University of Southern Queensland

Faculty of Engineering & Surveying

# Speech Compression System for Student Feedback

A dissertation submitted by

D. Hugo

in fulfilment of the requirements of

**ENG4112 Research Project**

towards the degree of

**Bachelor of Computer Systems Engineering**

Submitted: October, 2008

# Abstract

Sound is a vital part of daily life for most people, but has yet to be fully explored in the area of education, particularly in terms of student feedback. Feedback so far has generally been limited to documents if by students and teacher are apart, or by a mixture of documents and talking if students and teacher are face-to-face.

The aim of this project is to change this situation, by automating the processes of recording, editing, compression and playback of speech. By developing a series of applications, one for the teacher and another for the student, this project hopes to meet the needs of student feedback on such a scale as would be suitable for student assessment in a university environment. Specifically, the design needs to meed the following requirements:

- The teacher should be able to create, cut, edit, save and load sound files.
- The students should be able to open and play sound files sent to them by the teacher.
- As an optional extra (if time permits), the teacher should be able to create a naming convention using student ID's.
- Able to be operated safely by both student and teacher.

Areas that are covered in this document include: sound waveforms, codecs, the meaning of speech compression, comparison of development platforms during project development, issues identified during development, various libraries investigated, the program itself, and possible topics for future research.

University of Southern Queensland

Faculty of Engineering and Surveying

**ENG4111/2 *Research Project***

## Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

**Prof F Bullen**

Dean

Faculty of Engineering and Surveying

# Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

D. Hugo

0011222040

_____

Signature

_____

Date

# Acknowledgments

I would like to thank my sister for donating her time to proofread this document.

I would also like to thank my supervisors Mark Phythan and John Leis for organizing this project.

D. Hugo

*University of Southern Queensland*
*October 2008*

# Contents

**Abstract** **i**

**Acknowledgments** **iv**

**List of Figures** **ix**

**List of Tables** **x**

**Chapter 1 Introduction** **1**

1.1 Project Aims and Objectives . . . . . 2

1.2 Overview of the Dissertation . . . . . 3

**Chapter 2 Background** **4**

2.1 Background . . . . . 4

2.1.1 Software and Sound . . . . . 4

2.1.2 Sound Waveforms and Compression . . . . . 5

2.1.3 Codecs . . . . . 6

2.1.4 The Decision . . . . . 8

2.2 Research And Project Objectives . . . . . 9

2.2.1 Research . . . . . 9

2.2.2 Issue Identification . . . . . 10

2.2.3 Design and Development . . . . . 10

2.2.4 Final Program . . . . . 10

2.2.5 Dissertation . . . . . 10

2.3 Timelines . . . . . 11

2.3.1 Proposed Program . . . . . 11

2.4 Consequential Effects . . . . . 11

2.5 Safety Issues . . . . . 12

2.6 Summary . . . . . 12

**Chapter 3 Methodology And Design** **13**

3.1 Methodology . . . . . 13

3.1.1 Software Development . . . . . 13

3.1.2 Codec Choice . . . . . 14

3.2 Design Requirements . . . . . 14

3.3 Constraints and Alternatives . . . . . 15

3.3.1 Project Constraints . . . . . 15

3.3.2 Alternatives . . . . . 16

3.4 Ethics, Safety, and Other Requirements . . . . . 16

3.5 Resource Planning . . . . . 17

3.6 Summary . . . . . 18

**Chapter 4 Program Development** **19**

4.1 Issues With Development . . . . . 19

4.1.1 Development Platforms . . . . . 19

4.1.2 Development Environment . . . . . 23

4.2 Programming Issues . . . . . 23

4.2.1 Saving and Loading Files . . . . . 23

4.2.2 Sound Playback . . . . . 24

4.3 Summary . . . . . 24

**Chapter 5 The Final Version** **25**

5.1 The Program . . . . . 25

5.2 Summary . . . . . 27

**Chapter 6 Conclusions and Further Work** **28**

6.1 Achievement of Project Objectives . . . . . 28

6.2 Further Work . . . . . 28

**References** **29**

**Appendix A Project Specification** **30**

**Appendix B Source Code** **32**

# List of Figures

2.1 Inside a CELP encoder. (Juin-Hway 1992) . . . . . 7

2.2 Inside a CELP decoder. (Juin-Hway 1992) . . . . . 7

5.1 Sample screen-shot of final version in Windows XP. . . . . 26

5.2 Sample screen-shot of the decompressor working in Windows Vista. . . . 27

# List of Tables

2.1 Proposed program for project completion. . . . . 11

# Chapter 1

# Introduction

Sound is an essential part of daily life. It is not quite as an essential part of the process, however, when it comes to the feedback process between student and assessors. There is a limited number of programs available that cater to speech, and even less are specifically tailored to compression, editing, and playback on a wide scale such as would be suitable for student assessment at a university environment.

Feedback between student and assessor is limited due to time and medium, usually email, electronic document, or handwritten notes. Such forms are usually terse and to the point because of time constraints, and handwritten notes are generally the worst form of all to receive as feedback (this makes no comment as to handwritten assignments). Feedback via electronic documents, while seeming straightforward, is compounded by the lack of an industry standard document format, compounded by some vendor's tendency to continually change proprietary document format codes.

All this makes it hard for students to receive proper guidance on their work, or at the very least read the comments provided by the assessor. Nor does this mean that the feedback will be followed. For instance, according to van Till (2003), "There was no evidence to show that students responded to the teachers feedback. For example, a comment from a teacher suggested a student should make changes to a diagram. This was not actioned by the student."

The situation is slightly easier for students who can receive face-to-face feedback. Even then, however, difficulties may be encountered in terms of accent, speech that is too fast, too slow, etc. But this is still no help for students and teachers separated by distance.

This leads to the inevitable conclusion that adding the ability to provide sound files to the assessor's feedback abilities will improve the teaching methods a great deal. However, converting sound into electronic form encounters a number of problems. This project aims to attempts to address these problems and counter them with an application.

## 1.1 Project Aims and Objectives

The aim of the project is to automate the processes of recording, editing, compression and playback of speech for student assessment. Specifically, this involves several objectives that will need to be completed in order for the project to be considered complete. These include:

- Study existing material relating to sound
- Research methods relating to sound compression
- Identify issues likely to be encountered during the course of the project
- Design and develop a program that will meet the final criteria in the design brief
- Produce a final program
- Present the project at the engineering conference
- Write an academic dissertation on the lessons learned

## 1.2 Overview of the Dissertation

This dissertation is organized as follows:

**Chapter 2** describes the basic problem and provides background information.

**Chapter 3** discusses the methodology and design of the application(s).

**Chapter 4** considers the program's development over time.

**Chapter 5** explains the final program and how it works.

**Chapter 6** concludes the dissertation and suggests further work in the area of research and program development.

# Chapter 2

# Background

## 2.1 Background

Broadly speaking, the project was a combination of software and sound. There were plenty of sound applications available, and plenty of video applications. However, when one narrowed the field down to speech applications, the choice became much smaller. The fact was, there were not very many editors out there. There were programs that performed similar tasks, but they failed to cover all aspects of the given requirements. There were no one source that covered all aspects of the project due to its unique nature.

### 2.1.1 Software and Sound

Most sound applications, if not devoted almost entirely to music or video, when applied to speech have seemed to devote themselves to converting text to speech. The most common example of this were the screen-reader applications, such as JAWS for the blind, which was basically a speech synthesizer. Very few applications entered the gap in between, for converting actual speech into a sound file, let alone editing the file.

One possible approach to this gap would be that of speech recognition. This involved the tedious and time-consuming process of converting speech to text, and transmitting

this to the receiver, in this case the student. However, this is known to be inaccurate, with a low success rate, and it takes time to train the system to recognise the speech of the recorder (the assessor), and the system would have to be trained for each assessor's speech patterns. Needless to say, this approach was briefly considered and then rejected.

Editing a sound waveform with current applications was too generalised for the needs dictated by this project task. These applications were also not user-friendly and required significant user training for the user to become proficient in their usage. Thus a more specialised approach needed to be taken.

Before explaining this approach, perhaps it would be good to explain what sound waveforms are in this digital age, and why manipulating them was so problematic.

### 2.1.2 Sound Waveforms and Compression

Sound waveforms are generally stored in files, that is, bitstreams (a timed series of bits), more commonly known as streams. A file is simply a stationary stream, that is, a bitstream that has been captured and stored on computer storage media. Streams, on the other hand, can be moved over a network, be it a LAN or the Internet – hence the common Internet term 'streaming media', which technically refers to the method of delivery of the media and not the media itself.

According to Spanias (1994), "Speech Coding or Speech Compression is the field concerned with obtaining compact digital representations of voice signals for the purpose of efficient transmission or storage....The objective in speech coding is to represent speech with a minimum number of bits while maintaining its perceptual quality." The trouble comes in that there are a number of ways in which to create a representation of sampled sound waveform.

Sound waveforms themselves are sampled or compressed into files and are decompressed into streams via codecs. A codec, as the name suggests, is code in memory that is used whenever needed to compress or decompress a portion of a stream from one format to another. Common codecs are mpeg, aac, ogg-vorbis, flac, and wma.

### 2.1.3 Codecs

All codecs work by basically choosing a method of compression that is deemed to be most efficient and then using it to compress and decompress the sound waveform. Although that is a bit simplistically termed. According to (Liesenborgs 2000), the three most common codec forms are waveform coding, vocoders, and hybrid coding.

Waveform coding is where the signal is stored in such a way that the resulting signal will have the same general shape as the original. As such, they apply to audio signals in general and no just to speech. In brief, they include such methods as differential coding, differential PCM (DPCM), adaptive PCM (ADPCM), Delta modulation (DM), vector quantisation (VQ coding), and transform coding.

Vocoders exploit the fact that it is speech that is being compressed, and encode information about how the information was produced by the human vocal chords rather than the waveform itself. For instance, whether it is voiced ('a', 'f', etc.), silent ('sh', etc.), or mixed. (Spanias 1994) One of the more common forms is a linear predictive coding (LPC), which applies a difference equation to each sample such that each sample is basically a linear combination of the previous one.

It is well known, however, that waveform coding does not perform well below 16 kbps, and vocoders, while able to perform at low data rates and still produce intelligible speech, will also render the speaker unrecognisable. Hybrid coding seeks out the best of both worlds.

Hybrid coding seek to keep the waveform recognisable, that is, to keep both the speech intelligible and the speaker recognisable while still being able to perform in the low data rates. There are residual excited linear predictors (RELP), and codebook excited linear predictors (CELP). As the latter are what the project used, this is what will be focused on.

### Inside a CELP Codec

Think of a codec like a little black box which can wrap a present up and unwrap it again, where the present is the sound waveform. Inside the codec is a encoder and decoder, shown in Figures 2.1 and 2.2 respectively.

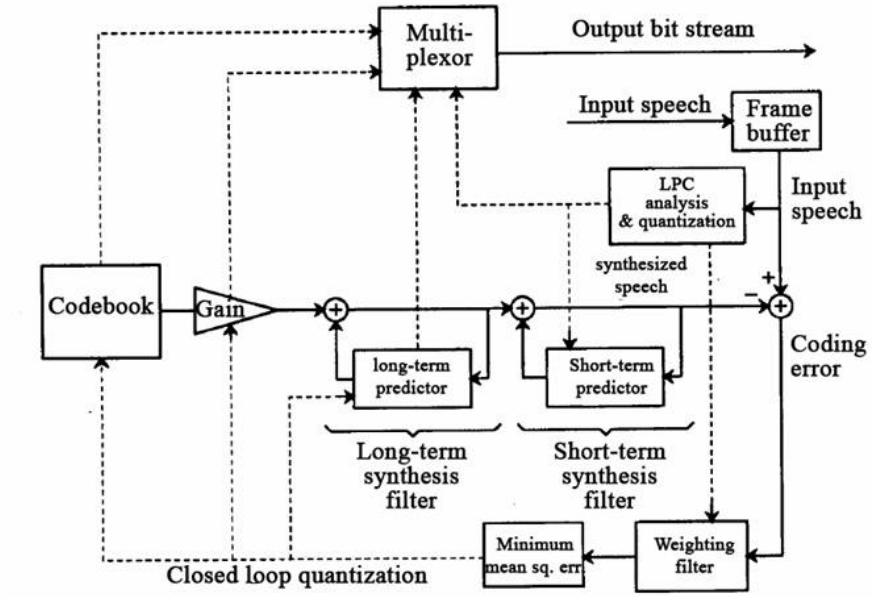


Figure 2.1: Inside a CELP encoder. (Juin-Hway 1992)

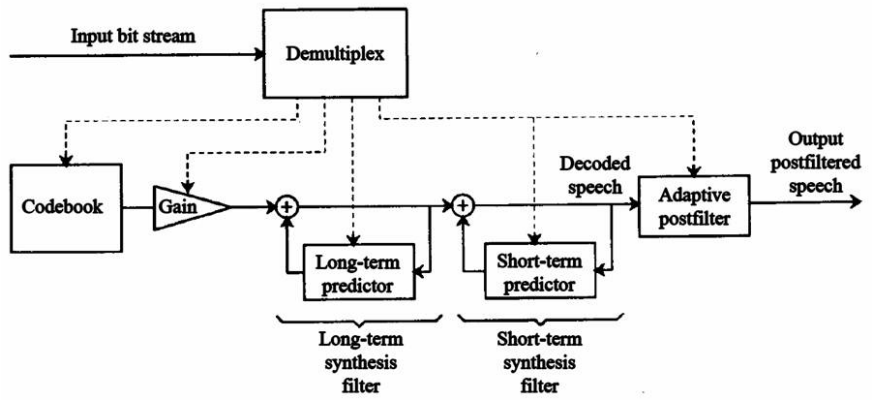


Figure 2.2: Inside a CELP decoder. (Juin-Hway 1992)

The CELP encoder splits the incoming data up into "chunks" called frames and attempts a process called *analysis-by-synthesis*. It has a codebook, like a dictionary, and

tries all candidates for the waveform data until the coding error reaches a minimum, at which point it turns the encoded data over to the decoder. In this way, it hopes to minimise the coding error between the synthesised speech and the input speech.

The decoder, as shown on page 7, is functionally opposite that of the encoder. The filter at the end is designed to make the speech appear natural, and not synthesised (as most vocoded speech appears to).

### 2.1.4 The Decision

As already noted, file compression is a major part of the project, and the codec is how this takes place. Therefore, the choice of codec is a critical part of the suitable compression system, and leaves two options: creation of one's own codec, or using a pre-existing codec.

Creating a codec for the use of the system is a potential choice, but risky. There are many codecs already available and fair proportions of them are proprietary and/or have patents underlying them. There is thus a risk that a new codec would infringe on these patents. Also, creating a codec specifically for the new application could tie the user to a new file format for that specific application. The new codec would also have to be carefully written to ensure that other audio programs, encoders and decoders, would be able to handle the new file format.

On the other hand, using a more widely known codec is safer, but not without its own share of drawbacks. The main drawback is patent infringements and the resulting legal minefield this causes. There are also the documented overheads of using pre-existing codecs which involve the quality and quantity of the compression that the codec can produce. The codec at a certain bit-rate (number of bits that are conveyed or processed per unit of time) can only compress so far, and the quality of the output will only be so good. For instance, a 128 kb ogg-vorbis stream is much superior to a 128 kb mp3 stream.

This example has proved why research was so important in this project to meet its aims.

## 2.2 Research And Project Objectives

The aim of the project was to automate the processes of recording, editing, compression and playback of speech for student assessment. Specifically, this involved several objectives that needed to be completed in order for the project to be considered complete. These included:

- Study existing material relating to sound
- Research methods relating to sound compression
- Identify issues likely to be encountered during the course of the project
- Design and develop a program that will meet the final criteria in the design brief
- Produce a final program
- Present an academic dissertation on the lessons learned

### 2.2.1 Research

The existing compression methods research was vital. The speech compression aspect was a significant part of this project, as file sizes had to be minimised to allow for storage and fast transfer. Research in this area identified suitable compression systems.

Tailoring the research for voice files and quality of sound was also vital because, as shown above, the final file given to students from the assessor was to be designed specifically for speech, with a reasonably intelligible quality for hearing. Again, research in this area identified a suitable codec system, eventually deciding on the CELP codec, as this area was closely tied to the first area of research.

Research into the ability to edit the waveform was another aspect that received attention, as due to the nature of the project, the editing was being done 'in-house', by the program itself.

### 2.2.2 Issue Identification

Issue identification provided the link between the research portion and the design and development stages of the project. What this implies was that issues were be identified in both stages, and required both stages working together in a kind of symbiosis to resolve these issues.

For example, as already noted, one of the biggest issues involved is speech compression (or how much the voice is compressed when creating the final file). This was a key area when selecting codecs for research. Then the issue of ideal bit-rates for speech compression while still maintaining quality was addressed, which prompted further specific research and program development. Other issues which arose will be discussed further in Chapter 4.

### 2.2.3 Design and Development

Design and development was another main area of the project. As indicated previously, it was a dynamic and ongoing area, where the information and data found during the research stage was applied and developed to the program. Exactly how dynamic will be discussed in Chapter 4.

### 2.2.4 Final Program

Construction of the final program according to the final design of this size took much time. Coding and debugging any application of this size and nature was always going to be massive undertaking, as this dissertation shows in Chapter 3 to Chapter 5.

### 2.2.5 Dissertation

The dissertation was the ultimate goal of the project. The dissertation has documented the research, design, and development and presented all these stages as part of a unified whole.

Table 2.1: Proposed program for project completion.

| Task | Status | Date in 2008 |
|---|---|---|
| Specification | Completion | 25 March |
| Appreciation | Completion | 26 May |
| Research | Completion | 30 June |
| Design and Development | Completion | 30 June |
| Final Program | Commence | 30 June |
| Tidy-up | Commence | 1 September |
| Draft Dissert | Completion | 14 September |
| Conference | | September |
| Project Completion | | 30 October |

## 2.3 Timelines

While the current schedule is hazy, there are some definite deadlines in place. However, some changes may yet take place to meet the goals as these deadlines approach.

### 2.3.1 Proposed Program

Please note that while Research is listed as being complete on 30 June, it does not mean that no further research will be carried out after that date. It is simply the projected date on which the major research, or a significant portion of which, is expected to be complete. Further research will be carried out as needed after that date. The same principle applies to design and development, although it is hoped that this adjustment will be limited.

## 2.4 Consequential Effects

Long-term consequences and effects are hard to determine and harder to measure, as most of them are intangible.

As an eventual part of the student feedback process, the assessor will be able to deliver much more complete and detailed feedback to the student. The student will then be able to complete later assignments and exams to a higher standard than would otherwise have been possible with lower quality feedback. By making use of voice recordings, this lightens the load on both assessor and student, as the student does not have to decipher handwriting and the assessor has a lighter examination load. It also forms a bond between teacher and student that would not otherwise have been formed, especially for external students who might have no other form of contact besides the written word. It would thus go a long way towards aiding the learning and discovery process, and move towards an excellent standard of online education.

## 2.5 Safety Issues

There are relatively few physical safety issues in this project, as it is a project based mainly in the digital world.

More intangible risks include such ones as a loss of data incident. This is a real danger to the project, and can only be avoided by regular and routine backup of data.

## 2.6 Summary

This project to develop recoding, editing, compression, and playback of speech is a fairly large and complex task. It requires careful research, design and development, and the writing of a reasonably involved application. This project combines both sound engineering skills with software. As with all projects of a software engineering nature, careful management of time and resources need to be maintained in order for the final objectives to be successfully completed.

It is hoped that this project will, at the very least, provide the groundwork for improved relations between student and assessor and vastly improve the student feedback process at universities in the future.

# Chapter 3

# Methodology And Design

## 3.1 Methodology

The overall design method used for the project is an incremental or progressive method. With the dissertation and presentation being the ultimate goals, these should be written progressively and concurrently with the other stages to ensure these remain up-to-date and advanced.

### 3.1.1 Software Development

The choice was made early on to use a Rapid Application Development (RAD) tool, in this case the CodeGear C++ Builder application. (CodeGear is a subsidiary of Borland.) This was in order to make sure the practical part of the project was complete with sufficient time for the written work.

RAD is known for speeding up the software development process. A task which would normally up to twelve or twenty-four months can be completed in six months, which is well within the scope of this project. RAD is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups

- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

It is a concept that seemed catered for project development, and applying all or some of its ideas here seemed that it would indeed lead to a higher quality final product.

### 3.1.2 Codec Choice

Research into this area is expected to identify a suitable compression system, or codec. The most likely candidate of research at this stage is that of benchmarking. A benchmark in itself is the standard by which something else is measured. The process of benchmarking is to measure the performance of an item relative to another item in an impartial and scientific manner. This process, in audio terms, generally involves collating a wide range of standard sample sound files. These are then run through various sound compression methods (codecs) and measured to see which method obtained the best results, mixing quality of output with compression rate.

## 3.2 Design Requirements

The design requirements for the project was to develop an application suite for the student and teacher. The teacher application had to capture input from the microphone and compress it, while the student application had to decompress sound data and play it back. The student application is thus a cut-down version of the teacher's (complete) version of the application. Depending on time within the project schedule, the teacher application may be extended to utilise a list of student identification numbers to set up a file-naming convention when saving files. (Refer to Appendix for original specification and details.)

## 3.3 Constraints and Alternatives

As with any software project, there are a number of things to consider in a software project of this size. Due to the open nature of the project specification, one would think that the sky is the limit – for instance, additional features could be added that do not conflict with the specifications as time permits. An example of this would be designing the user interface so that it is as user-friendly and accessible as possible (depending on the development platform, as will be later explained).

Another point to consider was the format to use for the sound files. Providing the method used to read the sound-files is itself free from patent issues, there is a choice of what format to use in which to read and to save the sound files. The options include pre-existing formats such mp3, wave, etc., and also the possibility of writing one's own format. This latter option, however, was deemed beyond the scope of the project.

The former option, relying on pre-existing formats, did however bring up the issue of project constraints and patents.

### 3.3.1 Project Constraints

What the application under development cannot do is infringe on software patents. Perhaps it is not clear what a software patent is. A software patent defines a method or idea implemented in software and prevents other software developers from making their own implementations of that idea or method without paying the inventor of the patent a fee. For commercial projects this fee would not be a problem because the revenue stream could cover the development costs. This project is for an educational institution which would like to see costs minimized, and thus development methods where software patents would be an issue were avoided.

### 3.3.2 Alternatives

One of the alternatives investigated at an early stage was compression and decompression using one of the free codecs implemented in the ffmpeg library. The ffmpeg library is a software library, that is, a collection of program routines in a single file that make it easier for the software developer and the computer to remember where the routines are.

This effort was hampered by lack of an overall guide to the ffmpeg library. Despite this, a decompression routine was written that could read the audio track of any format. However, lack of centralized documentation made writing a compression routine difficult. The various stages of writing a compression routine (selecting a format, selecting a codec, selecting parameters) was documented in comments of the source code in different parts the library, making it difficult to form the coherent view needed to code in confident manner. It would have been nice if a guide to writing a custom compression routine was included. This coding method was abandoned for these reasons.

The coding method that replaced it was a library that could compress with a few lines of code. It shall be discussed further in a later section of the thesis, Chapter 4.

Another topic that received much attention was the development platform under which the software was developed. As this played a part in program development, however, it shall be discussed further in Chapter 4, as will the development environment.

## 3.4 Ethics, Safety, and Other Requirements

The ethics of a project of this nature have mainly to do with plagiarism. The first area this appears is to with patents, and this has already been discussed and considered. The second area is that of the source code of the application itself. However, all code used was original, in that no source code in the actual application came from sources external to the author. The third area is that of references to the current "body of work" in the outside world. However, much of the information used in this project came from the pre-existing knowledge base of the author, therefore such references are

limited in number.

There is also not much that has to be said in terms of safety considerations. The final application has demonstrated no memory leaks or segmentation faults. Moreover, regular compilation after every few lines of completed code ensured rigorous and complete testing throughout program development, and all known errors have been eliminated. The application is also safe in the sense that when the user presses the wrong button (i.e. clicking "Paste" before "Copy"), relevant error message(s) appear warning the user of the mistake.

There are no other requirements to be considered.

## 3.5 Resource Planning

As with any project, some resources were required for project completion, and the degree and amount of resources used varied depending on the project and the expected outcome.

Initial negotiations and specifications stated that a program would be delivered to the university. This was the stated goal, and this was the aim of the project. Time constraints may yet cause this program to be a scaled-down version, a prototype of the final version. Regardless, there are certain aspects that can be stated with a high degree of confidence.

Certain resources that are definite to be required are as follows:

- ISP plan for relatively unrestricted Internet access with high download limits, as some (most) of the information relating to waveforms comes in large file sizes.
- Latest versions of software and licensed access to code and software.
- Computer time to access and collate the research from various sources.
- Computer-time to program and debug the code for the application.

- Reliable data storage facilities (with backups) to store all research, word-processing files, and code as they are found, collated, coded, etc.
- The costs involved will be covered by the researcher, using savings set aside for the project.

## 3.6 Summary

It was found through the course of program development that the best development platform to use was actually an open source, cross-platform combination of the GNU compiler with the Allegro library instead of a more commercial-based combination. Not only was this easier, but this will also make the final application(s) more widely available to a wider audience.

Further topics of research in this area could include creating a "prettier" GUI, and either improving the latency or finding a work-around.

The ethics and safety issues have been considered, and eliminated as a matter of concern as much as is possible.

# Chapter 4

# Program Development

As has been hinted at in previous chapters, development of the final version was not without its battles. From choice of codec to platform to development environment, a number of obstacles had to be overcome before the final version could be produced.

## 4.1 Issues With Development

As was discussed in Chapter 3, an initial decision was to use a Rapid Application Development (RAD), in this case the CodeGear C++ Builder application known as Borland. This is what is known as a development platform. However, certain issues arose with this, as will be discussed below.

### 4.1.1 Development Platforms

A development platform is a suite of tools provided by a third party to aid the software developer write code more quickly and efficiently. It may be supported by a text editor with a compiler and linker capability. This is known as a integrated development environment, shortened to an IDE. IDEs come in many versions, and as a result have different advantages and disadvantages that influence their use. In developing of this project's software, there was a choice of the development platfrom that leant towards

Borland or GNU as the compiler, with Win32/DirectX or Allegro being the respective code-base.

### Borland and Win32

The version of Borland in particular is Borland C Builder 5. The reason this version chosen was that it allowed for relatively quick generation of a windows-based graphical user interface, as all the interface code was done by the application itself. However, Borland was not without its drawbacks. The toolkit it came with was outdated (sevem years old at time of writing) and incomplete. It was incomplete in that it was not suited for making a serious sound-based application as the toolkit it provided had no sound support.

The main problem with this version of Borland, however, was that it placed limits on how much memory could be allocated – for the less computer-literate, access to memory via memory allocation is very important for calculations and sound manipulation. There were two options to access more memory when the limit was reached. The first was to access the Windows API and access memory that way; needless to say, it was a very complicated and painful method to do this every time memory needed to be accessed. The second method was to perform a memory-mapped allocation from a file on the disk (that is, a file that listed all the needed memory locations). Not only would updating this file be tedious every time the program changed or needed updated, but there was a security risk of leaving such an important file on the disk (the risk of bad sectors on the disk, inappropriate access, waste of space, time to access, etc.).

If Borland had allowed proper memory access, there would be no need for resorting to another vendor.

Another drawback of Borland was that to use Windows, via the Win32 API, one also needed to use DirectX.

The Win32 API is a 32-bit based Application Programming Interface designed to interact with the Windows environment. It is simply a number of header files with associated functions that allow the programmer and developer to perform various tasks

in the Windows environment, such as access memory, access hardware, create dialogs with various features, etc.

However, in terms of using the Win32 API in this application, a number of problems were encountered.

In terms of capturing sound and playing it back, using the Win32 API was very slow. It required constant polling (checking and accessing) of the sound-card, taking up resources which could have been used better elsewhere. It was also a fairly "messy" and consuming process to capture the data from the sound-card (see later comments on DirectX). Using Win32 API for sound-based application was, a whole, counter-intuitive. While using the API for many programs in the Windows environment may intuitive and relatively simple, using it to create a serious sound-based application is not.

The main reason for this counter-intuitiveness was based on the messaging system Windows uses to control its processes, in particular its idle processes. When the computer is running "idle", then Windows will switch over to one of these idle processes to occupy processor time until such time as the computer is "busy" again. Unfortunately, this is always not a desirable situation. In a sound-based application, an "idle" state occurs while the computer is recording sound, particular silence (aka "no sound"). To overcome this, the application would require complete control over the computer while recording to prevent the running of any other processes while recording. This is not something that Win32 is designed to allow.

As has been previously mentioned, DirectX is an integral part of the Win32 API. In particular, DirectX is how the Win32 API accesses hardware. However, like the Win32 API, it is also "bulky" to use, creating a lot of unnecessary lines of code to do simple hardware operations. In this way, it holds the programmer where there is no need to do so, even though its use is common practice.

The reasons listed above led to the consideration of the second development platform, namely that of the GNU and Allegro combination.

### GNU and Allegro

GNU refers to the GNU C/C++ compiler, which is free, open source, and cross-platform compatible. This compiler comes with the standard glibc and glibc++, that is, general C and C++ libraries, which contain standard functions and perform as the programmer expects, especially in the area of memory allocation. There are no problems with memory access as was experienced with Borland here. Moreover, GNU also can handle properly implemented messaging, which is where Allegro comes in.

Allegro is a library of functions, originally intended for computer games and other types of multimedia programming. Like GNU compiler, it is free, open source, and runs on a variety of platforms (Linux, Windows, DOS, Macintosh, etc.). Despite being its own library of functions, it can also interface with the standard C/C++ library. As a result, there is no need to worry about things like virtual memory, etc. and the core logic of the program written using Allegro is simpler. Another reason for the simpler logic is that Allegro is not "bulky" in how it accesses hardware. Whether it uses DirectX, OpenGL or some other method, it is hidden within a function, and yet the library is not cumbersome nor as painful to use as some higher-level libraries. In this senses, it provides more of a "middle-ware" function for the programmer and developer, with a stable interface that is relatively easy to use. And unlike the DirectX documentation, its manual actually makes sense.

However, Allegro does have its disadvantages. Its main disadvantage was its latency issues. There was a one second delay between input and display – i.e. between taking in input from the microphone and showing it on screen. On the other hand, output is shown much quicker. This was because the output buffer can be specified, but the input buffer was hard-wired into Allegro to be a certain amount. The second disadvantage was that Allegro has some very "un-pretty" GUI code, as it was originally based on Atari GEM system. At this point of development, however, the application was more focused on actually working, not on how pretty it looked – although that *is* a rather effective teaching tool for a GUI.

With the development platform decided, the development environment could now be chosen.

### 4.1.2 Development Environment

The development environment is the tool that allows one to use the development platform, in this case the GNU compiler. The environment that was chosen was that of DevCpp, by Bloodshed Software. The version in particular used was 4.9.9.2, or version 5 beta, which uses the GNU compiler.

## 4.2 Programming Issues

During the process of programming, other issues arose as well. These had mainly to do with saving and loading files, and sound playback – critical features of the final program.

### 4.2.1 Saving and Loading Files

Accessing the codec was a large part of the program's ability to save and load files. As part of the initial debugging process, the program would accept the names given it via the Save/Load dialog(s) as a "throw-away", that is, it would accept the names but would not do anything with them but would discard them. To perform the function properly of saving and loading files, it needed to access the codec. To do this, it needed the Speex library.

Technically, Speex is both a codec and a library and it comes from Xiph.Org Foundation, the people who also made the ogg-vorbis codec. Speex is a Open Source/Free Software patent-free audio compression format designed for speech which can achieve compression ratios from 10:1 up to 28:1. According to its own codec manual, "During development, we were careful not to use techniques known to be patented and we searched through the USPTO database to see if anything we were doing was covered." (Valin 2007) Interacting with Speex, however, is a manual task, where one must set most options manually, but there is also an added layer of control over the format of the final sound file.

A simpler method would be to access the libfishsound library, which also uses the Speex library. In general, to use libfishsound requires very few lines of code, and one would obtain a compressed sound waveform in an official file format. However, attempts to compile the library were repeatedly unsuccessful, and libfishsound itself was under a relatively open license, it was encumbered by restrictions by its dependent distribution (the developer of one of its dependent libraries refused to release a statically linked version).

Thus is was decided to write with Speex directly instead of libfishsound.

### 4.2.2 Sound Playback

Sound playback caused a few problems in development.

Because sound capture and playback is so interwined, it was had to work out which one was playing up when the program would not play the sound-file properly. Was it not capturing the sound properly in the first place, or was there something wrong with the playback routines?

It took some time to track down, and quite a bit of testing, but the problems were tracked down to a "divide by 2" bug in the playback routines. When this line of code was removed, the sound files captured played back very well.

## 4.3 Summary

Developing this program can only be described as an adventure. A number of obstacles were encountered and had to be overcome. From choice of codec, to development platform and environment, to sound problems, all of them were overcome quite successfully. The program that resulted works.

# Chapter 5

# The Final Version

## 5.1 The Program

The teacher's application window is separated into two parts, the primary buffer, and the secondary buffer. The primary buffer is where the capturing takes place, and the secondary buffer receives pasted portions from the primary buffer through simple copy and pasting techniques. Files are opened in primary buffer and saved through the secondary buffer. Each buffer has a viewing window and three sliders representing the beginning and end of a selection window and the current playback position. Buttons to the side of each viewing window correspond to the actions that can be performed on each buffer, such as copy, paste, play, stop, pause, etc. When a portion of a buffer is played, only the portion of the selection window is played. When a copy and paste movement is performed from the primary buffer, to the secondary buffer, the selected portion of the primary buffer is copied to the end of the secondary buffer and the size of the secondary buffer increases accordingly.

The capture window is displayed by clicking the capture button next to the primary buffer's viewing window or by selecting it from the file menu. A scope of the microphone is constantly displayed while the capture window is active to aid in microphone calibration. Buttons to the right start and stop recording, and accept or cancel the current session of microphone recording. Captured data can be appended onto the end

of an existing session by re-entering the capture window from the main window.

Perhaps it is not exactly clear how copy and pasting in the application works. Copy and pasting works through the first of the two sliders beneath each viewing window, which are also responsible for maintaining the current selection range of the viewing window's buffer. It is the selection range that decides what is copied and also played. Suppose the first bar is moved to 25

The student's window is very similar to the teacher's window. The secondary buffer, the capture ability of the primary buffer, and the ability to select a range from the viewing window are all removed from the application. It is essentially a read-only, open-only, play-only version of the teacher application as everything to do with manipulation of the waveform has been removed. To the side of the viewing window the play pause and stop buttons are larger to make the program more 'user-friendly' and since there is more space available.

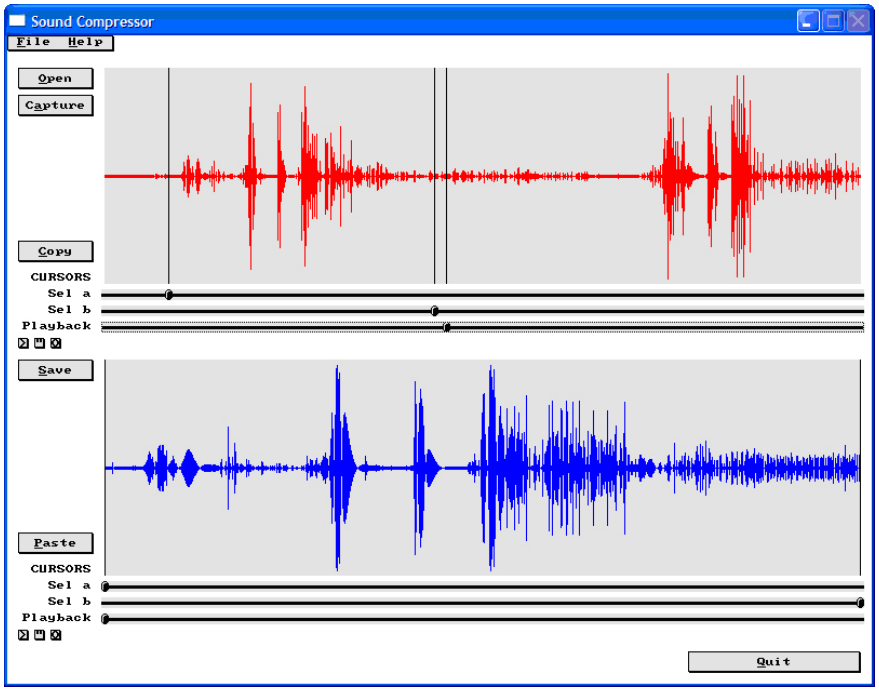


Figure 5.1: Sample screen-shot of final version in Windows XP.

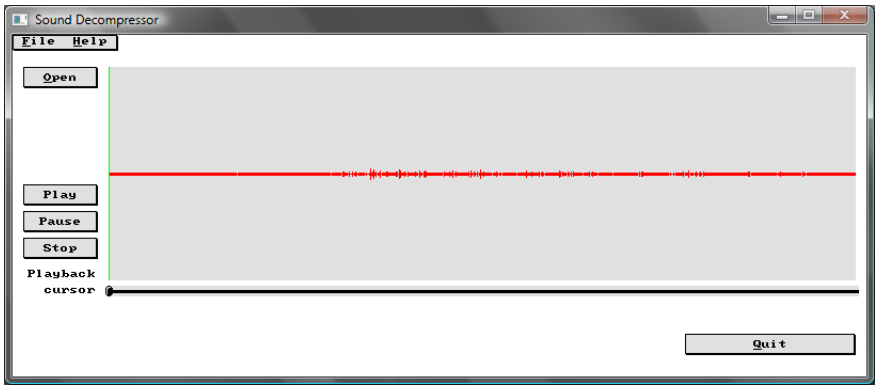


Figure 5.2: Sample screen-shot of the decompressor working in Windows Vista.

## 5.2 Summary

The final version of the application presents both student and teacher windows. It is a full and complete and working application that meets the stated aims of the project.

Due to time constraints, however, the optional extra of adding the ability of the teacher to have file-naming convention as per student ID's has not been implemented. This would necessitate adding database facilities to the application, which were beyond the time and scope of the project. This would perhaps be a topic for further research and development.

# Chapter 6

# Conclusions and Further Work

## 6.1 Achievement of Project Objectives

All Project objectives have been met.

## 6.2 Further Work

Because the application was programmed with Allegro, there are some latency issues, as noted in Chapter 3. Further development could improve these, or develop a work-around. Also noted in this chapter was certain drawbacks of Allegro's GUI code, not so much the positioning of the interface features but the lack of "prettiness". However, at this stage of development, it was determined that making the application work was the focus, and not the so-called "bells and whistles".

# References


Juin-Hway, C. e.-a. (1992), 'A low-delay celp coder for the ccitt 16 kb/s speech coding standard', *IEEE Journal on Selected Areas in Communications* **10**(5).

Liesenborgs, J. (2000), Voice over ip in networked virtual environments, Master's thesis, Universiteit Maastricht, Netherlands.
`http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Thesis`
current October 2008.

Spanias, A. S. (1994), Speech coding: A tutorial review, *in* 'Proceedings of the IEEE', Vol. 82, IEEE.

Valin, J. (2007), *The Speex Codec Manual*, version 1.2 beta 3 edn.
`http://speex.org/docs/manual/speex-manual.pdf`
current October 2008.

van Till, E. (2003), *An Evaluation of the Quality of Teacher Feedback to Students: A Study of Numeracy Teaching in the Primary Education Sector*, Massey University's Institute for Professional Development and Educational Research, Albany.
`http://www.aare.edu.au/03pap/kni03053.pdf`
current October 2008.

# Appendix A

# Project Specification

University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

**ENG4111/4112 Research Project**

***Project Specification***

| | |
|---|---|
| FOR: | David Ian HUGO |
| TOPIC: | Speech Compression System for student feedback |
| SUPERVISORS: | Mr Mark PHYTHIAN |
| PROJECT SPONSORSHIP: | University of Southern Queensland |
| PROJECT AIM: | This project aims to automate the process of recording, editing, compression and playback of speech for student assessment. |

PROGRAMME: (Issue A, March 25, 2008)

1) Research existing speech compression methods and comment on their suitability for producing low file sizes.
2) Research speaker-specific compression, such as whether improved performance and/or relative quality is desired.
3) Design and develop software to record, edit, compress speech from a microphone or portable recordable device.
4) Design and develop software for students to decompress and playback edited recordings.
5) Add optional features to software in (2) such as importing a list of student names to set up a file naming convention where appropriate.
6) Present an academic dissertation on the lessons learned.

AGREED _____ (student) _____ (supervisor)

Dated: / / 2008 Dated: / / 2008

Examiner/CoExaminer _____

# Appendix B

# Source Code

Modules developed were divided up according to function. In the teacher application this included the performed initialization and de-initialization (main.cpp), the main editor window (editor.cpp), the capture window (capture.cpp), and a fourth module containing miscellaneous helper functions (sndhelp.cpp).

The main module is responsible for initializing the allegro library, setting up the sound playback and capture devices, and a timer callback.

Listing B.1: A basic Windows program.

```
#include <allegro.h>
#include <winalleg.h>
#include "capture.h"
#include "editor.h"
void init();
void deinit();

volatile int frame_counter = 0;

void my_timer_handler()
{
    frame_counter++;
}

int main()
{
        init();
        editor_window();
        deinit();
        return 0;
}
END_OF_MAIN()


void init()
{
        int depth, res;
        allegro_init();
        depth = desktop_color_depth();
        if (depth == 0) depth = 32;
        set_color_depth(depth);
        res = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 800, 600, 0, 0);
        if (res != 0) {
                allegro_message(allegro_error);
                exit(-1);
        }
    set_window_title("Sound Compressor");

        detect_digi_driver(DIGI_AUTODETECT);
        if (install_sound(DIGI_DIRECTX(0), MIDI_NONE, NULL) == -1) {
       allegro_message(allegro_error);
       exit(-1);
    }
        if (install_sound_input(DIGI_DIRECTX(0), MIDI_NONE) != 0) {
       allegro_message(allegro_error);
```

```
        exit(-1);
    }
        install_timer();
        install_keyboard();
        install_mouse();

    LOCK_VARIABLE(frame_counter);
    LOCK_FUNCTION(my_timer_handler);
        install_int(my_timer_handler, 250);

    show_mouse(screen);
        /* add other initializations here */
}


void deinit()
{
        clear_keybuf();
        /* add other deinitializations here */
}
```

The editor module is responsible for keeping track of the playback positions and the current selection ranges. Controls reference each other in the control structure by index (particularly the playback and the selection range controls).

Listing B.2: A basic Windows program.

```
#ifndef editorH
#define editorH

void editor_window(void);
int my_button_proc (int msg, DIALOG *d, int c);
void init_dialog_colors(DIALOG *dlg);

#endif
```

Listing B.3: A basic Windows program.

```
#include <allegro.h>
#include <mem.h>
#include "sndhelp.h"
#include "capture.h"


extern volatile int frame_counter;
int newfile(void);
int openfile(void);
int capture(void);
int save(void);
int copy(void);
int paste(void);
int quit(void);
int about(void);
int testtone(void);
int my_button_proc    (int msg, DIALOG *d, int c);
int my_button_proc_ex(int msg, DIALOG *d, int c);
int d_waveform_proc   (int msg, DIALOG *d, int c);
int d_playback_proc   (int msg, DIALOG *d, int c);
int play(DIALOG *d);
int pause(DIALOG *d);
```

```
int stop(DIALOG *d);
void render_waveform(BITMAP *bmp, unsigned short *data, unsigned length, u

char *second_buffer = NULL;
int second_buffer_size = 0;

char *clipboard_buffer = NULL;
int clipboard_buffer_size = 0;


/* the first menu in the menubar */
MENU menu1[] =
{
    { "&New \tCtrl+N",       newfile,  NULL,       0,  NULL  },
    { "&Open \tCtrl+O",     openfile,  NULL,       0,  NULL  },
    { "Ca&pture \tCtrl+p", capture,    NULL,       0,  NULL  },
    { "&Save \tCtrl+S",         save,  NULL,       0,  NULL  },
    { "&Quit \tq/Esc",          quit,  NULL,       0,  NULL  },
    { NULL,                     NULL,  NULL,       0,  NULL  }
};


/* the help menu */
MENU helpmenu[] =
{
    { "&About \tF1",       about,  NULL,       0,  NULL  },
    { "&Test tone",     testtone,  NULL,       0,  NULL  },
    { NULL,                NULL,  NULL,       0,  NULL  }
};


/* the main menu-bar */
MENU the_menu[] =
{

    { "&File", NULL,     menu1,           0,       NULL  },
    //{ "&Second", NULL,    menu2,          0,       NULL  },
    { "&Help",  NULL,    helpmenu,        0,       NULL  },
    { NULL,     NULL,    NULL,            0,       NULL  }
};


/* here it comes... the big ugly dialog structure */
DIALOG editor_dialog[] =
{
    /* (dialog proc)     (x)   (y)   (w)   (h) (fg)(bg) (key) (flags)
(d1) (d2)     (dp)                  (dp2) (dp3) */

    /* this element just clears the screen, therefore it should come befor
    { d_clear_proc,        0,    0,    0,    0,    0,    0,    0,
0,       0,    0,    NULL,                  NULL, NULL  },

    /* a menu bar - note how it auto-calculates its dimension if they are
    { d_menu_proc,         0,    0,    0,    0,    0,    0,    0,
0,       0,    0,    the_menu,              NULL, NULL  },

    /* first waveform display */
    { d_waveform_proc,    90,   30,  700,  200,    0,    0,    0,
0,       2,    0,    NULL,                  (void*)&buffer, (void*)&buffe
```

```
    { d_slider_proc,       87, 235,  706,   10,   0,  0,    0,
0,     699,   0,    NULL,                      NULL, NULL  },
    { d_slider_proc,       87, 250,  706,   10,   0,  0,    0,
0,     699, 699,    NULL,                      NULL, NULL  },
    { d_slider_proc,       87, 265,  706,   10,   0,  0,    0,
0,     699,   0,    NULL,                      NULL, NULL  },

    /* a bunch of descriptive text elements */
    { d_rtext_proc,        78, 220,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"CURSORS",         NULL, NULL  },
    { d_rtext_proc,        78, 235,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Sel_a",           NULL, NULL  },
    { d_rtext_proc,        78, 250,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Sel_b",           NULL, NULL  },
    { d_rtext_proc,        78, 265,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Playback",        NULL, NULL  },

    /* second waveform display */
    { d_waveform_proc,     90, 300,  700,  200,   0,  0,    0,
0,      10,   0,    NULL,                      (void*)&second_buffer, (void*)
},
    { d_slider_proc,       87, 505,  706,   10,   0,  0,    0,
0,     699,   0,    NULL,                      NULL, NULL  },
    { d_slider_proc,       87, 520,  706,   10,   0,  0,    0,
0,     699, 699,    NULL,                      NULL, NULL  },
    { d_slider_proc,       87, 535,  706,   10,   0,  0,    0,
0,     699,   0,    NULL,                      NULL, NULL  },

    /* another bunch of descriptive text elements */
    { d_rtext_proc,        78, 490,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"CURSORS",         NULL, NULL  },
    { d_rtext_proc,        78, 505,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Sel_a",           NULL, NULL  },
    { d_rtext_proc,        78, 520,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Sel_b",           NULL, NULL  },
    { d_rtext_proc,        78, 535,    0,    0,   0,  0,    0,
0,       0,   0,    (void*)"Playback",        NULL, NULL  },

    /* the quit and side buttons use our customized dialog procedure, usin
    { my_button_proc,      10,  30,   70,   20,   0,  0,  'o', D_EXIT,
0,     0,    (void*)"&Open",           NULL, (void*)openfile  },
    { my_button_proc,      10,  55,   70,   20,   0,  0,  'a', D_EXIT,
0,     0,    (void*)"C&apture",        NULL, (void*)capture  },
    { my_button_proc,      10, 190,   70,   20,   0,  0,  'c', D_EXIT,
0,     0,    (void*)"&Copy",           NULL, (void*)copy  },

    { my_button_proc,      10, 300,   70,   20,   0,  0,  's', D_EXIT,
0,     0,    (void*)"&Save",           NULL, (void*)save  },
    { my_button_proc,      10, 460,   70,   20,   0,  0,  'p', D_EXIT,
0,     0,    (void*)"&Paste",          NULL, (void*)paste  },
    { my_button_proc,     630, 570,  160,   20,   0,  0,  'q', D_EXIT,
0,     0,    (void*)"&Quit",           NULL, (void*)quit  },

    /* playback buttons and nonvisual playback procedures */
    { my_button_proc_ex,   10, 280,   10,   10,   0,  0,    0, D_EXIT,
27,    0,    (void*)">",               NULL, (void*)play },
    { my_button_proc_ex,   25, 280,   10,   10,   0,  0,    0, D_EXIT,
27,    0,    (void*)"\"",              NULL, (void*)pause },
```

```
    { my_button_proc_ex,   40, 280,   10,   10,   0,   0,    0, D_EXIT,
27,   0,    (void*)"O",              NULL, (void*)stop },
    { d_playback_proc,      0,   0,    0,    0,   0,   0,    0,
0,      2,   0,    NULL,                  (void*)&buffer, (void*)&buffe
},

    { my_button_proc_ex,   10, 550,   10,   10,   0,   0,    0, D_EXIT,
31,   0,    (void*)">",              NULL, (void*)play },
    { my_button_proc_ex,   25, 550,   10,   10,   0,   0,    0, D_EXIT,
31,   0,    (void*)"\"",             NULL, (void*)pause },
    { my_button_proc_ex,   40, 550,   10,   10,   0,   0,    0, D_EXIT,
31,   0,    (void*)"O",              NULL, (void*)stop },
    { d_playback_proc,      0,   0,    0,    0,   0,   0,    0,
0,     10,   0,    NULL,                  (void*)&second_buffer, (void*)
},

    /* keyboard shortcuts */
    { d_keyboard_proc,      0,   0,    0,    0,   0,   0,    0,
0,  KEY_F1,   0,    (void *)about,         NULL, NULL },
    { d_keyboard_proc,      0,   0,    0,    0,   0,   0,  'n'-'`',
0,      0,   0,    (void *)newfile,       NULL, NULL },
    { d_keyboard_proc,      0,   0,    0,    0,   0,   0,  'o'-'`',
0,      0,   0,    (void *)openfile,      NULL, NULL },
    { d_keyboard_proc,      0,   0,    0,    0,   0,   0,  'a'-'`',
0,      0,   0,    (void *)capture,       NULL, NULL },
    { d_keyboard_proc,      0,   0,    0,    0,   0,   0,  's'-'`',
0,      0,   0,    (void *)save,          NULL, NULL },
    { d_yield_proc,         0,   0,    0,    0,   0,   0,    0,
0,      0,   0,    NULL,                  NULL, NULL },
    { NULL,                 0,   0,    0,    0,   0,   0,    0,
0,      0,   0,    NULL,                  NULL, NULL }
};


unsigned long xtosample(unsigned x, unsigned limit, unsigned width) {
   return (unsigned)((((unsigned long long)limit)*x)/width);
}


/* New file callback */
int newfile(void)
{
    if (buffer) {
        free(buffer);
        buffer = NULL;
        buffer_size == 0;
        render_waveform((BITMAP*)(editor_dialog[2].dp), (unsigned short*)
    }
    return D_O_K;
}


/* Open file callback */
int openfile(void)
{
    char filename[1024] = "";
    if (file_select_ex("Open_file", filename, NULL, 1023, 640, 480) != 0)
```

```
        alert("Selected file:", "", filename, "Ok", NULL, 0, 0);
        read_buffer_from_file(filename, buffer, buffer_size);
        render_waveform((BITMAP*)(editor_dialog[2].dp), (unsigned short*)b
    }
    return D_O_K;
}


/* Capture waveform callback */
int capture(void)
{
    capture_window();
    render_waveform((BITMAP*)(editor_dialog[2].dp), (unsigned short*)buffe
    return D_O_K;
}


/* save file callback */
int save(void)
{
    char filename[1024] = "";
    if (second_buffer_size == 0) {
        alert("Cannot save empty buffer", "", "Put some data in it first"
        return D_O_K;
    }

    if (file_select_ex("Save file", filename, NULL, 1023, 640, 480) != 0)
        alert("Selected file:", "", filename, "Ok", NULL, 0, 0);
        write_buffer_to_file(filename, second_buffer, second_buffer_size);
    }
    return D_O_K;
}


/* copy waveform section callback */
int copy(void)
{
    if (buffer_size == 0) {
        alert("Cannot copy empty buffer", "", "Put some data in it first"
        return D_O_K;
    }
    unsigned sel1 = editor_dialog[3].d2;          // get position of slid
    unsigned sel2 = editor_dialog[4].d2;          // get position of slid
    unsigned selmin = MIN(sel1, sel2);
    unsigned selmax = MAX(sel1, sel2);
    unsigned sample_begin = xtosample(selmin, buffer_size/2, editor_dialog
    unsigned sample_end = xtosample(selmax+1, buffer_size/2, editor_dialog
    unsigned sample_size = sample_end-sample_begin;

    if (clipboard_buffer) { free(clipboard_buffer); }
    clipboard_buffer = (char*)malloc(sample_size*2);
    if (clipboard_buffer != NULL) {
        clipboard_buffer_size = sample_size*2;
        memcpy(clipboard_buffer, buffer+(sample_begin*2), sample_size*2);
    } else {
        clipboard_buffer_size = 0;
        alert("Cliboard error", "", "Not enough memory for that operation"
    }
```

```
    return D_O_K;
}


/* paste waveform section callback */
int paste(void)
{
    if (clipboard_buffer_size == 0) {
        alert("Cannot paste empty buffer", "", "Put some data in it first"
        return D_O_K;
    }
    second_buffer = (char*)realloc(second_buffer, second_buffer_size + cli
    memcpy(second_buffer + second_buffer_size, clipboard_buffer, clipboar
    second_buffer_size += clipboard_buffer_size;
    render_waveform((BITMAP*)(editor_dialog[10].dp), (unsigned short*)seco
    return D_O_K;
}


/* Used as a menu-callback, and by the quit button. */
int quit(void)
{
    if (alert("Really Quit?", NULL, NULL, "&Yes", "&No", 'y', 'n') == 1)
        return D_CLOSE;
    else
        return D_O_K;
}


/* Our about box. */
int about(void)
{
    alert("* Sound Compressor *",
          "",
          "A program to compress audio from file and microphone",
          "Ok", 0, 0, 0);
    return D_O_K;
}


int testtone(void) {
    play_test_tone();
    return D_O_K;
}


/* Another menu callback. */
int menu_callback(void)
{
    char str[256];
    ustrzcpy (str, sizeof str, active_menu->text);
    alert("Selected menu item:", "", ustrtok (str, "\t"), "Ok", NULL, 0, 0
    return D_O_K;
}


/* A custom dialog procedure, derived from d_button_proc. It intercepts
 * the D_CLOSE return of d_button_proc, and calls the function in dp3.
```

```
 */
int my_button_proc (int msg, DIALOG *d, int c)
{
    int ret = d_button_proc (msg, d, c);
    if (ret == D_CLOSE && d->dp3) {
        d->flags ^= D_DIRTY;
        return ((int (*)(void))d->dp3)();
    }
    return ret;
}


/* A custom dialog procedure, derived from d_button_proc. It intercepts
 * the D_CLOSE return of d_button_proc, and calls the function in dp3.
 */
int my_button_proc_ex(int msg, DIALOG *d, int c)
{
    int ret = d_button_proc (msg, d, c);
    if (ret == D_CLOSE && d->dp3) {
        d->flags ^= D_DIRTY;
        return ((int (*)(DIALOG *))d->dp3)(d);
    }
    return ret;
}


void init_dialog_colors(DIALOG *dlg)
{
    /* set up colors */
    gui_fg_color = makecol(0, 0, 0);
    gui_mg_color = makecol(128, 128, 128);
    gui_bg_color = makecol(224, 224, 224);
    set_dialog_color (dlg, gui_fg_color, gui_bg_color);

    /* white color for d_clear_proc and all the other procs */
    dlg[0].bg = makecol(255, 255, 255);
    for (int i = 1; dlg[i].proc; i++)
        if (dlg[i].proc == d_text_proc ||
            dlg[i].proc == d_ctext_proc ||
            dlg[i].proc == d_rtext_proc)
            dlg[i].bg = dlg[0].bg;
}


void render_waveform(BITMAP *bmp, unsigned short *data, unsigned length, u
    clear_to_color(bmp, bg);
    if (data==NULL)
        return;

    unsigned w = bmp->w;
    unsigned h = bmp->h;
    unsigned limit = length/2;
    unsigned j, s1, s2;
    unsigned v;
    for (int i = 0; i < w; i++) {
        s1 = xtosample(i,limit,w);
        s2 = xtosample(i+1,limit,w);
        v = 0;
```

```
            for (j = s1; j < s2; j++) {
                v += (((unsigned long)data[j])*h)>>16;
            }
            v /= (s2 - s1);
                vline(bmp, i, h/2, v, fg);
                vline(bmp, i, h/2, h-v, fg);
        }
}

int d_waveform_proc(int msg, DIALOG *d, int c)
{
    char **buffer  = (char **)d->dp2;
    int *buffer_size = (int*)d->dp3;

    switch (msg) {
      case MSG_START:
        if (d->d1 < 10)
            d->fg = makecol(255, 0, 0);
        else
            d->fg = makecol(0, 0, 255);
        d->dp = (void*)create_bitmap(d->w, d->h);
        render_waveform((BITMAP*)d->dp, (unsigned short*)*buffer, *buffer_

      case MSG_END:
        destroy_bitmap((BITMAP*)d->dp);
        break;

      case MSG_DRAW:
        blit((BITMAP*)d->dp, screen, 0, 0, d->x, d->y, d->w, d->h);
        vline(screen, d->x+editor_dialog[d->d1+3].d2, d->y, d->y+d->h-1, 
        vline(screen, d->x+editor_dialog[d->d1+2].d2, d->y, d->y+d->h-1, 
        vline(screen, d->x+editor_dialog[d->d1+1].d2, d->y, d->y+d->h-1, 
        break;

      case MSG_IDLE:
        d->flags |= D_DIRTY;
        break;
    }
    return D_O_K;
}


struct PLAYBACKSTATE {
    int waveform_index;
    char **buffer;
    int *buffer_size;
    AUDIOSTREAM *stream;
    int delta;
};


const int STREAMINGSIZE = 4096;
const int STREAMINGLIMIT = 8192;


/* d_playback_proc updates an audiostream in MSG_IDLE if dp->stream is no
 * it is a non visual element
 * DURING INIT:
```

```
 * d1 is index of associated d_waveform_proc
 *
 * WHILE PLAYING:
 * dp is PLAYBACKSTATE
 * d1 is next sample to load
 * d2 is sample to end playback
 */
int d_playback_proc(int msg, DIALOG *d, int c)
{
    PLAYBACKSTATE* pbs = (PLAYBACKSTATE*)d->dp;
    switch (msg) {
      case MSG_START:
        pbs = (PLAYBACKSTATE*)malloc(sizeof(PLAYBACKSTATE));
        pbs->waveform_index = d->d1;
        pbs->buffer = (char**)d->dp2;//editor_dialog[d->d2].dp2;
        pbs->buffer_size = (int*)d->dp3;//editor_dialog[d->d2].dp3;
        pbs->stream = NULL;
        d->dp = (void*)pbs;
        d->d1 = 0;
        d->d2 = 0;
        break;

      case MSG_END:
        if (pbs->stream) stop_audio_stream(pbs->stream);
        free(pbs);
        break;

      case MSG_IDLE:
        if (pbs->stream != NULL) {
            unsigned short *mem_chunk = (unsigned short *)get_audio_strean
            if (mem_chunk != NULL) {  // Refill the stream buffer.
                int length = MIN(d->d2 - d->d1, STREAMINGSIZE);
                if (length > 0) {
                    memset(mem_chunk, 127, STREAMINGLIMIT);
                    //d->d1 += resample_buffer_up(mem_chunk, ((unsigned si
                    memcpy(mem_chunk, ((unsigned short*)*pbs->buffer)+d->d
                    free_audio_stream_buffer(pbs->stream);
                    d->d1 += STREAMINGSIZE;
                    if (d->d1 > d->d2 ) { d->d1 = d->d2; }
                    editor_dialog[editor_dialog[pbs->waveform_index].d1+3
                        xtosample(d->d1, editor_dialog[pbs->waveform_index
                } else {
                    memset(mem_chunk, 127, STREAMINGSIZE);
                    free_audio_stream_buffer(pbs->stream);
                    stop_audio_stream(pbs->stream);
                    pbs->stream = NULL;
                }

            }
        }
        break;
    }
    return D_O_K;
}


int play(DIALOG *d)
{
    PLAYBACKSTATE* pbs = (PLAYBACKSTATE*)editor_dialog[d->d1].dp;
```

```
    if ( pbs->stream == NULL) {
        unsigned sela = editor_dialog[editor_dialog[pbs->waveform_index].d
        unsigned selb = editor_dialog[editor_dialog[pbs->waveform_index].d
        unsigned selmin = MIN(sela, selb);
        unsigned selmax = MAX(sela, selb);
        int *buffer_size = (int*)editor_dialog[pbs->waveform_index].dp3;
        unsigned sample_begin = xtosample(selmin, *buffer_size/2, editor_d
        unsigned sample_end = xtosample(selmax+1, *buffer_size/2, editor_d
        editor_dialog[d->d1].d1 = sample_begin;
        editor_dialog[d->d1].d2 = sample_end;

        char str_msg[1024];
        usprintf(str_msg, "%d %d", sample_begin, sample_end);
        alert("Playing area", "", str_msg, "Ok", NULL, 0, 0);

        pbs->delta = 0;
        pbs->stream = play_audio_stream(STREAMINGSIZE, 16, FALSE, 11025, 2
        if (pbs->stream == NULL)
            alert("Playback", "", "Couldn't begin", "Ok I'll try again", NU
    }
    return D_O_K;
}


int pause(DIALOG *d)
{
    PLAYBACKSTATE* pbs = (PLAYBACKSTATE*)editor_dialog[d->d1].dp;
    if ( pbs->stream != NULL) {
        stop_audio_stream(pbs->stream);
        pbs->stream = NULL;
    }
    else
        pbs->stream = play_audio_stream(STREAMINGSIZE, 16, FALSE, 11025, 2
    return D_O_K;
}


int stop(DIALOG *d)
{
    PLAYBACKSTATE* pbs = (PLAYBACKSTATE*)editor_dialog[d->d1].dp;
    if ( pbs->stream != NULL ) {
        stop_audio_stream(pbs->stream);
        pbs->stream = NULL;
    }
    return D_O_K;
}


void editor_window(void)
{
        init_dialog_colors(editor_dialog);
    do_dialog(editor_dialog, -1);

    if (buffer) { free(buffer); }
        if (second_buffer) { free(second_buffer); }
    if (clipboard_buffer) { free(clipboard_buffer); }
```

```
}
```

The capture module is responsible for capturing data from the recording device and allocating data during recording.

Listing B.4: A basic Windows program.

```
#ifndef captureH
#define captureH

extern char* buffer;
extern int buffer_size;
void capture_window(void);


#endif
```

Listing B.5: A basic Windows program.

```
#include <allegro.h>
#include <mem.h>
#include "sndhelp.h"
#include "editor.h"

char* buffer = NULL;
int buffer_size = 0;

int start(void);
int stop(void);
int done(void);
int cancel(void);
int d_capture_proc(int msg, DIALOG *d, int c);

bool capturing;                 // if we are capturing data from the sound
int scratch_buffer_size;        // size of the intermediate scratch buffer
unsigned short *scratch_buffer; // intermediate scratch buffer where data

char samples_as_seconds[20];    // used by field 3 -- seconds
char samples_as_text[20];       // used by field 5 -- sample size of buffer

DIALOG capture_dialog[] =
{
    /* (dialog proc)     (x)   (y)   (w)   (h) (fg)(bg) (key) (flags)
(d1) (d2)     (dp)                    (dp2) (dp3) */

    /* this element is a shadow box therefore it should come before the o
    { d_shadow_box_proc,    0,    0,  600,  200,    0,    0,    0,
0,       0,    0,    NULL,                   NULL, NULL },
    { d_ctext_proc,        10,   10,  580,  180,    0,    0,    0,
0,       0,    0,    (void*)"Capture waveform", NULL, NULL },

    /* lots of descriptive text elements */
    { d_text_proc,         20,   50,    0,    0,    0,    0,    0,
0,       0,    0,    (void*)"Seconds captured:", NULL, NULL },
    { d_rtext_proc,       240,   50,    0,    0,    0,    0,    0,
0,       0,    0,    (void*)"00:00" ,          NULL, NULL },
    { d_text_proc,         20,   70,    0,    0,    0,    0,    0,
0,       0,    0,    (void*)"Size of buffer:", NULL, NULL },
    { d_rtext_proc,       240,   70,    0,    0,    0,    0,    0,
0,       0,    0,    (void*)"00000000000",  NULL, NULL },

    /* the cancel buttons and start/stop use our customized dialog procedu
    { my_button_proc,    430,  50,  160,   20,    0,    0,  's', D_EXIT,
0,    0,    (void*)"&Start",        NULL, (void*)start },
```

```
    { my_button_proc,     430, 80,  160,   20,   0,  0,  't', D_DISABLED|D
0,   0,    (void*)"S&top",          NULL, (void*)stop },
    { my_button_proc,     430, 140, 160,   20,   0,  0,  'd',
D_EXIT,       0,   0,    (void*)"&Done",          NULL, (void*)done },
    { my_button_proc,     430, 170, 160,   20,   0,  0,  'c', D_EXIT,
0,   0,    (void*)"&Cancel",        NULL, (void *)cancel },

    /* the record proc consumes all cpu cycles ^_^ */
    { d_capture_proc,     20,  90,  390,   90,   0,  0,    0,
0,       0,   0,    NULL,                    NULL, NULL },
    //{ d_yield_proc,       0,   0,    0,    0,   0,  0,    0,
0,       0,   0,    NULL,                    NULL, NULL },
    { NULL,                 0,   0,    0,    0,   0,  0,    0,
0,       0,   0,    NULL,                    NULL, NULL }
};

/* Used by the start button. */
int start(void)
{
    capture_dialog[6].flags |= D_DISABLED;
    capture_dialog[7].flags &= ~D_DISABLED;
    capture_dialog[7].flags |= D_DIRTY;
    capturing = true;
    return D_O_K;
}


/* Used by the stop button. */
int stop(void)
{
    capturing = false;
    capture_dialog[7].flags |= D_DISABLED;
    capture_dialog[6].flags &= ~D_DISABLED;
    capture_dialog[6].flags |= D_DIRTY;
    return D_O_K;
}


/* Used by the done button. */
int done(void)
{
    if (capturing) {
        if (alert("Paused", NULL, "Are you sure you want to stop?", "&Yes'
            return D_CLOSE;
        else
            return D_O_K;
    }
    return D_CLOSE;
}


/* Used by the cancel button. */
int cancel(void)
{
    if (capturing) {
        if (alert("Paused", NULL, "Are you sure you want to stop?", "&Yes'
            return D_CLOSE;
```

```
        else
            return D_O_K;
    } else {
        return D_CLOSE;
    }
}


/* Used to update the display of samples captured */
void update_display(void) {
    //int sec = buffer_size / (8000 * 2);
    int sec = buffer_size / (11025 * 2);
    int min = sec / 60;
    usprintf(samples_as_seconds, "%02d:%02d", min, sec%60);
    usprintf(samples_as_text, "%dk", buffer_size/1024);

    capture_dialog[3].dp = samples_as_seconds;
    capture_dialog[5].dp = samples_as_text;
    capture_dialog[3].flags |= D_DIRTY;
    capture_dialog[5].flags |= D_DIRTY;
}


/* Used to capture data */
int d_capture_proc(int msg, DIALOG *d, int c)
{
    switch (msg) {
      case MSG_START:
        d->bg = makecol(255,255,255);
        d->fg = makecol(255,0,0);
        update_display();
        scratch_buffer_size = start_sound_input(11025, 16, 0);
        if (scratch_buffer_size > 0) {
            scratch_buffer = (unsigned short*)malloc(scratch_buffer_size);
        } else {
            allegro_message("sound_card_not_capable_of_8khz_16bit_capture"
            return D_CLOSE;
        }
        d->dp = (void*)create_bitmap(d->w, d->h);
        break;

      case MSG_END:
        stop_sound_input();
        free(scratch_buffer);        // no need to check for null as it is
        scratch_buffer = 0;          // to be here if scratchbuffer is allo
        destroy_bitmap((BITMAP*)d->dp);
        break;

      case MSG_DRAW:
        {
            int w = MIN(d->w, scratch_buffer_size/2);
            int y, bg = d->bg, fg = d->fg;

            for (int i=0; i<w; i++) {
               if (scratch_buffer[i] < 0x8000) {
                  y = scratch_buffer[i]*d->h/65535;
                        vline((BITMAP*)d->dp, i, 0, y-1, bg);
                        vline((BITMAP*)d->dp, i, y, d->h/2, fg);
```

```
                        vline((BITMAP*)d->dp, i, d->h/2+1, d->h, bg);
                    }
                    else {
                        y = scratch_buffer[i]*d->h/65535;
                        vline((BITMAP*)d->dp, i, 0, d->h/2-1, bg);
                        vline((BITMAP*)d->dp, i, d->h/2,y, fg);
                        vline((BITMAP*)d->dp, i, y+1, d->h, bg);
                    }
                }
                blit((BITMAP*)d->dp, screen, 0, 0, d->x, d->y, d->w, d->h);
            }
                break;

        case MSG_IDLE:
            if (read_sound_input(scratch_buffer) && capturing) {
                    if (!buffer && buffer_size == 0) // fix up realloc of NULL
                        buffer = (char*)malloc(scratch_buffer_size);
                    else
                        buffer = (char*)realloc(buffer, buffer_size + scratch
                    memcpy(buffer+buffer_size, scratch_buffer, scratch_buffer
                    buffer_size += scratch_buffer_size;
                update_display();
            }
            d->flags |= D_DIRTY;
            break;
    }
    return D_O_K;
}


void capture_window(void)
{
    int start_buffer_size = buffer_size;

    capturing = false;
    scratch_buffer_size = 0;

    centre_dialog(capture_dialog);
    init_dialog_colors(capture_dialog);
    int ret = popup_dialog(capture_dialog, 6);
    if (ret == -1 || ret == 9) {
        if (buffer) {
            if (start_buffer_size == 0) {
                free(buffer);
                buffer = NULL;
                buffer_size = 0;
            } else {
                buffer = (char*)realloc(buffer, start_buffer_size);
                buffer_size = (buffer != NULL) ? start_buffer_size : 0;
            }
        }
    }
}
```

The helper module contains helper functions that belong to no particular module and are used throughout the application.

Listing B.6: A basic Windows program.

```
#ifndef sndhelpH
#define sndhelpH

void play_test_tone(void);
int resample_buffer_down(unsigned short *dest, unsigned short *src, int le
int resample_buffer_up(unsigned short *dest, unsigned short *src, int des
void write_buffer_to_file(char *filename, char *buffer, int buffer_size);
void read_buffer_from_file(char *filename, char* &buffer, int &buffer_size
#endif
```

Listing B.7: A basic Windows program.

```
#include <allegro.h>
#include <speex/speex.h>
#include <math.h>
#include <mem.h>
#include <stdio.h>
#include "sndhelp.h"

void play_test_tone(void)
{
    int dwLength = 36000;
    float  amp[] = {8000.0f, 8000.0f, 8000.0f, 8000.0f};
    unsigned begin[] = {0, 5000, 10000, 15000};
    float  factor[] = {M_PI/60.0, M_PI/50.0, M_PI/40.0, M_PI/30.0};
    SAMPLE *smp = create_sample(16, 0, 44100, dwLength);
    if (smp == NULL) { allegro_message("couldn't create sample buffer!");
    short *lpWrite = (short*)smp->data;  // Sound card data is unsigned b
                                         // calculating signed data is ea
    for (unsigned int i = 0; i < dwLength; i ++) {
                                lpWrite[i]  = (short)(amp[0]*sin(i*factor[0
            if (i>begin[1]) { lpWrite[i] += (short)(amp[1]*sin(i*factor[1
            if (i>begin[2]) { lpWrite[i] += (short)(amp[2]*sin(i*factor[2
            if (i>begin[3]) { lpWrite[i] += (short)(amp[3]*sin(i*factor[3
            ((unsigned short*)lpWrite)[i] ^= 0x8000;
    }
    int voice = play_sample(smp, 255, 128, 1000, 0);
    if (voice < 0) { allegro_message("no voices available!"); }
    while (voice_get_position(voice) >= 0);
    destroy_sample(smp);
}


#define FRAME_SIZE 160
void write_buffer_to_file(char *filename, char *buffer, int buffer_size)
{
    // buffer and buffer_size are length of the sample buffer in bytes,
    // not samples, so first we have to convert before doing math on it.
    // We pass arrays around this way to prevent allocation confusion.
    unsigned short *short_buffer = (unsigned short*)buffer;
    int short_buffer_size = buffer_size / 2;

    // speex manual says we should resample from 11025 Hz to 8000 Hz
    // before saving, but the codec seems to handle it.
    float input[FRAME_SIZE];
    char data[256];
    int numbytes;
    void *enc_state;
    SpeexBits bits;
```

```
    int pos = 0;
    int tmp = 8; // (15 kbps)
    int len;
    FILE *fp = fopen(filename, "wb");
    if (!fp) { alert("Error", "", "Couldn't write to file or file in use b
    enc_state = speex_encoder_init(&speex_nb_mode);
    speex_encoder_ctl(enc_state, SPEEX_SET_QUALITY, &tmp);
    speex_bits_init(&bits);
    while (true) {
        len = MIN(FRAME_SIZE, short_buffer_size - pos);
        if (len < FRAME_SIZE) { break; } // chuck out last frame
        // copy 16-bit values to float so speex can work on them
        for (int i = 0; i < len; i++)
            input[i] = short_buffer[pos + i] - 0x8000;
        pos += len;
        // flush the bits the struct to work on a new set
        speex_bits_reset(&bits);
        // encode the frame
        speex_encode(enc_state, input, &bits);
        numbytes = speex_bits_write(&bits, data, 256);
        fwrite(&numbytes, sizeof(int), 1, fp);
        fwrite(data, 1, numbytes, fp);
    }
    // clean up
    speex_encoder_destroy(enc_state);
    speex_bits_destroy(&bits);
    fclose(fp);
}

void read_buffer_from_file(char *filename, char* &buffer, int &buffer_size
{
    // buffer and buffer_size are length of the sample buffer in bytes,
    // not samples, so first we have to convert before doing math on it.
    // We pass arrays around this way to prevent allocation confusion.
    short *short_buffer = (short*)buffer;
    int short_buffer_size = buffer_size / 2;

    float output[FRAME_SIZE];
    char data[200];
    int numbytes;
    void *dec_state;
    SpeexBits bits;
    FILE *fp = fopen(filename, "rb");
    int tmp = 1;
    if (!fp) { alert("Error", "", "Couldn't open file or file in use by ar
    buffer_size = 0;
    dec_state = speex_decoder_init(&speex_nb_mode);
    speex_decoder_ctl(dec_state, SPEEX_SET_ENH, &tmp);
    speex_bits_init(&bits);
    if (buffer == NULL) {
        short_buffer = (short*)malloc(4); // fix up realloc of NULL bug
        buffer = (char*)short_buffer;
    }
    while (true) {
        fread(&numbytes, sizeof(int), 1, fp);
        if (feof(fp)) { break; }
```

```
            fread(data, 1, numbytes, fp);
            // decoder frame
            speex_bits_read_from(&bits, data, numbytes);
            speex_decode(dec_state, &bits, output);
            // output to buffer
            short_buffer = (short*)realloc(buffer, buffer_size + (FRAME_SIZE*
            buffer = (char*)short_buffer;
            for (int i=0; i<FRAME_SIZE; i++) {
               short_buffer[buffer_size/2 + i] = output[i];
               short_buffer[buffer_size/2 + i] ^= 0x8000;
            }
            buffer_size += FRAME_SIZE*2;
        }
        speex_encoder_destroy(dec_state);
        speex_bits_destroy(&bits);
        fclose(fp);

}
```