget *rec_type_top,
              GtkWidget *lc_top);
gint ccrec_clear(GtkWidget *widget, GdkPixmap *drawable_b, int dev);
gint ccrec_redraw(GtkWidget *widget, GdkPixmap *drawable_b, short *data,
                  int data_len, float vscale);
gint ccrec_expose_event(GtkWidget *widget, GdkEventExpose *event, GdkPixmap *drawable_b);
GtkWidget *make_patient_data(GtkWidget *window);
GtkWidget *make_top(GtkWidget *window);
GtkWidget *make_rec_type(GtkWidget *window);
void fill_patient_choice();
void fill_rec_type_choice();
```

B.6 gui.c

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/select.h>
#include <string.h>

#include <gtk/gtk.h>
#include <gpe/init.h>

#include "ccrec_audio.h"

#include "cb.h"
#include "gui.h"

#include "main.h"


void fill_rec_type_choice()
{
  FILE *rec_typefd;
  GtkWidget *menu, *item;
  char tmp[1024];

  if((rec_typefd=fopen("rec_types.conf","r"))==NULL) {
    printf("Error: Unable to open recording types configuration file!\n");
    return;
```

```
  }

  menu = gtk_menu_new();

  /* add the custom type first */
  item = gtk_menu_item_new_with_label("*Custom Type*");
  g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(rec_type_cb),
                    (gpointer) strdup("*Custom Type*"));
  gtk_widget_show(item);                                                    40
  gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);

  rec_type_cb(NULL,"*Custom Type*");

  while(!feof(rec_typefd)) {
    /* read a line from the file */
    fscanf(rec_typefd,"%[^\n]\n",tmp);
    /* add the recordin to the widget */
    item = gtk_menu_item_new_with_label(tmp);
    g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(rec_type_cb),   50
                      (gpointer) strdup(tmp));
    gtk_widget_show(item);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
  }

  gtk_option_menu_set_menu(GTK_OPTION_MENU(rec_type_option),menu);

}

                                                                            60
/* fill_patient_choice() initializes the menu for the patient name widget */
void fill_patient_choice()
{
  DIR *patient_dir;
  GtkWidget *menu, *item;
  struct stat stat_buf;
  int i;
  struct dirent *patient_dirent;
  char filename[256];
                                                                            70
  i=0;

  /* if we can't open the patient database, we can't do anything... */
  if((patient_dir=opendir("patientdb"))==NULL) {
    printf("Error: Unable to open patient database!\n");
    exit(1);
  }

  menu = gtk_menu_new();
                                                                            80
  do {
    /* get an entry from the patient database directory */
    patient_dirent=readdir(patient_dir);
    /* if it's null, we're at the end of the directory */
    if(patient_dirent==NULL)
```

```
      break;
    /* put together the full path */
    strncpy(filename,"patientdb/",11);
    strncat(filename,patient_dirent->d_name,strlen(patient_dirent->d_name));
    /* try to get info on the entry */                                          90
    if((stat(filename,&stat_buf))!=0) {
      printf("Error: Unable to get dir entry for %s.\n",filename);
      continue;
    }
    /* ignore certain entries */
    if(!strcmp(patient_dirent->d_name,".") || !strcmp(patient_dirent->d_name,"..") ||
       !strcmp(patient_dirent->d_name,"CVS"))
      continue;
    /* if it's a directory, then assume it's a patient */
    if(S_ISDIR(stat_buf.st_mode)) {                                            100
      /* i keeps track of how many patients we have */
      i++;
      /* if it's the first patient, then pretend we've selected it */
      if(i==1)
        strncpy(patient_name_tmp,patient_dirent->d_name,MAX_PATIENT_NAME);
      /* add the patient to the widget */
      item = gtk_menu_item_new_with_label(patient_dirent->d_name);
      g_signal_connect (G_OBJECT (item), "activate", G_CALLBACK(patient_name_cb),
                        (gpointer) strdup(patient_dirent->d_name));
      gtk_widget_show(item);                                                   110
      gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
    }
  } while(patient_dirent!=NULL);
  closedir(patient_dir);
  gtk_option_menu_set_menu(GTK_OPTION_MENU(patient_name),menu);

  /* pretend we've selected this patient (so that the recording names get filled in) */
  patient_name_cb(patient_name,patient_name_tmp);
}
                                                                               120
/* ccrec_clear() clears the waveform in a drawing area and pixmap buffer */
gint ccrec_clear(GtkWidget *widget, GdkPixmap *drawable_b, int dev)
{
  GtkDrawingArea *drawing_area;
  GdkDrawable *drawable;
  GdkGC *tmp_gc;
  guint height,width;

  g_return_val_if_fail(widget!=NULL,TRUE);
  g_return_val_if_fail(GTK_IS_DRAWING_AREA(widget),TRUE);                     130

  /* get the actual drawing area */
  drawing_area = GTK_DRAWING_AREA(widget);
  drawable = widget->window;
  /* get the dimensions of the drawing area */
  width = widget->allocation.width;
  height = widget->allocation.height;

  /* draw a filled, white box, clearing everything */
```

```
  tmp_gc = widget->style->white_gc;                                          140
  gdk_draw_rectangle(drawable_b, tmp_gc, TRUE, 0, 0, width-1, height-1);

  /* draw a black box around the edge */
  tmp_gc = widget->style->black_gc;
  gdk_draw_rectangle(drawable_b, tmp_gc, FALSE, 0, 0, width-1, height-1);

  /* clear the list of drawable data points for this device */
  data_len[dev]=0;

  /* drawing is done, so copy into the actual on-screen widget */              150
  gdk_draw_pixmap(drawable, tmp_gc, drawable_b, 0, 0, 0, 0, width, height);

  return TRUE;
}

/* ccrec_redraw() updates the waveform display when new data is available */
gint ccrec_redraw(GtkWidget *widget, GdkPixmap *drawable_b, short *data,
                  int datalen, float vscale)
{
  int i;                                                                     160
  float y;
  GtkDrawingArea *drawing_area;
  GdkDrawable *drawable;
  GdkGC *tmp_gc;
  guint height,width;

  /* allocate the list of points to draw */
  GdkPoint points[256];

  g_return_val_if_fail(widget!=NULL,TRUE);                                   170
  g_return_val_if_fail(GTK_IS_DRAWING_AREA(widget),TRUE);

  /* get the actual drawing area */
  drawing_area = GTK_DRAWING_AREA(widget);
  drawable = widget->window;
  width = widget->allocation.width;
  height = widget->allocation.height;

  /* clear everything in the buffer */
  tmp_gc = widget->style->white_gc;                                          180
  gdk_draw_rectangle(drawable_b, tmp_gc, TRUE, 0, 0, width-1, height-1);

  /* draw the border in the buffer */
  tmp_gc = widget->style->black_gc;
  gdk_draw_rectangle(drawable_b, tmp_gc, FALSE, 0, 0, width-1, height-1);

  /* go through all the drawable data points */
  for(i=0;i<datalen;i++) {
    /* are we putting new points on the right of the chart, or the left? */
    if(LEFT_TO_RIGHT)                                                        190
      points[i].x=width-((unsigned short)((float)i/(256.0/(float)width)));
    else
      points[i].x=(unsigned short)((float)i/(256.0/(float)width));
```

```
    /* scale the y data by the visual scaling factor */
    y=(float)data[i]*vscale;
    /* now figure out where it should go, scaling the pixel position based on
       the on-screen height of the display */
    y=y*(float)height/65536.0;
    y=y+((float)height)/2.0;                                                    200
    if(y<0.0)
      points[i].y=0; /* we're below the bottom of the display */
    else if(y>(float)height)
      points[i].y=height; /* we're above the top of the display */
    else
      points[i].y=(unsigned short)y; /* we're somewhere in the middle: good! */
  }
  /* draw the points (and connecting lines) into the buffer */
  gdk_draw_lines(drawable_b, tmp_gc, points, datalen);
                                                                                210
  /* drawing is done, so copy into the actual on-screen widget */
  gdk_draw_pixmap(drawable, tmp_gc, drawable_b, 0, 0, 0, 0, width, height);

  return TRUE;
}

/* make_patient_data() is the function that allocates, creates, and initializes the widgets
   for the new patient dialog */
GtkWidget *make_patient_data(GtkWidget *window)
{                                                                               220
  GtkWidget *vbox = gtk_vbox_new(FALSE,0);
  GtkWidget *hbox_a = gtk_hbox_new(FALSE,0);
  GtkWidget *hbox_b = gtk_hbox_new(FALSE,0);
  GtkWidget *hbox_c = gtk_hbox_new(FALSE,0);
  GtkWidget *hbox_d = gtk_hbox_new(FALSE,0);
  GtkWidget *hbox_e = gtk_hbox_new(FALSE,0);

  GtkWidget *patient_data_label = gtk_label_new("New Patient");
  GtkWidget *patient_id_entry_l = gtk_label_new("Patient ID");
  patient_id_entry = gtk_entry_new();                                           230
  GtkWidget *patient_name_entry_l = gtk_label_new("Patient Name");
  patient_name_entry = gtk_entry_new();
  GtkWidget *button_new_patient = gtk_button_new_with_label("Create Patient");

  gtk_box_pack_start(GTK_BOX(hbox_b),patient_id_entry_l,FALSE,FALSE,10);
  gtk_box_pack_start(GTK_BOX(hbox_b),patient_id_entry,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(hbox_a),patient_name_entry_l,FALSE,FALSE,10);
  gtk_box_pack_start(GTK_BOX(hbox_a),patient_name_entry,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(vbox),patient_data_label,FALSE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(vbox),hbox_a,FALSE,FALSE,0);                       240
  gtk_box_pack_start(GTK_BOX(vbox),hbox_b,FALSE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(hbox_c),hbox_d,TRUE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(hbox_c),button_new_patient,FALSE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(hbox_c),hbox_e,TRUE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(vbox),hbox_c,FALSE,FALSE,0);

  g_signal_connect(G_OBJECT(button_new_patient),"clicked",
```

```
                G_CALLBACK(ccrec_create_patient_cb),NULL);

  return vbox;                                                          250
}

/* make_rec_type() allocates, creates, and initializes the widgets for
   the recording type selection dialog */
GtkWidget *make_rec_type(GtkWidget *window)
{
  GtkWidget *vbox = gtk_vbox_new(FALSE,0);
  GtkWidget *hbox_a = gtk_hbox_new(FALSE,0);

  GtkWidget *rec_type_cust_label = gtk_label_new("Custom Type");        260
  rec_type_cust_entry = gtk_entry_new();
  rec_type_option = gtk_option_menu_new();
  GtkWidget *rec_type_button = gtk_button_new_with_label("Start Recording");

  gtk_box_pack_start(GTK_BOX(hbox_a),rec_type_cust_label,FALSE,FALSE,10);
  gtk_box_pack_start(GTK_BOX(hbox_a),rec_type_cust_entry,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(vbox),rec_type_option,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(vbox),hbox_a,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(vbox),rec_type_button,TRUE,TRUE,10);
                                                                        270
  g_signal_connect(G_OBJECT(rec_type_button),"clicked",
                G_CALLBACK(ccrec_start_rec_cb),NULL);

  return vbox;
}

/* make_label_change() allocates, creates, and initializes the widgets for
   the label changing dialog */
GtkWidget *make_label_change(GtkWidget *window)
{                                                                       280
  GtkWidget *vbox = gtk_vbox_new(FALSE,0);

  lc_entry = gtk_entry_new();
  GtkWidget *lc_button = gtk_button_new_with_label("Set Label");

  gtk_box_pack_start(GTK_BOX(vbox),lc_entry,TRUE,TRUE,10);
  gtk_box_pack_start(GTK_BOX(vbox),lc_button,TRUE,TRUE,10);

  g_signal_connect(G_OBJECT(lc_button),"clicked",
                G_CALLBACK(ccrec_lc_done_cb),NULL);                     290

  return vbox;
}

/* make_top() allocates, creates, and initializes the widgets for the
   main window */
GtkWidget *make_top(GtkWidget *window)
{
  int dev,i;
  char tmp[256];                                                        300
  GtkWidget *radio_l, *radio_r;
```

```
GtkWidget *vbox = gtk_vbox_new(FALSE,0);
GtkWidget *vbox_l = gtk_vbox_new(FALSE,0);
GtkWidget *hbox_a = gtk_hbox_new(FALSE,0);
GtkWidget *hbox_b;
GtkWidget *hbox_c = gtk_hbox_new(FALSE,0);

GtkWidget *stopimg = gtk_image_new_from_file ("stop.xpm");
button_stop = gtk_button_new();
gtk_container_add (GTK_CONTAINER (button_stop), stopimg);
GtkWidget *recimg = gtk_image_new_from_file ("rec.xpm");
button_rec = gtk_button_new();
gtk_container_add (GTK_CONTAINER (button_rec), recimg);
GtkWidget *playimg = gtk_image_new_from_file ("play.xpm");
button_play = gtk_button_new();
gtk_container_add (GTK_CONTAINER (button_play), playimg);
GtkWidget *pauseimg = gtk_image_new_from_file ("pause.xpm");
button_pause = gtk_button_new();
gtk_container_add (GTK_CONTAINER (button_pause), pauseimg);
patient_name = gtk_option_menu_new();
patient_id = gtk_button_new_with_label("");
recording_name = gtk_option_menu_new();

gtk_box_pack_start(GTK_BOX(hbox_a),patient_name,TRUE,TRUE,0);
gtk_box_pack_start(GTK_BOX(vbox_l),patient_id,TRUE,TRUE,0);
gtk_box_pack_start(GTK_BOX(hbox_a),vbox_l,FALSE,FALSE,15);

gtk_box_pack_start(GTK_BOX(vbox),hbox_a,FALSE,FALSE,0);
rec_type_label=gtk_label_new("");
gtk_box_pack_start(GTK_BOX(vbox),rec_type_label,FALSE,FALSE,0);

radio_l=radio_r=NULL;
/* for each audio device, add two labels and two windows (one for left
   channel, one for right channel) */
for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
  /* labels are done vertically (so add a carriage return after each character) */
  for(i=0;audio_device[dev].name_l[i]!='\0'&&i<254;i++) {
    tmp[2*i]=audio_device[dev].name_l[i];
    tmp[2*i+1]='\n';
  }
  tmp[2*i]='\0';
  l_label[dev] = gtk_button_new_with_label(tmp);
  gtk_button_set_relief(GTK_BUTTON(l_label[dev]),GTK_RELIEF_NONE);
  for(i=0;audio_device[dev].name_r[i]!='\0'&&i<254;i++) {
    tmp[2*i]=audio_device[dev].name_r[i];
    tmp[2*i+1]='\n';
  }
  tmp[2*i]='\0';
  r_label[dev] = gtk_button_new_with_label(tmp);
  gtk_button_set_relief(GTK_BUTTON(r_label[dev]),GTK_RELIEF_NONE);

  if(radio_l==NULL)
    radio_l = gtk_radio_button_new (NULL);
  else
    radio_l = gtk_radio_button_new_from_widget(GTK_RADIO_BUTTON(radio_r));
```

```
  radio_r = gtk_radio_button_new_from_widget (GTK_RADIO_BUTTON (radio_l));

  /* add the callback for the audio selection buttons */
  g_signal_connect(G_OBJECT(radio_l),"clicked",
                   G_CALLBACK(ccrec_audioselect_cb),(void *)dev);
  g_signal_connect(G_OBJECT(radio_r),"clicked",
                   G_CALLBACK(ccrec_audioselect_cb),(void *)dev+NUM_AUDIO_DEVICES);

  /* allocate the left and right channel drawing widgets */
  draw_l[dev] = gtk_drawing_area_new();
  draw_r[dev] = gtk_drawing_area_new();

  /* left channel on top, right channel on bottom */
  hbox_b = gtk_hbox_new(FALSE,0);
  gtk_box_pack_start(GTK_BOX(hbox_b),l_label[dev],FALSE,FALSE,3);
  gtk_box_pack_start(GTK_BOX(hbox_b),draw_l[dev],TRUE,TRUE,0);
  gtk_box_pack_start(GTK_BOX(hbox_b),radio_l,FALSE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(vbox),hbox_b,TRUE,TRUE,0);
  hbox_b = gtk_hbox_new(FALSE,0);
  gtk_box_pack_start(GTK_BOX(hbox_b),r_label[dev],FALSE,FALSE,3);
  gtk_box_pack_start(GTK_BOX(hbox_b),draw_r[dev],TRUE,TRUE,0);
  gtk_box_pack_start(GTK_BOX(hbox_b),radio_r,FALSE,FALSE,0);
  gtk_box_pack_start(GTK_BOX(vbox),hbox_b,TRUE,TRUE,0);

  /* add callbacks for the labels (so they can be changed) */
  g_signal_connect(G_OBJECT(l_label[dev]),"clicked",
                   G_CALLBACK(ccrec_label_change_cb),(void *)dev);
  g_signal_connect(G_OBJECT(r_label[dev]),"clicked",
                   G_CALLBACK(ccrec_label_change_cb),(void *)dev);

}

/* add all the buttons below the drawing widgets */
gtk_box_pack_start(GTK_BOX(hbox_c),button_stop,FALSE,FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox_c),button_play,FALSE,FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox_c),button_rec,FALSE,FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox_c),button_pause,FALSE,FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox_c),recording_name,TRUE,TRUE,0);

gtk_box_pack_start(GTK_BOX(vbox),hbox_c,FALSE,FALSE,0);

/* add callbacks for the buttons */
g_signal_connect(G_OBJECT(button_stop),"clicked",
                 G_CALLBACK(ccrec_stop_cb),NULL);
g_signal_connect(G_OBJECT(button_rec),"clicked",
                 G_CALLBACK(ccrec_rec_cb),NULL);
g_signal_connect(G_OBJECT(button_play),"clicked",
                 G_CALLBACK(ccrec_play_cb),NULL);
g_signal_connect(G_OBJECT(button_pause),"clicked",
                 G_CALLBACK(ccrec_pause_cb),NULL);
g_signal_connect(G_OBJECT(patient_id),"clicked",
                 G_CALLBACK(ccrec_start_new_patient_cb),NULL);

gtk_option_menu_set_menu(GTK_OPTION_MENU(patient_name),gtk_menu_new());
```

```
  return vbox;
}


/* init_gui() is called in main() to set up all of the windows for the gui */
void init_gui(GtkWidget *top, GtkWidget *patient_top, GtkWidget *rec_type_top,
              GtkWidget *lc_top)
{
  int dev;
  int height,width;
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  patient_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  rec_type_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  lc_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  /* allocate and lay out all the widgets in the main window */
  top = make_top(window);
  gtk_container_add(GTK_CONTAINER(window),top);
  /* allocate and lay out all the widgets in create new patient
     dialog */
  patient_top = make_patient_data(patient_window);
  gtk_container_add(GTK_CONTAINER(patient_window),patient_top);
  /* allocate and lay out all the widgets in the recording type
     selection dialog */
  rec_type_top = make_rec_type(rec_type_window);
  gtk_container_add(GTK_CONTAINER(rec_type_window),rec_type_top);
  /* allocate and lay out all the widgets in the label change
     dialog */
  lc_top = make_label_change(lc_window);
  gtk_container_add(GTK_CONTAINER(lc_window),lc_top);

  /* set a reasonable default size for the main window. this depends
     on the processor speed, since the bigger the window is, the more
     processing is needed for updating the display. these values are
     reasonable for a PIII 850, with four total channels (two audio
     devices). */
  gtk_window_set_default_size(GTK_WINDOW(window),330,550);
  gtk_window_set_title(GTK_WINDOW(window),"CCRec");

  gtk_window_set_title(GTK_WINDOW(patient_window),"New Patient");

  /* show the main window */
  gtk_widget_show_all(window);

  /* allocate the pixmap buffers for the drawing widgets. width will
     always be less than the screen width, height will always be less
     than the screen height divided by the number of channels. */
  height=gdk_screen_height()/(NUM_AUDIO_DEVICES*2);
  width=gdk_screen_width();
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    draw_l_b[dev] = gdk_pixmap_new(draw_l[dev]->window,width,height,-1);
    draw_r_b[dev] = gdk_pixmap_new(draw_r[dev]->window,width,height,-1);
  }
```

```
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* add callbacks for expose events for drawing widgets */
    g_signal_connect(G_OBJECT(draw_l[dev]),"expose_event",
                     G_CALLBACK(ccrec_expose_event),draw_l_b[dev]);
    g_signal_connect(G_OBJECT(draw_r[dev]),"expose_event",
                     G_CALLBACK(ccrec_expose_event),draw_r_b[dev]);
  }

}
```

B.7 init.h

```
int opensound(FILE **dsp, char *dev_name,int *dspfd,short **buf,
              int *bufsize, int do_init);
int read_config_file();
```

B.8 init.c

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/select.h>
#include <sys/timeb.h>

#include <gtk/gtk.h>
#include <gpe/init.h>

#include "ccrec_audio.h"

#include "cb.h"
#include "gui.h"

#include "main.h"

/* opensound() opens a particular sound device and initializes the settings for the device */
int opensound(FILE **dsp, char *dev_name,int *dspfd,short **buf, int *bufsize, int do_init)
{
  /* open the device and allocate a file descriptor */
  printf("Opening %s. . .",dev_name);
  if((*dsp=fopen(dev_name,"a+"))==NULL) {
    printf("Error: Unable to open sound device.\n");
```

```
    return 1;
  }

  /* get the file descriptor number */
  *dspfd=fileno(*dsp);

  /* if we need to initialize this device, then call sound_init to set the sample rate, etc */
  if(do_init) {
    if((*bufsize=sound_init(*dsp,SOUND_16BIT,SOUND_STEREO,44100))==1) {
      printf("Error: Unable to initialize sound card.\n");
      return 1;
    }
  } else {
    /* if we didn't call sound_init, then we'll use a buffer size of 4k */
    *bufsize=4096;
  }
  printf("Buffer size: %d. . .",*bufsize);
  /* allocate the buffer */
  *buf = (short *)malloc(*bufsize);
  if(*buf == NULL) {
    printf("Error: Unable to create local buffer.\n");
  }
  printf("allocated.\n");
  return 0;
}

/* read_config_file()
   Each line of ccrec.conf contains a description of a sound device to read from:
   device name, initialize?, left channel name, right channel name
   e.g., "/dev/dsp1",1,"Carotid L","Carotid R"
   read_config_file parses the config file and fills in the audio_device struct
*/
int read_config_file()
{
  FILE *conf;
  char devname[256],leftname[256],rightname[256];
  int doinit,i;
  struct ad_struct *tmp_ada;
  struct ad_struct *tmp_adb=NULL;

  /* open the config file */
  conf=fopen("ccrec.conf","r");

  /* just starting, so no audio devices */
  NUM_AUDIO_DEVICES=0;

  /* go until we get to the end of the file */
  while(!feof(conf)) {
    /* attempt to read in a whole line at once. if it fails, break out of the while loop */
    if(fscanf(conf,"\"%[^\"]\",%d,\"%[^\"]\",\"%[^\"]\"\n",devname,&doinit,
              leftname,rightname)<4)
      break;
    /* if we successfully read a line, then we have another audio device */
    NUM_AUDIO_DEVICES++;
```

```
    /*printf("**%s**%d**%s**%s\n",devname,doinit,leftname,rightname);*/
    /* allocate a new list of ad_struct */
    tmp_ada=(struct ad_struct *)malloc(sizeof(struct ad_struct)*NUM_AUDIO_DEVICES);
    /* if the previous one is not NULL, copy it over into the new one */
    if(tmp_adb!=NULL)
      memcpy(tmp_ada,tmp_adb,sizeof(struct ad_struct)*NUM_AUDIO_DEVICES-1);
    /* and now fill in the new struct in the list */
    strncpy(tmp_ada[NUM_AUDIO_DEVICES-1].devname,devname,256);
    tmp_ada[NUM_AUDIO_DEVICES-1].init=doinit;
    strncpy(tmp_ada[NUM_AUDIO_DEVICES-1].name_l,leftname,256);
    strncpy(tmp_ada[NUM_AUDIO_DEVICES-1].name_r,rightname,256);
    /* free the old one, set the old one to be the one we just created */
    if(tmp_adb!=NULL)
      free(tmp_adb);
    tmp_adb=tmp_ada;
  }
  printf("num_audio_devices=%d\n",NUM_AUDIO_DEVICES);
  /* we're done reading the file, set the main list to be the last valid list */
  audio_device=tmp_adb;
  for(i=0;i<NUM_AUDIO_DEVICES;i++)
    printf("**%s**%d**%s**%s**\n",audio_device[i].devname,audio_device[i].init,
           audio_device[i].name_l,audio_device[i].name_r);
  fclose(conf);
  return 1;
}
```

B.9 main.h

```
/* the maximum length of a patient name, arbitrarily chosen */
#define MAX_PATIENT_NAME 128

/* lists for various display widgets which depend on the number of
   audio devices we're recording from */
GtkWidget *window, *patient_window, *rec_type_window, *lc_window;
GtkWidget **draw_l, **draw_r; /* actual drawing area on screen */
GtkWidget *patient_name, *recording_name, *patient_id;
GtkWidget *patient_name_entry, *patient_id_entry;
GtkWidget *rec_type_cust_entry, *rec_type_option, *rec_type_label;
GtkWidget *button_stop, *button_play, *button_rec, *button_pause;
GtkWidget **r_label, **l_label, *lc_entry;
/* pixmaps for double-buffering the drawing area */
GdkPixmap **draw_l_b, **draw_r_b;

/* name of the currently selected recording */
char *recording_name_tmp;
/* name of the currently selected patient */
char patient_name_tmp[MAX_PATIENT_NAME];

/* the currently selected channel for aural playback */
int audioselection;
```

```
/* current mode of the program */
int mode;
#define STOPPED 1
#define PLAYING 2
#define PLAYING_LIVE 3
#define RECORDING 4
/* whether or not the output display is paused */
int paused;

/* how many points are displayable */
int *data_len;
/* total length of data recorded */
int *total_len_l;
int *total_len_r;

/* size of the smallest audio device hardware buffer */
unsigned long smallestbuf;

float *sscale[2];
```

B.10 main.c

```
#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <sys/select.h>
#include <sys/timeb.h>
#include <signal.h>

#include <gtk/gtk.h>
#include <gpe/init.h>

#include "ccrec_audio.h"

#include "cb.h"
#include "gui.h"
#include "init.h"
#include "wave.h"

#include "main.h"

int NUM_AUDIO_DEVICES;

short **data_l;
short **data_r;
```

```
short **buf;
short **sep_buf_l;
short **sep_buf_r;
int *bufsize;
int *dspfdnum;


/* separate_channels() extracts the left and right channels into two
   different buffers.  length is the length of buf, which is twice
   the length of left or right.  This assumes that left and right are
   big enough to hold all of the samples. */
int separate_channels(short *buf, short *left, short *right, int length)
{
  int i,j;
  /* if the length is not even, this it doesn't make sense */
  if((length>>1)<<1 != length)
    printf("separate_channels(): odd number of samples!\n");

  i=j=0;
  while(i<length) {
    left[j]=buf[i++];
    right[j++]=buf[i++];
  }

  return j;
}


/* combine_channels() interlaces the left and right channels into one
   buffers.  length is the length of left or right (which should be
   the same length), which is half the length of buf. This assumes
   that buf is big enough to hold all of the samples. */
int combine_channels(short *left, short *right, short *buf,int length)
{
  int i,j;

  i=j=0;
  while(j<length) {
    buf[i++]=left[j];
    buf[i++]=right[j++];
  }

  return i;
}


int wait_input()
{
  fd_set set;
  int i;
  int dev;
  struct timeval timeout;

  FD_ZERO(&set);
```

```
  i=0;
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* find max FD number */
    i=i>dspfdnum[dev] ? i : dspfdnum[dev];
    /* add FD to select() struct */
    FD_SET(dspfdnum[dev], &set);
  }                                                                    90
  /* one second timeout for select() */
  timeout.tv_sec=1;
  timeout.tv_usec=0;
  /* call select to see if data available */
  if((i=select(i+1,&set,NULL,NULL,&timeout)) == -1) {
    printf("Error: select!\n");
  }

  return i;
}                                                                      100

/* sampleloop() is the main loop of this application
   it pushes audio data between the audio devices, files, and the GUI. */
int sampleloop()
{
  int dev,i,j[NUM_AUDIO_DEVICES],j_max[NUM_AUDIO_DEVICES];
  int len[NUM_AUDIO_DEVICES];
  int avg_save_l[NUM_AUDIO_DEVICES][128],avg_save_r[NUM_AUDIO_DEVICES][128];
  /*struct timeb timeout1;*/
  /*struct timeb timeout2,timeout3;*/                                  110

  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++)
    j_max[dev]=j[dev]=0;

  do
    {
      wait_input();
      /*ftime(&timeout2);*/ /* used for timing this loop */
      /* clear the select() struct */
      /*printf("%d:%ld:%ld\n",i,timeout.tv_sec,timeout.tv_usec);*/   120

      /* call gtk main to update GUI */
      gtk_main_iteration_do(FALSE);

      if(mode != STOPPED) {
        if(mode == PLAYING_LIVE || mode == RECORDING) {
          /* if we're not playing, then we are moving data from the
             audio devices to the GUI */
          for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
            /* try to read as much data from the audio device as we can   130
               (max to read is the size of the audio device's hardware
               buffer).  Read from each device in turn */
            if(dsp[dev]!=NULL &&
               (len[dev]=fread((char *)buf[dev],sizeof(char),smallestbuf,dsp[dev])) == -1) {
              printf("Error: Unable to read audio data.\n");
              return 1;
            }
```

```
    /* the number of bytes read should be a multiple of 4
       (stereo = 2 values/sample, 16bit = 2 bytes/value). If
       not, the alignment of the data may get off. This really
       shouldn't happen (the driver & hardware don't allow it),
       so if we see it, there's really something more serious
       going on (driver bug, memory error). */
    if(((len[dev]>>2)<<2)!=len[dev])
      printf("Warning: read non-multiple of 4 number of bytes from audio: %d\n",
             len[dev]);
    /* de-interlace the channels */
    separate_channels(buf[dev],sep_buf_l[dev],sep_buf_r[dev],
                                len[dev]);
    /* if we're also recording, dump the audio data to the
       recording files. */
    if(mode == RECORDING) {
      if(datafd_l[dev]!=NULL &&
         (total_len_l[dev]+=fwrite((char *)sep_buf_l[dev],sizeof(char),
                                   len[dev]/2,datafd_l[dev])) == -1) {
        printf("Error: Unable to write audio data file for left channel.\n");
        return 1;
      }
      if(datafd_r[dev]!=NULL &&
         (total_len_r[dev]+=fwrite((char *)sep_buf_r[dev],sizeof(char),
                                   len[dev]/2,datafd_r[dev])) == -1) {
        printf("Error: Unable to write audio data file for right channel.\n");
        return 1;
      }
    }
  }
}
/*printf("length=%d\n",len[dev]);*/
/*next three lines used for timing this loop*/
/*ftime(&timeout3);*/
/*printf("%d\n",timeout3.millitm);*/
/*printf("%d:%d:%d\n",timeout1.millitm,timeout2.millitm,timeout3.millitm);*/
if(mode == PLAYING && !paused) {
  /* if playing, then we are moving data from files to the GUI
     (and audio devices). if paused, then don't do anything. */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* try to read some audio data from the file */
    if(datafd_l[dev]!=NULL &&
       (len[dev]=fread((char *)sep_buf_l[dev],sizeof(char),smallestbuf/2,
                       datafd_l[dev])) == -1) {
      printf("Error: Unable to read audio data file for left channel.\n");
      return 1;
    }
    if(datafd_r[dev]!=NULL &&
       (len[dev]=fread((char *)sep_buf_r[dev],sizeof(char),smallestbuf/2,
                       datafd_r[dev])) == -1) {
      printf("Error: Unable to read audio data file for right channel.\n");
      return 1;
    }
    if(len[dev] == 0) {
      /* if we reach the end of the file, pretend somebody
```

```
        pressed the stop button, thereby resetting state */
      ccrec_stop_cb(NULL,NULL);
    }
    /* check to see if this device is selected for writing to
       the sound card */
    if(audioselection==dev || audioselection==dev+NUM_AUDIO_DEVICES) {
      /* if it is, see if it's the left or right channel
         that's selected and re-form a stero stream */
      if(audioselection<NUM_AUDIO_DEVICES)                                    200
        combine_channels(sep_buf_l[dev],sep_buf_l[dev],buf[dev],len[dev]/2);
      else
        combine_channels(sep_buf_r[dev],sep_buf_r[dev],buf[dev],len[dev]/2);
      /* also output it to the audio device it came from (in case
         there's a listener) */
      if(dsp[dev]!=NULL &&
         (len[dev]=fwrite((char *)buf[dev],sizeof(char),len[dev]*2,dsp[dev])) == -1) {
        printf("Error: Unable to write audio data.\n");
        return 1;
      }                                                                       210
    }
  }
}

/* We've just spent a bunch of time processing, so give the GUI a chance to update */
gtk_main_iteration_do(FALSE);

/* only update the display if we're not paused */
if(!paused) {
  /* prepare the data for display by the GUI (average it,                    220
     since we don't have enough pixels to display
     everything) */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* i will keep track of the offset from the pointer to the
       data we just read */
    i=0;
    /* j keeps track of how many display data points have been
       generated since the last update */
    if(j[dev]<16) {
      /* if it is less than 16, generate more */                              230
      /* take AVG_AMOUNT samples and average them together,
         generating one display point. Do this until we run
         out of data. */
      while((i+(AVG_AMOUNT-1))<(len[dev]/4)) {
        avg_save_l[dev][j[dev]]=avg_samples(sep_buf_l[dev]+i,AVG_POW);
        avg_save_r[dev][j[dev]++]=avg_samples(sep_buf_r[dev]+i,AVG_POW);
        i+=AVG_AMOUNT;
      }
    } else {
      /* if it is 16 or more, then update the display (prevents us from doing   240
         gratuitous display updates, saving on some processor time). */
      j_max[dev]=j[dev];
      /* shift everything in the display data to the right */
      for(i=255;i>=j_max[dev];i--) {
        data_l[dev][i]=data_l[dev][i-j_max[dev]];
```

```
            data_r[dev][i]=data_r[dev][i−j_max[dev]];
          }
          /* insert the new data points */
          for(i=0;i<j_max[dev];i++) {
            data_l[dev][i]=avg_save_l[dev][j_max[dev]−1−i];                                    250
            data_r[dev][i]=avg_save_r[dev][j_max[dev]−1−i];
            if(data_len[dev]<256)
              data_len[dev]++;
          }
          /* reset j */
          j[dev]=0;
          /* redraw the display */
          ccrec_redraw(draw_r[dev],draw_r_b[dev],data_r[dev],data_len[dev],sscale[1][dev]);
          ccrec_redraw(draw_l[dev],draw_l_b[dev],data_l[dev],data_len[dev],sscale[0][dev]);
        }                                                                                      260
      }
    }
    /*next two lines used for timing this loop*/
    /*ftime(&timeout3);
      printf("%d %d = %d\n",timeout3.millitm,timeout2.millitm,
      timeout3.millitm-timeout2.millitm);*/
  } else { /* mode is UNKNOWN or STOPPED */
    /* nothing better to do, so just spin in the GUI */
    gtk_main_iteration_do(FALSE);
  }                                                                                            270
 }
 while (1); /* repeat until killed */
}



/* alloc_vars() allocates and initializes the variables whose size depends
   on the number of audio devices */
void alloc_vars()
{                                                                                              280
 int i,dev;

 /* most variables in this application are arrays, the length of the array
    is the number of audio devices there are */

 /* dsp and datafd are the file descriptors for the audio devices and
    recording/playback files */
 dsp=(FILE **)malloc(sizeof(FILE *)*NUM_AUDIO_DEVICES);
 datafd_l=(FILE **)malloc(sizeof(FILE *)*NUM_AUDIO_DEVICES);
 datafd_r=(FILE **)malloc(sizeof(FILE *)*NUM_AUDIO_DEVICES);                                  290

 /* draw_[lr] are the empty canvas widgets for outputting waveforms */
 draw_l=(GtkWidget **)malloc(sizeof(GtkWidget *)*NUM_AUDIO_DEVICES);
 draw_r=(GtkWidget **)malloc(sizeof(GtkWidget *)*NUM_AUDIO_DEVICES);
 /* draw_[lr]_b are pixmaps used for double-buffering draw_[lr] */
 draw_l_b=(GdkPixmap **)malloc(sizeof(GtkPixmap *)*NUM_AUDIO_DEVICES);
 draw_r_b=(GdkPixmap **)malloc(sizeof(GtkPixmap *)*NUM_AUDIO_DEVICES);
 /* [rl]_label are the labels used for the right and left channels of
    the display */
```

```
  r_label=(GtkWidget **)malloc(sizeof(GtkWidget *)*NUM_AUDIO_DEVICES);
  l_label=(GtkWidget **)malloc(sizeof(GtkWidget *)*NUM_AUDIO_DEVICES);

  /* length of the display data array (reset when display is cleared) */
  data_len=(int *)malloc(sizeof(int)*NUM_AUDIO_DEVICES);
  /* total length of data written to recording file */
  total_len_l=(int *)malloc(sizeof(int)*NUM_AUDIO_DEVICES);
  total_len_r=(int *)malloc(sizeof(int)*NUM_AUDIO_DEVICES);

  /* sscale is a float used to scale the visual display data before drawing */
  sscale[0]=(float *)malloc(sizeof(float)*NUM_AUDIO_DEVICES);
  sscale[1]=(float *)malloc(sizeof(float)*NUM_AUDIO_DEVICES);

  /* data_[lr] are the temp buffers for the display data */
  data_l=(short **)malloc(sizeof(short *)*NUM_AUDIO_DEVICES);
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++)
    data_l[dev]=(short *)malloc(sizeof(short)*256);
  data_r=(short **)malloc(sizeof(short *)*NUM_AUDIO_DEVICES);
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++)
    data_r[dev]=(short *)malloc(sizeof(short)*256);

  /* buf is the temp buffer for data read from the audio device */
  buf=(short **)malloc(sizeof(short *)*NUM_AUDIO_DEVICES);
  /* sep_buf is the temp buffer for separated left-right audio data */
  sep_buf_l=(short **)malloc(sizeof(short *)*NUM_AUDIO_DEVICES);
  sep_buf_r=(short **)malloc(sizeof(short *)*NUM_AUDIO_DEVICES);
  /* bufsize is the length of the buffer (which is allocated to be
     the same size as the hardware buffer of the audio device */
  bufsize=(int *)malloc(sizeof(int)*NUM_AUDIO_DEVICES);

  /* the file descriptor number of the audio device FDs, cached for
     use by select() */
  dspfdnum=(int *)malloc(sizeof(int)*NUM_AUDIO_DEVICES);

  /* initialize some vars */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    /* FDs should be NULL, since nothing's been opened yet */
    datafd_l[dev]=datafd_r[dev]=dsp[dev]=NULL;
    /* default scale is 8.0, unless overridden by the command-line option */
    sscale[0][dev]=sscale[1][dev]=8.0;
    /* no display data yet */
    data_len[dev]=total_len_l[dev]=total_len_r[dev]=0;
    for(i=0;i<256;i++)
      data_r[dev][i]=data_l[dev][i]=0;
  }
}

void sigint_handler(int sig)
{
  int dev;
  printf("Closing files and exiting...");
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    if(datafd_l[dev]!=NULL) {
      write_wave_header_size(datafd_l[dev],total_len_l[dev]);
```

```
      fclose(datafd_l[dev]);
    }
    if(datafd_r[dev]!=NULL) {
      write_wave_header_size(datafd_r[dev],total_len_r[dev]);
      fclose(datafd_r[dev]);
    }
    if(dsp[dev]!=NULL)                                                    360
      fclose(dsp[dev]);
  }
  printf("done.\n");
  exit(0);
}

int main(int argc, char **argv)
{
  int dev;
  GtkWidget *top, *patient_top, *rec_type_top, *lc_top;                   370
  char *tmp;

  smallestbuf=-1;

  /* register SIGINT so we can shut down cleanly */
  signal(SIGINT,&sigint_handler);

  /* fill in audio_device struct from config file */
  read_config_file();
                                                                          380
  /* now that we know the number of audio devices, allocate
     variables */
  alloc_vars();

  /* open file descriptors for audio devices, initialize devices, set
     sample rate, etc */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    if(opensound(&dsp[dev],audio_device[dev].devname,&dspfdnum[dev],
                 &buf[dev],&bufsize[dev],audio_device[dev].init)) {
      return 1;                                                           390
    }
    smallestbuf=smallestbuf<bufsize[dev] ? smallestbuf : bufsize[dev];
  }
  /* allocate the separated buffers */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++) {
    sep_buf_l[dev]=(short *)malloc(sizeof(char)*smallestbuf);
    sep_buf_r[dev]=(short *)malloc(sizeof(char)*smallestbuf);
  }

  /* get visual scaling options from command line: */                     400
  /* read in as many pairs as there are audio devices, delimited by
     the ':' character */
  if(argc>2)
    if(!strcmp(argv[1],"-s")) {
      tmp=strtok(argv[2],":");
      for(dev=0;dev<NUM_AUDIO_DEVICES&&tmp!=NULL;dev++) {
        sscale[0][dev]=atof((char *)tmp);
```

```
        tmp=strtok(NULL,":");
        if(tmp!=NULL)
          sscale[1][dev]=atof((char *)tmp);
        tmp=strtok(NULL,":");
        printf("visual scaling for dev %d = %f:%f\n",dev,sscale[0][dev],sscale[1][dev]);
      }
    }

  /* initialize gpe & gtk */
  if(gpe_application_init(&argc,&argv)==FALSE) {
    printf("gpe failed to init!\n");
    exit(1);
  }

  /* allocate & initialize the display widgets */
  init_gui(top, patient_top, rec_type_top, lc_top);

  /* read the patients from the patientdb directory */
  fill_patient_choice();

  /* read the recording types from the config file */
  fill_rec_type_choice();

  /* select will block until the timeout the first time it's called on
     an audio device if nothing has read from it first, so read some
     junk from the audio device */
  for(dev=0;dev<NUM_AUDIO_DEVICES;dev++)
    fread(buf[dev],sizeof(char),smallestbuf,dsp[dev]);

  /* when we start, we are stopped and not paused */
  ccrec_stop_cb(NULL,NULL);

  /* go to the main loop (sample, display, repeat) */
  if(sampleloop())
    return 1;

  /* shouldn't get here - if we do, return with non-zero exit code */
  return 1;
}
```

B.11 wave.h

```
void write_wave_header_init(FILE *datafd,int rate);
void write_wave_header_size(FILE *datafd,int datasize);

struct wave_header_single_chunk_pcm {
  unsigned long a;
  unsigned long filesize;
  unsigned long b[4];
  unsigned long samplerate;
  unsigned long bytespersecond;
```

```
  unsigned long c[2];
  long datasize;
} __attribute__((packed));
```

B.12 wave.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "wave.h"

void write_wave_header_init(FILE *datafd,int rate)
{
  struct wave_header_single_chunk_pcm hdr;
  char b[16]={ 'W','A','V','E','f','m','t',' ',
               0x10,0x00,0x00,0x00,
               0x01,0x00,
               0x01,0x00 };
  char c[8]={ 0x04,0x00,
            0x10,0x00,
            'd','a','t','a' };

  memcpy(&hdr.a,"RIFF",4);
  memcpy(&hdr.b,&b,16);
  memcpy(&hdr.c,&c,8);

  /* don't know file size yet */
  hdr.filesize=hdr.datasize=0;
  /* fill in sample rate */
  hdr.samplerate=(unsigned long)rate;
  /* sample rate * 2 bytes per sample * 1 channels */
  hdr.bytespersecond=rate*2;

  rewind(datafd);
  fwrite(&hdr,1,sizeof(struct wave_header_single_chunk_pcm),datafd);

  return;
}

void write_wave_header_size(FILE *datafd,int datasize)
{
  struct wave_header_single_chunk_pcm hdr;

  /* read current header */
  rewind(datafd);
  fread(&hdr,1,sizeof(struct wave_header_single_chunk_pcm),datafd);

  /* file size = data chunk size + header size - 8 */
  hdr.filesize=datasize+sizeof(struct wave_header_single_chunk_pcm)-8;
  /* fill in data chunk size */
  hdr.datasize=datasize;
```

```
  /* write new header */
  rewind(datafd);
  fwrite(&hdr,1,sizeof(struct wave_header_single_chunk_pcm),datafd);

  return;
}
```

Bibliography

[1] Analog Devices. *AD627: Micropower, Single and Dual Supply Rail-to-Rail Instrumentation Amplifier, Rev. B*, 2001.

[2] Analog Devices. *AD7654: Dual 2-Channel Simultaneous Sampling SAR 500 kSPS 16-Bit ADC*, 2002.

[3] Shawn Carlson. Home is where the ecg is. *Scientific American*, June 2000.

[4] Compaq, Intel, Microsoft, NEC. *Universal Serial Bus Specification Revision 1.1*, September 1998.

[5] Gary W. Duncan M.D. et al. Evaluation of carotid stenosis by phonoangiography. *New England Journal of Medicine*, 293:1124–1128, November 1975.

[6] M.T. Magyar et al. Carotid artery auscultation–anachronism or useful screening procedure? *Neurological Research*, 24(7):705–708, October 2002.

[7] J.S. Sauve et all. Can bruits distinguish high-grade from moderate symptomatic carotid stenosis? the north american symptomatic carotid endarterectomy trial. *Annals of Internal Medicine*, 120(8):633–637, April 1994.

[8] Henry Gray F.R.S. *Gray's Anatomy*. Barnes & Noble Books, New York, 1995. reproduction of fifteenth edition, 1901.

[9] Takuro Ikeda. *Fundamentals of Piezoelectricity*, chapter 1. Oxford University Press, Oxford, 1996.

[10] Myer Kutz, editor. *Standard Handbook of Biomedical Engineering and Design*, chapter 17. McGraw-Hill, New York, 2003.

[11] Maxim. *MAX889: High-Frequency, Regulated, 200mA, Inverting Charge Pump*, July 2000.

[12] Mark H. Beers M.D. and Robert Berkow M.D., editors. *The Merck Manual of Diagnosis and Therapy*, chapter 201. Merck Research Laboratories, 17th edition, 1999.

[13] Mark H. Beers M.D. and Robert Berkow M.D., editors. *The Merck Manual of Diagnosis and Therapy*, chapter 174. Merck Research Laboratories, 17th edition, 1999.

[14] Mark H. Beers M.D. and Robert Berkow M.D., editors. *The Merck Manual of Diagnosis and Therapy*, chapter 197. Merck Research Laboratories, 17th edition, 1999.

[15] Microsoft Corporation. *AVI RIFF File Reference*, 2002.

[16] National Semiconductor. *LM124/LM224/LM324/LM2902 Low Power Quad Operational Amplifiers*, August 2000.

[17] Ali Shoeb. Wavelet tutorial. personal communication, July 2003.

[18] (unattributed). Preventing stroke brain basics. http://www.ninds.nih.gov/health_and_medical/pubs/preventing_stroke.htm, July 2001. accessed May 14, 2003.