LRN# 03E001I

Project Number: MXC-I003 - 43

LINUX FOR THE BLIND

An Interactive Qualifying Project

Submitted to the faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Batchelor of Science

By

Nicholas A. Pinney

[illegible]

Owen P. Smith

Abraham M. Shultz

Date: April 27, 2003

1. Screen Reader
2. Linux
3. Speakup

Approved:

[illegible]

Professor Michael Ciaraldi

Abstract

Out project reduced the total cost of ownership of a computer for a blind Linux user. We accomplished this by making an existing screen reader (Speakup) able to communicate with an existing text-to-speech program (Festival). A set of open-source programs are now available that allows a blind user to interact with a Linux terminal without the need for expensive speech synthesis hardware.

Authorship

Nicholas A. Pinney

Nicholas is a 3rd year undergraduate at WPI studying for a Bachelor and a Master of Science degree in Computer Science. He designed and wrote parts of the Speakup driver, helped with the design of the middleware program, and wrote and edited portions of this paper.

Owen P. Smith

Owen is a 3rd year undergraduate at WPI studying for a Bachelor of Science degree in Computer Science. He wrote parts of the Speakup driver, designed and wrote the middleware program and wrote and edited portions of this paper.

Abraham M. Shultz

Abe is a 3rd year undergraduate at WPI studying for a Bachelor of Science degree in Computer Science. He helped design and write the Speakup driver, and wrote and edited portions of this paper.

Table of Contents

1 Introduction

Although much research has been done on improving the user interface of computers, little has been done to make computers accessible to the visually impaired or the blind. On most systems, the only input is a keyboard and mouse and the only output is a monitor. To use a mouse, the user must know where the pointer is located in relation to other items on the screen. Much of the data on the screen is displayed to the user as graphics, such as buttons, scroll bars, and visual representations of files and programs. For a visually impaired person, there are methods to make the data on the screen more readable, but to a blind person, the data on the screen is useless.

In recent years, some major software manufacturers have made additions or modifications to their products to assist the visually impaired. Software is available for Linux and Windows that will enlarge parts of the screen. Windows offers accessibility options such as very large text and high contrast images to allow people with poor vision to read their computer screens. The X windowing system for Linux can also be configured to have high contrast color schemes and large fonts. However these large text solutions can cause problems when the software was written with the assumption that the text would be of a certain size, which occurs in both windows and Linux applications. Also these solutions will only work the those whose vision is below normal but still partially functional, the visually impaired, it does nothing for the fully blind.

Unfortunately, much of the software for blind users is expensive. The limited audience combined with potentially expensive licensing fees can raise the price above what the average person can afford. One such package, JAWS, is a very full-featured screen reader for Microsoft Windows. It includes support for Braille hardware, multiple languages, and many popular windows programs as well as an integrated Text-To-Speech program. JAWS also has a talking installation program so that a blind user can install it. JAWS for Windows 95/98/Me and XP home costs $895 dollars; the professional version costs $1,095 and supports Windows NT and 2000 as well as Win95/98/Me [A1].

The X windows system for Linux has functionality that would allow a system similar to JAWS to be created for it. Versions of X since X11R6 have included a system of "hooks" to allow a screen reader or similar program to interpret the data on the screen [A2]. The Mercator Project and the National Security Agency, the NSA, were exploring this alternative as of approximately 1995 [A2]. The NSA project was called Sonic X [A2]. The Mercator and Sonic X projects developed into Ultrasonix, which was implemented on Sun's Sparc architecture. In 1996-97, there was some interest in porting Ultrasonix from Sparc to Linux, but this project appears to have failed or lost support.

Alternative input and output devices are available to blind or vision impaired users. A few manufacturers produce Braille keyboards and Braille terminals. There are also hardware voice synthesizers available. The main drawback to these devices is that they are extremely expensive. The DecTalk Express is one of the best-known hardware speech synthesizers. It is very expensive, but very versatile. Despite DEC's demise they are still available through www.artictech.com. With appropriate software it is supported by both Linux and Windows and can speak multiple languages [A3]

As a substitute for expensive hardware, it is possible to have software do most of the work of converting the screen into speech. The speech may be played through speakers connected to the PC sound card. While Ultrasonix and JAWS allow a screen reader to work with a GUI, many simple screen readers only support a command line interface (CLI). With a CLI, the user types the names of commands, and the computer executes them and displays the results as text. All the input and output is text, so a screen reader can read it faster and more accurately. A text interface also eliminates graphic objects like scroll bars and buttons, which cannot be easily expressed in speech.

However, the software must be put on the computer before blind people can start using it. It appears that the market is small enough that major computer manufacturers have not considered it or decided that the niche market is to small to be worth targeting. It may well be that the small market size would put the price of a pre-configured system above many people's budgets. The only other alternative is for blind people to rely on a sighted person to install and initially configure it. This may be difficult, since most sighted people are used to using a GUI to interact with a computer.

2 Background

The goal of this project is to create a cheap and easily distributable software system to facilitate the use of a computer by a visually impaired or blind person. This would serve the larger goal of reducing the total cost of ownership of a computer for a blind person to the expense of the computer itself and some speakers or headphones. This system needs to run on common hardware to keep the cost down. Ideally, it should be able to run on any modern home computing system. It needs to be able to run a broad range of applications and should not be restricted to running code that has been written specifically for it.

In planning this project, it became clear that the best operating system for our needs was Linux. Linux runs on most commonly available hardware, is free, and is open source. Since Linux can run on almost all modern home computers, most people who have a computer can switch to Linux without purchasing additional hardware. Many distributions of Linux are designed for the IA32 family of processors, such as all Intel Celeron and Pentium series and the AMD Duron and Athalon._There are also distributions for Apple hardware, SPARC, and MIPS architectures, as well as the Playstation 2 gaming console [A4]. There are several other open-source operating systems available, such as FreeBSD, but Linux is the most commonly used on desktop systems and is arguably the easiest to modify.

Anyone who wants Linux can download a copy from the Internet for free. If a user wants to spend money, they can purchase a book that comes with a Linux distribution for about $30. In contrast, recent versions of Windows cost as much as $200 to $300 [A5]. Although this price difference seems appealing, most people never encounter it as they purchase computers with the operating system preinstalled.

The open source aspect of Linux was another major motivation in selecting it for our project. The entire source code for Linux and the programs we planned to use is freely available to anyone who wants it. There is no need to reverse engineer or hack the programs to modify them. Linux and many Linux programs have no licensing issues to prevent releasing modified versions of the software free of charge or sharing the source code with anyone who is interested. In contrast, systems such as Windows charge developers large fees and/or make them sign non-disclosure agreements before they can have the source code. Since we did not have to deal with licensing or non-disclosure agreements, our project cost no money to develop and may be freely modified and improved by others.

2.1 Screen Readers

There are several pre-existing screen reading packages for Linux that were available prior to our project. All of them have some advantages and disadvantages, and several of the most commonly used ones are examined here. Ideally a screen reader should be able to speak any errors that occur while the Linux kernel is loading and should be light enough in size to be able to be included on a boot floppy.

2.1.1 Emacspeak

Emacspeak runs as a subsystem of Emacs, a text editor for Unix-like systems that has grown massively in functionality over the years of its development. Emacs provide users with the ability to edit text, but it also has an enormous amount of other functionality built into it and available as separate modules. It is entirely possible to start a session of Emacs and do everything that you could do from the Linux command line, and more, without leaving Emacs.

Instead of simply reading the screen, Emacspeak also creates "auditory icons". According to the homepage of the Emacspeak project, "Spoken information is overlaid with changes in voice characteristic and inflection and combined with non-speech auditory icons to produce rich aural presentations that significantly enhance the band-width of aural communication" [A6].

Emacspeak does a significant amount of processing of the screen contents before creating output to the user. It can recognize most common devices, and consider that context when it is formatting the output. In addition to this automatic recognition, there are also modules to extend Emacspeak and improve its audio formatting for specialized tasks such as coding and browsing the Internet [A7].

However, Emacspeak is not entirely suited to our project for two reasons. First, a strong knowledge of how to use Emacs is required to use Emacspeak. Emacs is a very complex program and has a considerably high learning curve. Although it is quite popular, many people chose not to use it, favoring either a simpler text editor or another editor of comparable functionality, such as vi. Requiring Emacs to be running would effectively force the users to use specific software, a limitation that goes against one of the goals of this project.

Second, the hardware requirements of a heavily enhanced version of Emacs can be considerable. Although any current system would be able to run Emacspeak and all of the modules a user might want to use, legacy systems may be unable to effectively run such a modified version of Emacs. Moreover, as Emacs is a normal userland process using it for screen reading while the kernel is booting would be impossible. Finally, creating a boot CD that would allow Emacspeak to run on almost any system could be quite difficult._The boot CD would need to autodetect the type of sound card the user had and load the appropriate driver. If this failed, the system would be unresponsive. An installation script that ran within Emacs would also need to be created. While this is possible, there are preexisting text-based installation scripts, so the creation a new one would be an unnecessary effort.

2.1.2 YASR

As its name suggests, Yet Another Screen Reader, YASR, is a simplistic screen reader for Linux. It works by opening a terminal and running a shell in it. It monitors all the input and output to that terminal and maintains a virtual screen containing a copy of the user's screen. It refers to the virtual screen when the user requests text to be spoken.

YASR supports four hardware speech synthesizers and software voice synthesis through Emacspeak or the Flite speech synthesizer. However, must like Emacspeak, it is a purely userland process solution and is incapable of providing screen reading capability as the kernel boots.

2.1.3 Speakup

Speakup is an extensible screen reading package for the Linux kernel. It uses a modular design and currently supports eleven hardware speech synthesizers. Additional synthesizer drivers may be added as modules through a reasonably simple API. It is completely open source under the GPL, allowing us to develop for and modify it as needed.

Speakup is compiled into the Linux kernel, which makes it available as soon as the kernel and the sound card drivers have been loaded and initialized. This is important as it allows the user to hear if something goes wrong while the kernel is booting. Although there is room for something to go wrong prior to the sound drivers being loaded, this is the earliest time in the boot sequence at which a software screen reader can reasonably start, as explained later in the Screen reader start time section.

Speakup is currently in active development and is constantly being enhanced by its development team. Our group was able to contact the main developer of Speakup, Kirk Reiser, and found him to be very responsive and helpful to our queries. Although the documentation on how to write a new module for Speakup is quite limited, Kirk's helpfulness and availability was invaluable for developing ours.

2.2 Speech Synthesizers

There is quite a variety of programs available for Linux that can synthesize speech. Although there are many ways to classify speech synthesizers, for our purposes it is sufficient to consider three aspects: quality of the output, size of the program, and format of the input.

Obviously, the quality of output needs to be good to be useful as a part of a screen reading package. Although it is possible to become used to a given speech synthesizer's output, it is far preferable to start with a high quality one. Additionally, it is fairly important that the rate at which the synthesizer talks be adjustable as many screen reader users prefer to adjust this to as high as possible within the limits of their hearing. However, in general a higher quality voice requires a larger program to generate it.

Size is a consideration for a screen reader that is capable of speaking immediately upon kernel startup. Although when installed on a hard drive it is possible for a Linux kernel to be arbitrarily large, we would like to be able to have our screen reader be able to fit on a single-floppy kernel to allow for use when installing from an install floppy or CD.

For synthesis, speech is usually broken down into atomic utterances called phonemes. These basic units of speech are similar to syllables in words; often a single syllable will be made up of one or two phonemes of pronunciation. Almost all speech synthesizers create their voice by stringing series of phonemes and pauses together.

Input to speech synthesizers can be loosely broken into two categories: those that take a string of phonemes in for pronunciation and those that take in a string of text and interpret the sequence of phonemes from that. Interpreting text streams into a stream of phonemes is a complex subject and varies considerably not only by language but by dialect and accent. As such, we will only be considering speech synthesizers that process text into phonemes as a part of the speech synthesis package. Programs such as this are more correctly termed Text-To-Speech systems as by the technical definition speech synthesizers only deal with phoneme to sound conversions. This both greatly simplifies this project and allows us to rely on far more advanced phoneme generation algorithms than we would be able to provide ourselves.

2.2.1 TuxTalk

TuxTalk is a very small voice synthesis program for Linux made freely available under the GPL. Although the output is rather poor, its small size makes it ideal for embedding into the kernel and for use on low-memory systems. Additionally, the program's current designer, Kirk Reiser, was very responsive and helpful with our queries regarding the program.

Developed by the same lead programmer as Speakup, TuxTalk was designed to be adaptable for use with that screen reader. Although it does not provide all of the features of a larger speech synthesizer, it is enough for basic screen reading and can interpret several control sequences for annotating the spoken text with modified pronunciation. It accepts a stream of plain ASCII text to be spoken on standard input, and so would be easy to adapt to accept input from a screen reader.

TuxTalk has been designed with size and the potential to be made into a kernel module in mind. Although it does have many external library dependencies, most of those are either already available within the Linux kernel or are easy to include as a part of the code. It does make extensive use of floating-point arithmetic which should be avoided in the Linux kernel, but that restriction may be ignored as a FPU would be included in any system with enough processing power to support software screen reading.

Some of the central design of TuxTalk is based on speech synthesis work done at Digital Equipment Corporation in the early 1980s. The original designers of the system are unavailable to comment on their code, and it is suspected that they have retired or are no longer active in the software development community [O1].

However, the current designer, Mr. Reiser, was quite helpful when initially approached with the idea of modifying TuxTalk to work with Speakup fully and gave several useful hints and an interesting history of the project to date. The apparent ease with which TuxTalk could be modified to suit our needs when combined with the assistance of Mr. Reiser made TuxTalk an excellent choice for our kernel-level speech synthesis kernel program.

2.2.2 Festival

Festival is a high-quality speech synthesizer freely available for a number of systems including Linux. It is highly configurable and has multiple voices available in several different languages. Although it cannot be easily adapted to a Linux kernel module, its quality of output and rich feature set make it a good choice for our main speech synthesizer.

Festival was created partially as a text to speech system and partially as a test bed for new voice synthesis methods and voices. It has several very complete interfaces ranging from an interactive command line interface to language bindings for C, C++, LISP, and SCHEME. These language bindings make it very easy to work with and facilitate interfacing it with a screen reader.

Festival was designed for, "those who simply want high quality speech from arbitrary text with the minimum effort and those who are developing large systems and wish to include voice synthesis output." [O3] All aspects of Festival's spoken output can be tuned both in general and for specific utterances, making it perfect for screen reader output. Furthermore, the variety of available voices and languages makes it quite comfortable to work with.

The one drawback to Festival is that it is quite large and rather complex to install. Its size prevents it from being made into a kernel module and makes it unusable on some older systems, but it should work fine on any system made in the last few years.

Compiling and installing it is a non-trivial task, however its license permits us to generate and distribute a pre-compiled version for easier install.

Analysis

Many pre-existing software packages with many qualities to them that needed consideration before deciding upon a particular method of completing our goals. All of the components required to create a complete screen reading system were already available when we started this project, usually implemented in several ways across several different software packages. Qualifying the differences between these packages and deciding upon which to use comprised a considerable portion of our work.

3.1 Design Requirements

Several aspects of the design of a screen reader affect its usefulness. It must be robust as it may be the user's only means of receiving information from their computer. It must be fairly simple and quick to use, but needs to be reasonably rich in features. Finally, the stage of the boot process at which the screen reader starts is a consideration.

For many programs, it is acceptable for crashes to occasionally occur; while they may be annoying and some data may be lost, it is generally quite possible to recover from such a problem. It is also acceptable for a program that received incorrect arguments to just spit out a help message on screen. However, this would be unacceptable behavior for a screen reader or similar accessibility-enabling program

As it is possibly the only means of getting data from a computer to the user, it is vital that a screen reader be very strongly tested and that it provide audible feedback in the event of an error. Furthermore, if the reader does crash it, it is important that it automatically start itself again, possibly having spoken an error message to the user. Finally, it must be reasonably simple to set up, as if something does go wrong it is possible that it will need to be started with no feedback to the user.

Finally, it is important that a screen reader start talking as soon as is possible. Until it starts it is quite possible that the user is receiving no feedback from their computer other than the sound of their hard drive clicking. Under normal working conditions on a computer the user doesn't require any feedback until they log in, but for testing purposes, e.g. when trying out a new kernel, it would be desirable to be able to listen to the kernel boot messages.

3.2 Screen reader start time

An ideal screen reader would start talking the moment your computer turned on. However, this is not possible in a purely software driven reader. We analyzed the boot process of a Linux system running on an Intel architecture machine and identified several stages in the boot process at which a screen reader theoretically could be started:

3.2.1 System Power-Up

This would be the ideal time at which to have a screen reader start, but it would be nearly impossible without the aid of additional hardware. The only storage that is available at this point is the system's BIOS; although it could be possible to embed a screen reader in there it would be a very non-portable solution. Such a reader would likely require altering for each different chipset or motherboard that it was adapted to, and would be dependant on the sound hardware in the system. Additionally, the available storage for the speech synthesizer would be very small, forcing poor output. Finally, if there was an error in the code for this it might render the system inoperable.

3.2.2 Linux Loader Boot Prompt

A boot loader, the most common being LILO, the Linux Loader, loads the Linux kernel. Although the storage space of the boot disk drive would be made available to a screen reader at this point, the sound hardware for the system would still be uninitialized and unavailable. A specialized boot loader could likely be used at this point, however this would require considerable research and coding and again would be largely sound hardware dependent.

There is little information that needs to be provided to the user at this point; for the majority of cases the user just hits enter or waits for the default kernel to load after a brief timeout. At most, the user will be entering a short string of text to configure their kernel. Even if it were considerably easier to implement a screen reader at this point, it would be a low-priority feature, mainly useful for debugging a system.

3.2.3 Linux Kernel

The Linux Kernel presents the first strongly feasible place to start a screen reader. Sound card initialization could be promoted to occur early in the system's initialization and a speech synthesizer could be started immediately after that. Additionally, access to the screen and keyboard will be available through the kernel in a standard way that will be used by all textual applications.

Running the screen reader on this level also offers additional advantages. As it would be a part of the Linux kernel process, it gains full access to all aspects of the system. Specifically, reading text from the screen and taking control of a part of the keyboard can be done with the highest degree of access at this system level.

Storage space at this point becomes dependant on the boot media. If booted from a hard drive the kernel image can be any size, and as such may contain a sophisticated speech synthesizer. However, if booted from a floppy disk the space is limited to 1.44Mb; although this is ample space for a modularized kernel with an embedded screen reader, it only leaves space enough for a small speech synthesizer

3.2.4 Userland processes (Init script)

The simplest and most flexible way to run any program is to run it as a normal userland process and not attempt to place it into the kernel. The kernel provides some ways to access the contents of the screen and ways to bind a set of keys to a specific application. Additionally, no restrictions are placed on the size of such a program and it may use the standard interface to the sound hardware. Finally, most speech synthesis software that is freely available runs at this level.

However, this level does not start until late in the boot process. If there was an error while booting or if a kernel panic occurs while the system is running, a screen reader running at this level would be unable to provide any audio feedback.

3.3 Data Passing

There are a number of ways to pass a stream of data from the kernel to a userland program, all of which have both advantages and disadvantages. The primary concerns for each method proposed was ease of implementation, adaptability, consistency with pre-existing code, and ease of use. With these concerns in mind, we examined three basic ways to transfer the information to the user and eventually decided to use the character device.

3.3.1 Character Device

Character devices are designed for data input and output as a continual stream, sending data to and from userland processes in arbitrarily sized chunks. These devices are designed for use with any data stream that does not have a strict structure imposed on the act of reading or writing. They are ideal for any continuous data stream and may be accessed using standard Unix file semantics. They are designed for transferring an unstructured data stream such as the output from a screen reader in a buffered fashion.

Character devices do have one downside, however. They are accessed by referring to their major and minor numbers; each class of character devices has its own major number and each device has its own minor number. At present, these numbers are restricted to a maximum of 255, and most major numbers are in use. This means that the major number could not be made consistent across all machines, making a standardized install program harder to write.

However, work is in progress to fix the current crunch of major numbers in the Linux kernel. Future versions will have the restriction on major numbers relaxed, and at present an alternative is available in the form of the device file system, which allows devices to be registered by hierarchical name instead of by an arbitrary index.

Although there are drawbacks to using a character device, it is the most logical choice for passing data from a kernel screen reader to a userland speech synthesizer. Being able to utilize file semantics greatly simplifies the interface and all of the current limitations of character devices are being corrected for future versions of the Linux kernel. Additionally, most hardware speech synthesizers already supported by Speakup are accessed over the serial port in this fashion.

3.3.2 System Call

System calls provide direct low-level access to the kernel's functionality. A system call passes data directly to the kernel, requesting that some action be taken by the kernel for the process. System calls are a very fast way of communicating with the kernel and provide the most flexible method of communication between the kernel and a userland process.

However, the system call interface does not provide any functionality past the ability to send small pieces of data rapidly from a process to the kernel. The programmer must implement any required semantics for passing data to a userland program.

Although more flexible and universally accessible than a character device, a system call does not support any of the standard file IO abstractions. This is a notably different method of data passing from the format used by other Speakup drivers and would be more difficult to implement than a character device would be.

3.3.3 /proc Filesystem

The /proc file system provides a simple interface for monitoring and adjusting parameters of the Linux kernel. It was designed to provide real-time information on the state of a running kernel using standard file semantics for access. Although it appears to be a normal file system to the user, the directories and file contents are generated on access to reflect the current state of the system.

Speakup uses the /proc file system for several tunable variables for the system in general and for each driver in specific. Although it would be possible to use it for passing data from our driver to a speech synthesizer, this would be an unusual use for it.

4 Procedure

The design of our code proved to be considerably more of a challenge than we had expected, mostly because none of us were familiar with Linux kernel coding before we started the project. The following is an overview of our design process and the design itself. For a more detailed explanation of our code as of the time of this writing, see Appendix B.

4.1 Pre-existing Software

'Laziness is a virtue' is said to be one of the tenets of computer programming. If it is possible to put a system together using pre-existing software then it is usually best to do so. As such we decided to take three pieces of existing software and use them in our design. All of these software packages are available under the GPL, a license acceptable to all Linux users.

4.1.1 Speakup

Speakup is a robust and feature-rich screen reader, but it currently only works with hardware speech synthesizers. It was designed to work with multiple different synthesizers, though, and has a strong modular design. As this screen reader already does a good job of interacting with the user and processing the contents of the screen, we decided to write a driver for it to let it communicate with a software speech synthesizer.

4.1.2 TuxTalk

Although the quality of its output is somewhat poor, TuxTalk is very small and appears to be well suited for adaptation into a kernel module. It currently only runs as a userland process but would require only minimal modifications to its core to become a part of the kernel. As such we decided to try and adapt it to function as a minimal speech synthesizer for times when using a userland synthesizer would not be available.

4.1.3 Festival

The Festival Speech Synthesis System is fully featured and the output is good. It provides a flexible API that allows us to reproduce all of the speech output features of a hardware synthesizer with a simple layer of glue code. It provides a variety of voices and intonation control that would allow the user to heavily configure and customize the sound of the outputted speech. While our design allows for the use of any userland software synthesizer to be easily added, Festival meets all of our needs and provides a rich set of additional features for the user.

4.2 Initial Design - Two-Synthesizer Approach

Our analysis of screen reader start times suggested that there were two optimal times for a reader to start: The most robust solution would have the screen reader start as soon as possible as the kernel boots, but the best synthesized speech is only available as a userland process. After some consideration, we decided to go with a toggling two-synthesizer design to gain the benefits of both approaches. Our design resulted in several modular parts:

Speakup provides us with a stream of text to be spoken. All of the code for receiving input from the user and parsing the contents of the screen is in the Speakup core code; we never need to know how the internals of the kernel handle keyboard input or screen output, or for that matter any more than the basics of how Speakup processes that text.

This stream of text needs to be sent to one of two locations; when possible it should go to the userland synthesizer, and otherwise it should be sent to TuxTalk inside the kernel. Initially we had planned to have this switching functionality be a separate module but it rapidly proved easiest to just make that a part of our Speakup module.

TuxTalk would be loaded as a kernel module and would be always on. Ideally, it would be loaded directly after the sound system and directly before Speakup so that it would be ready to speak as soon as possible and be ready when Speakup starts talking. Passing text to it to be spoken would be handled with a function call.

A character device is used to send text from Speakup to the userland speech synthesizer. This allows the data stream to be accessed as a file, a common and powerful approach among Linux and Unix device drivers. Additionally the use of file semantics allows us to easily know if the userland synthesizer is active. Finally, if the userland program crashes then the Linux kernel will automatically close the character device, further simplifying the implementation.

To further generalize our program we added a small layer of middleware to go between Speakup and the speech synthesizer. This allows our code to be adapted to a variety of different speech synthesizers with minimal modification. The middleware buffers, interprets, and translates the text stream from our character device into Festival-specific commands. Additionally, in the event that an error occurs our middleware can play pre-recorded audible error messages.

Festival is accessible through its simpler C API. In addition to initialization and deinitialization, the API provides a simple call for speaking a string of text and an additional call for executing an arbitrary command in Festival's Lisp interpreter. Festival uses a client-server model, so the Festival server must be running before the middleware program starts.

4.3 Design Revisions

Several aspects of our initial design proved to be either infeasible or insufficient. As such, our code went through several different levels of revision and in many cases, simplification.

4.3.1 TuxTalk

Unfortunately embedding TuxTalk into the kernel proved to be quite difficult. In the end, we were unable to convert it into a kernel module for reasons specified below. As such, our software screen reader implementation lacks the ability to speak as the kernel is loading. However, it would be simple to add support to our code for a kernel speech synthesizer if one should exist. In our final design, incoming text is ignored if our character device is not open but could easily be re-routed elsewhere.

The first problem we encountered in making TuxTalk into a kernel module was that it used calls from the C standard library. These calls are not available to kernel programs. One member of our team was able to circumvent most of this problem by removing or reimplementing many of these calls, and by locating several equivalent function calls already implemented for their general utility.

The second problem was that TuxTalk performed file operations that are not available within the kernel. These operations, such as open(), are written to be called as operating system calls from outside the kernel, and as such do not have analogous operations for use within the kernel. Removing these operations from TuxTalk proved impractical, as they are required for outputting synthesized voice to the sound card.

One possible work around was to locate and call the kernel functions associated with opening the sound card's character device directly. However, this would require a level of understanding of the Linux I/O system well beyond our means. As such, this plan was rejected as it would be overly complicated, very messy, and would never be accepted as a standard part of the Linux kernel.

4.3.2 Festival

The text tokenizer in our middleware went through several revisions. A tokenizer is an algorithm for taking a given string, or piece of text, and breaking it into one or more chunks for further processing. Initially we had no way to mute the output from Festival, so we tried sending text to it one word at a time. This resulted in unacceptable performance; the spoken output had no intonation and there was a notable pause between each word. A temporary solution was to pass text to Festival one sentence at a time, but then it continues to speak for a while after a 'shut up' code is received.

The apparent solution to this problem is to put Festival into an asynchronous mode wherein calls to say text return immediately, queuing the text. This allows for a simple single-threaded approach and for the ability to tell Festival to mute immediately. This is currently unimplemented due to time constraints.

4.4 Future work

There are a number of improvements planned for future versions of our work. Speakup provides a method of marking text for emphasis when spoken; support for this can be added soon as we now know how to modify the pitch of Festival's spoken output. Support for adjusting the speech rate of Festival is the most highly desired feature among our current users, and this feature will be added soon.

A complete package of Speakup, Festival, and our code should be created to simplify installation. At present, there are versions of Slackware and Redhat Linux that support Festival using a hardware speech synthesizer; customized versions of several major Linux distributions (Debian Woody, Slackware, and Mandrake) using our code are planned.

Support for the device file system would make our code be more convenient for many users to use and is a possible requirement for consideration for inclusion in the official Linux kernel. Additionally, adding support for speech synthesizers other than Festival would be simple but potentially quite useful.

Finally, our middleware program should probably be renamed to something more intuitive and suggestive of its function.

4.5 Procedure

5 Results

Overall, our project has been a success, albeit an imperfect one. Our attempt to turn TuxTalk into a module failed, but our Speakup driver still works quite well with Festival. While this does limit our code's use for system diagnostics, it is still quite good for general use and has been well received among the blind Linux community.

The TuxTalk kernel module idea failed due to problems inherent with the code and with the Linux kernel, but the userland version of TuxTalk can be used in conjunction with our userdev system for low-end systems with little modification. The only changes that are needed are to open() the device instead of using stdin and to use read() instead of fgetc() because fgetc()'s buffering of Speakup's output can cause irregularities in the speech. We have not released a version of TuxTalk with these modifications due to the poor quality of its output; we instead have support for Festival's more advanced speech synthesis.

Festival provides good quality output for our Speakup driver and is currently in use by a number of people. Its variety of features will allow us to provide all of the audible functionality of a hardware speech synthesizer in a free software solution. Although not all of these features have been implemented, we are still actively developing our software and should have it fully functional soon.

Our usrdev driver is complete and we have a userland middleware program that allows it to interface with festival easily. For our program to be work you need to be booted into a kernel that has usrdev support compiled in and with either the usrdev driver selected as the default driver or with the boot-time option speakup_synth=usrdev. Then you must run festival in server mode (not normal mode) using either festival --server or with the festival_server.sh shell script to be found in the src/scripts subdirectory of festival. After festival has successfully started in server mode using the default port of 1314 then the middleware program should be run in the background at which point speech should be enabled. Many people have found it highly convenient to put both the server startup script and the invocation of the middleware program into /etc/rc.d/rc.local or some other startup script. Since Speakup uses a statically allocated buffer and because of the large amount of data generated at boot time it was decided to not buffer and text send to Speakup until the character device had been opened. This means that once the middleware program is running no speech should occur until additional data has been generated at the current console and that the data spoken will only be the new data generated. The usrdev driver is working and has all the functionality required but still has some known problems.

6 Conclusion

Overall, the project has been a success. We have patch files available at http://users.wpi.edu/~blinux/ for download that are installable by a knowledgeable user without too much difficulty. While the TuxTalk problem was a setback, the goal of removing the $300+ hardware speech synthesizer from the needs of blind Linux users has been achieved. The total cost for a computer that a blind person can use has been significantly reduced, making this project a success.

7 Bibliography

A1: Humanware Price list. Online. Accessed 2/25/03
http://www.humanware.com/K/K2.html

A2: Novak, Mark. Accessability to the X window System. Online. Accessed 2/25/03.
http://trace.wisc.edu/docs/x_win_accessibility/x_ablity.htm

Jaws for Windows. Accessed 2/22/03
http://www.freedomscientific.com/fs_products/software_jaws.asp

A3: Fonix DecTalk Products. Accessed 2/22/03.
http://www.worklink.net/products/dectalk.html

A4: Linux for Playstation 2 Community. Online. Accessed 3/5/03.
http://playstation2-linux.com/

A5: Pricing and licensing. Online. Accessed 3/5/03.
http://www.microsoft.com/windows2000/professional/howtobuy/pricing/

A6: Raman, T. V. Emacspeak : The Complete Audio Desktop. Online. Accessed 4/20/03.
http://emacspeak.sourceforge.net/smithsonian/study.html

A7:Raman, T. V. Emacspeak : Direct Speech Access. Online. Accessed 4/20/03
http://emacspeak.sourceforge.net/publications/assets-96.html

O1: Accessed 2003-03-10 03:05
http://www.linux-speakup.org/TuxTalk.html

O2: Accessed 2003-03-10 02:56
ftp://linux-speakup.org/pub/speakup/disks/00-INDEX

O3: Accessed 2003-05-05 15:33

http://www.cstr.ed.ac.uk/projects/festival/manual/festival_5.html#SEC9

O4: accessed 2003-05-17 18:30
http://weblab.deis.unibo.it/Linux4D/english/

8 Appendix A: Our code as of 2003-04-23

8.1 middleware/Makefile:

```
OBJS= middleware.o
INCLUDE= -I/usr/local/include -I../speakup
LIBS= -lpthread
CFLAGS= $(INCLUDE) -O2
TARG= middleware

all: $(TARG)

$(TARG): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARG) $(OBJS) $(LIBS)
	strip $(TARG)

clean:
	 /bin/rm -f $(OBJS) $(TARG)
```

8.2 middleware/middleware.c:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <pthread.h>
#include "speakup_usrdev_shared.h"

#include "festival.h"

#define BUFF_SIZE  1000
#define SLEEPTIME  100000

int init_festival();
int close_festival();
int say_word(char *str);
void spawn_client(void *);
char *tokenize(char chr);

// Shared data.
char buff[BUFF_SIZE];
int buff_read = 0, buff_to_read = 0;
pthread_mutex_t mutex;

// Data used by the parent only.
int file = 0;
int child;

int main (void) {
  pthread_t speaker;

  // Init file
  file = open("/dev/usrdev", O_RDONLY);
  if(file < 0){
    perror("Failed to open file");
    exit(-1);
  }

  // Init mutex cannot fail
  pthread_mutex_init(&mutex, NULL);

  // Spawn a client for festival.
  if( 0 != pthread_create(&speaker, NULL, (void *) spawn_client, NULL) ) {
    perror("Failed to create speeker thread");
    exit(-3);
  }

  // Read in data for reading and check to see if there is a shutup character.
  while(1){
    int amount_sent;
    pthread_mutex_lock(&mutex);
    // If the buffer is filling up wait for it to empty.
    while(buff_to_read >= BUFF_SIZE - 10){
      pthread_mutex_unlock(&mutex);
      usleep(SLEEPTIME);
      continue;
    }
    amount_sent = read(file, buff + buff_to_read, BUFF_SIZE - buff_to_read);
    // If the read fails try again.
    if(amount_sent < 0){
      perror("read failed: ");
      pthread_mutex_unlock(&mutex);
      continue;
    }
    // If we were given the shutup character clear the buffer to stop talking.
    if(memchr(buff + buff_to_read, SYNTH_CLEAR_CHAR, amount_sent) != NULL){
      // printf(" SYNTH CLEAR CHAR recieved.\n");
      buff_read = buff_to_read = 0;
      pthread_mutex_unlock(&mutex);
      continue;
    }
    buff_to_read += amount_sent;

    pthread_mutex_unlock(&mutex);
  }

  // In case of cosmic rays.
  close(file);

  return 0;
}

void spawn_client(void * ignored){
  init_festival();

  while(1){
    pthread_mutex_unlock(&mutex);

    // If there's no data to send wait for some.
    if(buff_to_read == 0){
      pthread_mutex_lock(&mutex);
      usleep(SLEEPTIME);
      continue;
    }

    // If we've caught up then re-set the buffer.
    if(buff_read >= buff_to_read){
      buff_read = buff_to_read = 0;
      pthread_mutex_lock(&mutex);
      continue;
    }

    say_word(tokenize('\n'));
    pthread_mutex_lock(&mutex);
  }

  // In case of cosmic rays.
  close_festival();
}

char *tokenize(char chr){
  int i;
  for(i = buff_read; i <= buff_to_read; i++){
    if(buff[i] == chr){
      buff[i] = '\0';
      return &buff[buff_read];
    }
  }
  buff[buff_to_read + 1] = '\0';
  return &buff[buff_read];
}

int say_word(char *str){
  int retval;
  retval = festivalSpeech(info, str);
  // retval = printf("%s ",str);
  if(retval < 0)
    printf("Speech Error!\n");
  buff_read += strlen(str) + 1;
  return retval;
}

int init_festival() {

  // Configuration information goes here, otherwise defaults used.
  char *server=0;
  int port=-1;

  info = festival_default_info();
  if (server != 0)  info->server_host = server;
  if (port != -1)  info->server_port = port;
  printf("Festival server: %s, on port: %d, with mode: %s\n",
        info->server_host, info->server_port, info->text_mode);
  info = festivalOpen(info);
  if(info == NULL){
    perror("failed to open Festival: ");
    return(-1);
  }

  return 0;
}

int close_festival() {
  return festivalClose(info);
}
```

8.3 middleware/festival.h:

```
/**************************************************************************/
/*                                                                        */
/*                 Centre for Speech Technology Research                  */
/*                      University of Edinburgh, UK                       */
/*                         Copyright (c) 1999                             */
/*                         All Rights Reserved.                           */
/*                                                                        */
/*  Permission is hereby granted, free of charge, to use and distribute   */
/*  this software and its documentation without restriction, including   */
/*  without limitation the rights to use, copy, modify, merge, publish,   */
/*  distribute, sublicense, and/or sell copies of this work, and to       */
/*  permit persons to whom this work is furnished to do so, subject to    */
/*  the following conditions:                                             */
/*   1. The code must retain the above copyright notice, this list of     */
/*      conditions and the following disclaimer.                          */
/*   2. Any modifications must be clearly marked as such.                 */
/*   3. Original authors' names are not deleted.                          */
/*   4. The authors' names are not used to endorse or promote products    */
/*      derived from this software without specific prior written         */
/*      permission.                                                       */
/*                                                                        */
/*  THE UNIVERSITY OF EDINBURGH AND THE CONTRIBUTORS TO THIS WORK         */
/*  DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING       */
/*  ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT    */
/*  SHALL THE UNIVERSITY OF EDINBURGH NOR THE CONTRIBUTORS BE LIABLE      */
/*  FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES     */
/*  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN    */
/*  AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,           */
/*  ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF        */
/*  THIS SOFTWARE.                                                        */
/*                                                                        */
/*************************************************************************/
/*             Author :  Alan W Black (awb@cstr.ed.ac.uk)                 */
/*             Date   :  March 1999                                       */
/*-----------------------------------------------------------------------*/
/*                                                                        */
/* Client end of Festival server API (in C) designed specifically for    */
/* Galaxy Communicator use, though might be of use for other things      */
/*                                                                        */
/*=======================================================================*/


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>



#ifndef _FESTIVAL_CLIENT_H_
#define _FESTIVAL_CLIENT_H_

#define FESTIVAL_DEFAULT_SERVER_HOST "localhost"
#define FESTIVAL_DEFAULT_SERVER_PORT 1314
#define FESTIVAL_DEFAULT_TEXT_MODE "fundamental"

typedef struct FT_Info
{
    int encoding;
    char *server_host;
    int server_port;
    char *text_mode;
    int server_fd;
} FT_Info;

typedef struct FT_Wave
{
    int num_samples;
    int sample_rate;
    short *samples;
} FT_Wave;

void delete_FT_Wave(FT_Wave *wave);
void delete_FT_Info(FT_Info *info);

#define SWAPSHORT(x) ((((unsigned)x) & 0xff) << 8 | \
                      (((unsigned)x) & 0xff00) >> 8)
#define SWAPINT(x) ((((unsigned)x) & 0xff) << 24 | \
                    (((unsigned)x) & 0xff00) << 8 | \
                    (((unsigned)x) & 0xff0000) >> 8 | \
                    (((unsigned)x) & 0xff000000) >> 24)

/* Sun, HP, SGI Mips, M68000 */
#define FAPI_BIG_ENDIAN (((char *)&fapi_endian_loc)[0] == 0)
/* Intel, Alpha, DEC Mips, Vax */
#define FAPI_LITTLE_ENDIAN (((char *)&fapi_endian_loc)[0] != 0)


/*****************************************************************/
/*  Public functions to interface                                */
/*****************************************************************/
/* If called with NULL will attempt to access using defaults */
FT_Info *festivalOpen(FT_Info *info);
FT_Wave *festivalStringToWave(FT_Info *info,char *text);
int festivalClose(FT_Info *info);

#endif






static FT_Info *festival_default_info();
static int festival_socket_open(const char *, int);
FT_Info *festivalOpen(FT_Info *);
int festivalSpeech(FT_Info *,char *);
int festivalCommand(FT_Info *,char *);
int festivalClose(FT_Info *);


FT_Info *info;



static FT_Info *festival_default_info()
{
    FT_Info *info;
    info = (FT_Info *)malloc(1 * sizeof(FT_Info));

    info->server_host = FESTIVAL_DEFAULT_SERVER_HOST;
    info->server_port = FESTIVAL_DEFAULT_SERVER_PORT;
    info->text_mode = FESTIVAL_DEFAULT_TEXT_MODE;

    info->server_fd = -1;
    return info;
}

static int festival_socket_open(const char *host, int port)
{
    /* Return an FD to a remote server */
    struct sockaddr_in serv_addr;
    struct hostent *serverhost;
    int fd;

    fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (fd < 0)
    {
        fprintf(stderr,"festival_client: can't get socket\n");
        return -1;
    }
    memset(&serv_addr, 0, sizeof(serv_addr));
    if ((serv_addr.sin_addr.s_addr = inet_addr(host)) == -1)
    {
        /* its a name rather than an ipnum */
        serverhost = gethostbyname(host);
        if (serverhost == (struct hostent *)0)
        {
            fprintf(stderr,"festival_client: gethostbyname failed\n");
            return -1;
        }
        memmove(&serv_addr.sin_addr,serverhost->h_addr, serverhost->h_length);
    }
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);

    if (connect(fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) != 0)
    {
        fprintf(stderr,"festival_client: connect to server failed\n");
        return -1;
    }

    return fd;
}



/***********************************************************************/
/* Public Functions to this API                                        */
/***********************************************************************/

FT_Info *festivalOpen(FT_Info *info)
{
    /* Open socket to server */

    if (info == 0)
        info = festival_default_info();

    info->server_fd =
        festival_socket_open(info->server_host, info->server_port);
    if (info->server_fd == -1)
        return NULL;

    return info;
}


int festivalSpeech(FT_Info *info,char *text) {
    FT_Wave *wave;
    FILE *fd;
    char ack[4];
    char *p;
    int n;


    if (info == 0)
        return -1;

    if (info->server_fd == -1) {
        fprintf(stderr,"festival_client: server connection unopened\n");
        return -1;
    }

    fd = fdopen(dup(info->server_fd),"wb");


    /* Copy text over to server, escaping any quotes */

    fprintf(fd,"(SayText \"");
    for (p=text; p && (*p != '\0'); p++)
    {
        if ((*p == '"') || (*p == '\\'))
            putc('\\',fd);
        putc(*p,fd);
    }
    fprintf(fd,"\")\n");
    fclose(fd);


    return 0;
}



int festivalCommand(FT_Info *info,char *text) {

    FT_Wave *wave;
    FILE *fd;
    char ack[4];
    char *p;
    int n;


    if (info == 0)
        return -1;

    if (info->server_fd == -1) {
        fprintf(stderr,"festival_client: server connection unopened\n");
        return -1;
    }

    fd = fdopen(dup(info->server_fd),"wb");


    /* Copy text over to server, escaping any quotes */

    fprintf(fd,"(");
    for (p=text; p && (*p != '\0'); p++)
    {
        if ((*p == '"') || (*p == '\\'))
            putc('\\',fd);
        putc(*p,fd);
    }
    fprintf(fd,")\n");
    fclose(fd);


    return 0;
}



int festivalClose(FT_Info *info)
{
    if (info == 0)
        return 0;

    if (info->server_fd != -1)
        close(info->server_fd);

    return 0;
}
```

8.4 speakup/speakup.patch

```
Index: Config.in
===================================================================
RCS file: /usr/src/CVS/speakup/Config.in,v
retrieving revision 1.4
diff -u -r1.4 Config.in
--- Config.in   24 Jan 2001 16:56:24 -0000      1.4
+++ Config.in   23 Apr 2003 01:53:05 -0000
@@ -12,6 +12,7 @@
        bool "DoubleTalk LT or LiteTalk, ltlk" CONFIG_SPEAKUP_LTLK
        bool "Speak Out, spkout" CONFIG_SPEAKUP_SPKOUT
        bool "Transport, txprt" CONFIG_SPEAKUP_TXPRT
+       bool "Software synth driver, usrdev" CONFIG_SPEAKUP_USRDEV
        comment 'Enter the four to six character synth string from above or
none.'
        string "Default synthesizer for Speakup" CONFIG_SPEAKUP_DEFAULT "none"
        bool "Use Speakup keymap by default" CONFIG_SPEAKUP_KEYMAP
Index: Makefile
===================================================================
RCS file: /usr/src/CVS/speakup/Makefile,v
retrieving revision 1.10
diff -u -r1.10 Makefile
--- Makefile    5 Jul 2002 17:51:58 -0000       1.10
+++ Makefile    23 Apr 2003 01:53:05 -0000
@@ -12,10 +12,13 @@
 DRIVERS := speakup_dtlk.c speakup_ltlk.c speakup_acntpc.c \
        speakup_acntsa.c speakup_spkout.c speakup_txprt.c \
        speakup_bns.c speakup_audptr.c speakup_decext.c \
-       speakup_dectlk.c speakup_apolo.c
+       speakup_dectlk.c speakup_apolo.c speakup_usrdev.c
 O_TARGET       := spk.o
 O_OBJS         :=
 export-objs    :=
+ifeq ($(CONFIG_SPEAKUP_USRDEV),y)
+O_OBJS      += speakup_usrdev.o
+endif

 ifeq ($(CONFIG_SPEAKUP),y)
 O_OBJS         += speakup.o speakup_drvcommon.o
Index: speakup.c
===================================================================
RCS file: /usr/src/CVS/speakup/speakup.c,v
retrieving revision 1.77
diff -u -r1.77 speakup.c
--- speakup.c   14 Dec 2002 01:15:48 -0000      1.77
+++ speakup.c   23 Apr 2003 01:53:05 -0000
@@ -83,6 +80,9 @@

 /* These are ours from synth drivers. */
 extern void proc_speakup_synth_init (void);    // init /proc synth-specific
subtree
+#ifdef CONFIG_SPEAKUP_USRDEV
+extern struct spk_synth synth_usrdev;
+#endif
 #ifdef CONFIG_SPEAKUP_ACNTPC
 extern struct spk_synth synth_acntpc;
 #endif
@@ -138,6 +140,9 @@
 unsigned short mark_y = 0;
 static char synth_name[10] = CONFIG_SPEAKUP_DEFAULT;
 static struct spk_synth *synths[] = {
+#ifdef CONFIG_SPEAKUP_USRDEV
+       &synth_usrdev,
+#endif
 #ifdef CONFIG_SPEAKUP_ACNTPC
        &synth_acntpc,
 #endif
```

8.5 speakup/speakup_usrdev_shared.h:

```
/* This character is sent to userland to signal the end of something that
 * should be spoken.
 * Currently a newline as it seems as reasonable as anything. */
#define START_TALKING_CHAR    '\n'
#define START_TALKING_STRING "\n"
#define SYNTH_CLEAR_CHAR      '\0'
#define SYNTH_CLEAR_STRING    "\0"
#define CAPS_START_STRING     " Capitol "
#define CAPS_STOP_STRING      " Lower case "
#define PITCH_STRING          " *squeak* "
```

8.6 speakup/speakup_usrdev.h:

```
/* speakup_usrdev.h
 *
 * Assorted defines and headders for the Userland Character Device driver
 * for speakup.
 *
 * !!! Add a more verbose headder here
 */

/***** TWEAKABLE SETTINGS *************************************************/

/* The printk level to send internal debugging messages as
 * Set this to something low to display the messages,
 * or to KERN_DEBUG to more or less desable them
 */
```

```
#define DEBUG_MESSAGE_PRIORITY KERN_DEBUG

#include "speakup_usrdev_shared.h"

/***** CHARACTER DEVICE OBJECT *******************************************/
/* Constants:
 *   CHAR_DEV_NAME
 *   CHAR_REQUESTED_MAJOR_NUMBER
 *
 * Variables:
 *   char_dev_created
 *   char_fops
 *   char_major
 *   char_is_open
 *   char_mutex
 *   char_wait_queue
 *
 * Functions
 *   char_create_dev()
 *   char_destroy_dev()
 *   char_open()
 *   char_poll()
 *   char_read()
 *   char_close()
 */

/* !!! Verify where this actually appears */
#define CHAR_DEV_NAME "speakup_usrdev"

/* Internal init / deinit functions */
int char_create_dev(void);
int char_destroy_dev(void);

/* Flag signeling the result of char_create_dev */
int char_dev_created = 0;

/* Userland interface functions */
int char_open(struct inode *, struct file *);
unsigned int char_poll(struct file *, struct poll_table_struct *);
ssize_t char_read(struct file *, char *, size_t, loff_t *);
int char_close(struct inode *, struct file *);

/* The File Operations structure that tells Linux of the above functions */
struct file_operations char_fops = {
  open:char_open,
  poll:char_poll,
  read:char_read,
  release:char_close,
};

/* The major number for the device.  char_major will contain the actual
 * major number assigned after char_init() is called.
 *
 * Available experemental use majors are 60-63, 120-127, 240-254
 * Major of 0 will chose the first free major number
 */
#define CHAR_REQUESTED_MAJOR_NUMBER 252
unsigned int char_major;

/* Boolean state of wether the device is currently open.
 * Note: currently only one program may have the device open at a time
 */
int char_is_open;

/* A generic semaphor for character device stuff access control.
```

```
 * This hopefully will prove unnecessairy
* !!! This comment needs correcting later
 */
struct semaphore char_mutex;

/* The wait queue used to put reading processes to sleep. */
DECLARE_WAIT_QUEUE_HEAD(char_wait_queue);


/* Note: The assorted char_* variables above were in a single structure,
 * but that ran into odd problems so they're all on their own now.
 */



/***** SPEAKUP INTERFACE ************************************************
 * Functions:
 *   spk_probe()
 *   spk_do_catch_up()
 *   spk_write()
 *   spk_is_alive()
 *
 * Data structures:
 *  A whole whopping lot of them.  :]
 */


/*** Function headders
 * In this driver I've prepended spk_ to all of these functions to make them
 * more distinct from the userland-interface function names.  The actual
 * naming of these functions is entierly up to the driver's programmer,
 * but probe, do_catch_up, write, and is_alive are the standard names.
 * See the .c file for descriptions of what each function does.
 */

int spk_probe(void);
void spk_do_catch_up(unsigned long l);
void spk_write(char ch);
int spk_is_alive(void);


/*** Data Structures
 * These are what Speakup uses to connect to a module.  Everything that
 * is defined above (functions & constants) that the main part of
 * Speakup needs to use is embedded into one or more of these data
 * structures.
 *
 * To be more exact, all but the last of these data structures is placed
 * into the final one.  The main part of Speakup interacts with this
 * entire module through that last structure.  Neat, huh?
 *
 * As a note, all but the last structure should be static; !!! explain
 * why
 */


/* The initialization string.  !!! How does this work?  Is it what we
 * get back, what we send, both?
 */
static const char init_string[] = "Speakup userland synth output started.";


/* The reinitialization string.  !!! How does this work?
 */
static const char reinit_string[] =
"Speakup userland synth output restarted.";


/* Control characters and variables.
 * Unless set as NO_USER, they can be set in /proc/speakup
 *
 * The format for the entries, taken from speakup.h:
 *
 ***    struct spk_variable {
 *
 ***    char *id;
 *
 * The name of the command.  This can be anything, but the following
 * variables are always included and should not be specified:
 *
 * jiffy_delta, delay_time, queued_bytes, sent_bytes, trigger_time,
* full_time, version
 *
 * Additionally, the first 4 items must be:
 * flush, pitch, caps_start, caps_stop
 *
 ***    char *param;
 *
 * The value of the parameter to the command.  This will be inserted
 * into the command string as described below.
 *
 *
 ***    char *build;
 *
 * The command format string.  A string, describing how to construct
 * the string sent to the synth, with the character '_' to be filled
 * in with the parameter.
 * eg. "foo _" will convert to "foo bar", if param is "bar"
 *
 *
 ***    int flags;
 *
 * Flags to control how this command is sent to the synth.  They should be
* Or'ed together if more than one is used.
 *
 * Available flags are:
 * HARD_DIRECT   Send this variable direct to the synth
 * SOFT_DIRECT   Send this variable to speakup
 * ALLOW_BLANK   Alias-only flag to allow a blank parameter
 * BUILDER       Variable must be built into the reset string
 * NO_USER       Variable is not allowed to be set by the user
 * MULTI_SET     ASCII string of chars, each must be one of the valid set
 * USE_RANGE     All values in ASCII range specified below are accepted
 * NUMERIC       ASCII-represented numerical value currently requires
 *                 USE_RANGE to be set
 *
 *
 ***    char *valid;
 *
 * If the flag USE_RANGE is given, valid is a range described by
 * "lowval,highval".  For example, "0,20" is the range 0-20.  If
 * USE_RANGE is not given, valid is a NULL terminated set of valid
 * values for param. eg. "aeiou".  Wildcard "*" matches anything.
 *
 *
 *** };
*
 * Final note: this struct must be ended with an entry of END_VARS
 */
static struct spk_variable vars[] = {
  /* Pretty well all of this needs to be set by the user to match their
   * synth.  Reasonable basic defaults are set here. */
  {"flush", SYNTH_CLEAR_STRING, "_", (BUILDER | MULTI_SET), "*"},
  {"pitch", "", "_", (MULTI_SET | ALLOW_BLANK), "*"},
  {"caps_start", "", "_", (MULTI_SET | ALLOW_BLANK), "*"},
  {"caps_stop", "", "_", (MULTI_SET | ALLOW_BLANK), "*"},

  /* This structure MUST be terminated with this !!! (does it?) */
  END_VARS
};


/* The config strings.  !!! What do each of these do?
 *
 * The first 4 of these should be the control strings for:
 * flush, pitch, caps start, caps stop
 *
 * The rest can be anything.  !!! Can they?
 */
static char *config[] = {
  START_TALKING_STRING,  // Flush string
  PITCH_STRING,
  CAPS_START_STRING,
  CAPS_STOP_STRING,
  "[free_to_configure1]",
  "[free_to_configure2]",
  "[free_to_configure3]",
  "[free_to_configure4]",
};


/*** Everything is bound together in this structure: ***/
/* This is the structure that is referred to by the main body of
 * Speakup.  Everything that needs to be accessed from a module is in
 * this or one of the sub-structures. !!! Waht do these do?
 */
struct spk_synth synth_usrdev = {
  /* Short name of this module.  !!! max 6 chars? */
  "usrdev",

  /* Version string for the module.  No required format, but the
   * convention seems to be Version-#.##
   */
  "Version-0.00 (work in progress)",

  /* Full name of this module */
  "Userland character-device output",

  /* The initialization and reinitilization strings */
  init_string,
  reinit_string,

  /* delay_time, trigger_time, jiffy_delta, full_time */
  500, 50, 5, 1000,

  /* !!! The variables */
  vars,

  /* !!! The config */
  config,

  /* !!! config_map */
  0,

  /*** The base functions ***/
  /* All of these provide references to the functions for Speakup to
   * use.  The downside of this is that this can make your C compiler
   * not notice that some of the function prototypes don't match
   * (!!! is that true?)
   */

  /* !!! The probe function */
  /* int probe(void) */
  spk_probe,

  /* !!! ? */
  /* void catch_up(unsigned long) */
  spk_do_catch_up,

  /* Send a single character to the synth.  !!! ? */
  /* void write(char c) */
  spk_write,

  /* !!! Check that the synth is still alive */
  /* int is_alive(void) */
  spk_is_alive
};


/***** RANDOM OTHER USEFUL STUFF *******************************************/

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

/* There used to be a character buffer here, but it was removed for favour
 * of Speakup's own buffer.
 */
```

8.7 speakup/speakup_usrdev.c:

```
// !!! Copied from speakup_drvcommon, probably a bad idea but needed elsewhere.
#define synthBufferSize 8192
/** speakup_usrdev.c - Userland character-device driver for Speakup
 *
 *   author: Nick Pinney <lutris@wpi.edu>
 *
 *   This driver provides a userland character device output for Speakup.
 *
 *   Notable features of the device are that it can only be opened by one
 *   process at a time, all data from speakup while it is not opened are
 *   discarded, ...
 *
 *   Comments starting with a !!! note things that I don't yet fully
 *   understand or that need to be more fully commented.
 **/

#include <linux/config.h>    // Everything [un]defined by `make config`
#ifndef SPEAKUP_USRDEV_C
#define SPEAKUP_USRDEV_C
#define KERNEL
```

```
/***** INCLUDES ******************************************************/

/* Kernel stuff */
#include <linux/version.h>   // For any version-specific #ifdef's
#include <linux/kernel.h>    // Lots of useful stuff (printk log levels)
#include <linux/types.h>     // assorted types used by the kernel.
#include <linux/errno.h>     // for -EBUSY and other error states
#include <asm/uaccess.h>     // copy_to_user()
#include <linux/poll.h>      // For constants and poll_wait() in char_poll().

/* Speakup's headders */
#include <linux/tty.h>       // required by speakup.h
#include <linux/speakup.h>

/* stuff for the character device */
#include <linux/fs.h>        // Filesystem access !!!
#include <asm/semaphore.h>   // Semaphore protection for the buffer

/*** Headders and the structures                                      ***/

#include "speakup_usrdev.h"


/***** CHARACTER BUFFER **********************************************/
/* This has been removed for favour of Speakup's own buffer          */
/***** END CHARACTER BUFFER ******************************************/


/***** CHARACTER DEVICE **********************************************/
/* Functions:
 *   char_open()
 *   char_poll()
 *   char_read()
 *   char_close()
 *   char_create_dev()
 *   char_destroy_dev()
 *
 * Constants:
 *   CHAR_DEV_NAME
 *   CHAR_REQUESTED_MAJOR_NUMBER
 *
 * Variables:
 *   char_fops
 *   char_major
 *   char_is_open
 *   char_mutex
*   char_wait_queue
 */

/*** Initialization and Destruction                                   ***/

/* Create the character device
 *
 * Returns: 0 on success, an error code on failure
 */
int char_create_dev(void)
{
  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_create_init().\n");

  //// Init stuff.  Done before registering the device
  // to avoid race conditions
  sema_init(&char_mutex, 1);
  char_is_open = FALSE;
  // char_wait_queue is already initialised
  //// Register the device
  char_major
    = register_chrdev(
                        CHAR_REQUESTED_MAJOR_NUMBER, CHAR_DEV_NAME, &char_fops);

  //// If the registration failed
  if(char_major < 0) {
    char_dev_created = FALSE;
    printk(KERN_WARNING
           "Speakup usrdev: Could not register char device major %d, error
%d\n"
,
           CHAR_REQUESTED_MAJOR_NUMBER, char_major);
    return char_major;
  }

  //// If the registration succeeded
  char_dev_created = TRUE;
  printk("Speakup usrdev: Registered as char device major #%d\n",
        CHAR_REQUESTED_MAJOR_NUMBER > 0 ? char_major:
         CHAR_REQUESTED_MAJOR_NUMBER);
  printk("Speakup usrdev: Minor number is 0\n");
  return 0;
}


/* Destroy the character device
 *
 * This would be called when a driver is made inactive, but I suspect that in
 * practice it will only be used if Speakup ever becomes modular.
 *
 * Returns: 0 on success, an error code on failure
 */
int char_destroy_dev(void)
{
  char_dev_created = FALSE;
  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_destroy_dev()\n");
  return unregister_chrdev(char_major, CHAR_DEV_NAME);
}

/*** Userland interface functions                                           ***/


/* Userland open the character device
 *
 * Returns: 0 on success, an error code on failure
 */
int char_open (struct inode *inode, struct file *filp)
{
  int retval;
  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_open()\n");

  // Protect against concurrent access
  if( down_interruptible(&char_mutex) ){
    printk(DEBUG_MESSAGE_PRIORITY
           "usrdev: char_open() failed to grab semaphore, try again later.\n");
    return -ERESTARTSYS;
  }

  //// Error conditions
  retval = 0;

  // Wrong minor number
  if( MINOR(inode->i_rdev) != 0 ) {
    retval = -ENODEV;
    printk(DEBUG_MESSAGE_PRIORITY
           "usrdev: char_open() called with incorrect minor number %d.\n",
           inode->i_rdev);
  }

  // Device already opened
  if( char_is_open ){
    retval = -EBUSY;
    printk(DEBUG_MESSAGE_PRIORITY
           "usrdev: char_open() called while device already in use.\n");
  }

  // Only read-only access is allowed
  if( (filp->f_flags && O_ACCMODE) != O_RDONLY ){
    retval = -EPERM;
    printk(DEBUG_MESSAGE_PRIORITY
           "usrdev: char_open() device was opened in other than read-only
mode.\
n");
  }

  //// Did any of the above errors occur?
  if( retval ) {
    up(&char_mutex);
    printk(DEBUG_MESSAGE_PRIORITY "usrdev:  failed with error %d\n",retval);
    return retval;
  }

  // Declare the buffer empty.
  synth_sent_bytes = synth_queued_bytes = 0;

  //// No errors (mutex is still in use)
  // If this were a kernel module it would also have its use count increased
  char_is_open = TRUE;

  // Set up the file operations structure
  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_open() setting fops.\n");
  filp->f_op = &char_fops;

  // Release the mutex for further access
  up(&char_mutex);

  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_open() finished fine.\n");
  return 0;
}


/* Userland read from the character device
 *
 * Sleeps the process (woken in catch_up()) if the read is blocking.
 * Copies data from Speakup's buffer into userspace
 *
 * Returns: number of bytes sent on success, a negative error code on failure
 */
ssize_t char_read (struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
  int amount_to_send;

//  printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_read() called.\n");

  // Wait for data or return -EAGAIN if the read is non-blocking.
  if((filp->f_flags && O_NONBLOCK) && ((synth_queued_bytes - synth_sent_bytes)
=
= 0))
```

```
      return -EAGAIN;
    else
      interruptible_sleep_on(&char_wait_queue);

//    printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_read() woke up.\n");
//    printk(DEBUG_MESSAGE_PRIORITY "usrdev: char_read() %d bytes
queued.\n",syn
th_queued_bytes);

  //!!! Mutex protect from here to the -2- exit points
    down_interruptible(&char_mutex);

    // Device may have closed while waiting on the semaphore.
    if (!char_is_open){
      up(&char_mutex);
      return 0;
    }

  // Is there any data to send?
    amount_to_send = (count > synth_queued_bytes - synth_sent_bytes) ?
      synth_queued_bytes - synth_sent_bytes: count;

  // Copy the data to userland
  if( copy_to_user(buf, synth_buffer+synth_sent_bytes, amount_to_send) ) {
    printk(DEBUG_MESSAGE_PRIORITY
      "usrdev: char_read() copy_to_user() failed.\n");
    up(&char_mutex);
    return -EFAULT;
  }

  // Update the buffer counters and reset if the whole buffer has been sent.
  synth_sent_bytes += amount_to_send;
  if (synth_sent_bytes == synth_queued_bytes) {
    synth_sent_bytes = synth_queued_bytes = 0;
  }

  // Theoretically this should be done; doens't seme necessary, though.
  //    *f_pos += amount_to_send;

/*   printk(DEBUG_MESSAGE_PRIORITY */
/*       "usrdev: char_read() sent %d bytes to userland.\n", amount_to_send);
*/

  up(&char_mutex);

  return amount_to_send;
}

/* Userland status poll on the device
 *
 * Returns: normal data available !!! Implement non-blocking reads again
 */
unsigned int char_poll (struct file *filp, struct poll_table_struct *wait)
{
  printk(KERN_DEBUG "usrdev: char_poll_status() called.\n");
  // POLLRDNORM should only be returned if data is available.
  // I think that it can block until data is available but doesn't have to.
  //!!! How does poll_wait work?
  poll_wait(filp, &char_wait_queue, wait);

  if((synth_queued_bytes - synth_sent_bytes) > 0)
    return (POLLIN | POLLRDNORM);
  else
    return 0;
}
```

```
/* Userland closes the character device
 *
 * Returns: 0, it cannot fail
 */
int char_close (struct inode *i, struct file *f)
{
  printk(DEBUG_MESSAGE_PRIORITY "usrdev: device_closed()\n");

  down(&char_mutex);
  char_is_open = FALSE;
  up(&char_mutex);

  printk(DEBUG_MESSAGE_PRIORITY
         "usrdev: device_closed() mutex worked fine.\n");
  return 0;
}


/***** END CHARACTER DEVICE ***************************************************/


/***** SPEAKUP FUNCTIONS ******************************************************/
/* Functions:
 *   spk_probe()
 *   spk_do_catch_up()
 *   spk_write()
 *   spk_is_alive()
 */


/* Note: All of the structures assosiated with a speakup driver are in the
 *       .h file.
 */


/* Probe for the presence of a device
 *
 * This function is used to detect the presence of the synth device.
 *
 * Returns: 0 on success
 *          -ENODEV or another sensible error.
 *
 * For this driver probing for the device creates the device file and fails
* if the device couldn't be created
 *
 * !!! Does this get called more than once?  If so, this needs to be modified
 */
int spk_probe(void)
{
  printk(DEBUG_MESSAGE_PRIORITY
         "usrdev: probe() creating Speakup userland device\n");
  return char_create_dev();
}


/* catch_up() is called when Speakup's bufer needs flushing, normally due
 * to it becomming full.  It is also called in our spk_write when we parse
 * something that causes a buffer flush (start talking).
 *
 * Write-up note: This normally is only called when the buffe is overfilling
 *
 * The unsigned long argument to catch_up() appears to never be used inside
 * the function. Perhaps some synths need it.
 */
void spk_do_catch_up(unsigned long l)
{
  //printk(DEBUG_MESSAGE_PRIORITY "usrdev: spk_do_catch_up(%ld) with %d
queued\n
", l, synth_queued_bytes);

  // If no-one is listening, then just clear the buffer
  if(!char_is_open){
    synth_sent_bytes = synth_queued_bytes = 0;
  }

  // Reactiveate or deactiveate some timer? I think this is related to the
  // synth_delay function, as it appears to create timers that trigger
  // events when they go off
  synth_timer_active = 0;

  // Wake up the userland process if it is waiting for data
  if (waitqueue_active(&char_wait_queue))
    wake_up_interruptible(&char_wait_queue);
}

/* Accepts characters to be buffered
 *
 *   Characters that are to be sent to the buffer are filtered through this
 * function.  Dueties here are mainly checking that it is a valid character
 * to send and checking to see whether the buffer should be emptied
* immediately if possible.
 *
 * !!! Does this even get called if the synth is muted?  Probably not.
 *
 * Input:  Character to be sent to the buffer.
 */
void spk_write(char ch)
{
  //// Should we buffer the character at all?
  if (!char_is_open) return;         // Don't buffer if nothing is listening
  if (ch < 0x00){
    return;             // Ignore unprintable chars
  }

  // See if we need to flush the buffer.
  if(ch == SYNTH_CLEAR_CHAR) {
    printk(DEBUG_MESSAGE_PRIORITY "SYNTH_CLEAR_CHAR recieved.");
    down_interruptible(&char_mutex);
    // Check that the device wasn't closed while waiting on the semaphore.
    if (char_is_open)
      synth_timer_active = synth_queued_bytes = synth_sent_bytes = 0;
    else return;
    up(&char_mutex);
  }

  // If not then make sure that we don't overflow the buffer.
  if(synth_queued_bytes >= synthBufferSize){
    printk(KERN_WARNING "Speakup: userdev: Buffer limit reached, loosing
data.")
;
    return;
  }

  //// Buffer the character
  down_interruptible(&char_mutex);
  // Check that the device wasn't closed while waiting on the semaphore.
  if (char_is_open)
    synth_buffer_add(ch);
  up(&char_mutex);

  ///// Send data on start-talking, line feed, or carriage return characters
  if ( (ch == START_TALKING_CHAR) || (ch == '\x0A') || (ch == '\x0D') ||
       (ch ==  SYNTH_CLEAR_CHAR)) {
    //printk(DEBUG_MESSAGE_PRIORITY "usrdev: spk_write() calling
spk_do_catch_up
()");
    spk_do_catch_up(0);
  }

  return;

  ///////// for the write-up //////////

  /* This seems to always be true.  May wish to comment out later. */
  // if (synth_buffering) return;

  //  if (synth_timer_active == 0) synth_delay( synth_trigger_time );

  /* !!! It's a good idea to check how your buffer is doing here and
   * call synth_delay() if you're getting full.
   * --Owen: This is done by synth_buffer_add.
   *
   * synth_delay(int ms) is defined in speakup_drvcommon.c: I don't
   * entirely know how it works, but it appears to delay for ms
   * miliseconds.
   */
}


/* Check the functional status of the synth
 *
* This device doens't really ever not work once its set up.
 * This is all this function does?
 */
int spk_is_alive(void)
{
  printk(KERN_DEBUG "usrdev: is_alive()\n");

  /* This driver is always alive */
  return char_dev_created;
}


//#ifndef SPEAKUP_USRDEV_C
#endif
```

9 Appendix B: Detailed explanation of our code

As explained in the Design Overview earlier in this document, our code is broken into several major sections: Speakup, our Speakup driver, our Middleware, and Festival. Much of our project was spent designing and implementing the Speakup driver and the Middleware program, so a full explanation will be given here. Speakup and Festival were pre-existing pieces of software written by different groups and will not be explained in detail here.

9.1 Speakup / Driver interface

Speakup provides a fairly simple but powerful interface for Speakup drivers to use. All of the structures and functions used in this interface are accessed through pointer references and as such need not have any specific name, however we tried to follow the naming scheme used in most modules to make our code more legible.

Unfortunately, much of Speakup's driver interface is undocumented. Although Speakup's creator, Kirk Reiser, was very helpful in answering our questions on the interface, there are still aspects of the interface that we do not fully understand as they did not appear to be important to the design of our module. These will be explained in greater detail below.

9.1.1 Associated files

Three files in the main Speakup distribution need to be modified in order to add a module to Speakup. First is the Config.in file; it controls the options available when configuring Linux and needs to have an extra bool item added. The variable name of that item will be referred to in both the Makefile and in speakup.c for conditional includes. The convention for naming follows `CONFIG_SPEAKUP_[short-name]`, so ours was named `CONFIG_SPEAKUP_USRDEV` because we created a userland device.

The Makefile needs to have the new c file added to the DRIVERS list and a conditional include needs to be added further down in order to include the new object file for linkage. This will cause the new module code to be compiled and to be linked into Speakup's kernel code.

speakup.c needs to have a couple of lines added to it. Both of these lines should be protected with ifdef's to ensure that they are only included if that particular driver was compiled into Speakup. Both of these additions will be inserted in with lists of the available drivers, and for legibility should be inserted in alphabetical order.

First, the driver's interface structure, explained below, needs to be declared as an extern variable to make it available to Speakup:

```
#ifdef CONFIG_SPEAKUP_USRDEV
extern struct spk_synth synth_usrdev;
#endif
```

Next, the structure needs to be added to the list of available drivers:

```
static struct spk_synth *synths[] = {
. . .
#ifdef CONFIG_SPEAKUP_USRDEV
extern struct spk_synth synth_usrdev;
#endif
. . .
}
```

9.1.2 Speakup's character buffer

Speakup provides a buffer for characters being sent out to the speech synthesizer. In addition to simple buffering, the length of the buffer is monitored; to prevent overflow the `do_catch_up()` function, explained below, is called when the buffer is nearing full. The buffer has a low-level interface, providing a function for adding a character but leaving reading from the buffer to the individual drivers. As with all of Speakup, this buffer is not protected against concurrent access, so we added a mutex around all of our buffering code for protection.

Internally, there are three elements to the buffer. `synth_buffer` is a pointer to the beginning of the buffer. `synth_sent_bytes` is the offset to the beginning of the unsent buffered data. `synth_queued_bytes` is the offset to the end of the buffered data.

Reading from the contents of the buffer involves reading the buffered data between `synth_buffer + synth_sent_bytes` and `synth_buffer + synth_queued_bytes` and then either updating `synth_sent_bytes` if the buffer was only partially read or setting both `synth_sent_bytes` and `synth_queued_bytes` to 0 if the buffer was emptied.

Clearing the buffer, as is done by our driver when the character device is not yet opened, is simply done by setting `synth_sent_bytes = synth_queued_bytes = 0`.

Adding a character is simple; the function `synth_buffer_add(character)` adds a character to the buffer and adjusts `synth_queued_bytes` automatically. Additionally, it checks to see if the buffer is nearing full and calls the driver's `do_catch_up()` if needed.

9.1.3 Struct synth_name

Speakup keeps a reference to a single data structure in every module and accesses the entire functionality of that driver through that structure. The format for the structure is as follows:

```
struct spk_synth synth_usrdev = {
```

The structure is specified in the driver and is accessed in the main body of Speakup's code as an external variable, as defined above in the Associated files section.

This should be named as per the convention "synth_name" where "name" is the name of the module, and should be limited to no more than 6 characters.

```
/* driver short name, version, and long name */
"usrdev",
"Version-0.00 (work in progress)",
"Userland character-device output",
```

The first three items in the structure are stings describing the module. The first is the short name for the module; it is used by Speakup to differentiate and identify modules when selecting a module at boot time and (once this is implemented) when the user sets a different module at run time. The second is the version number for the module and is only used for reference; almost all modules follow the format "Version-#.##" for this string. The third is a detailed name of the module.

```
/* The initialization and reinitilization strings */
init_string,
reinit_string,
```

Our driver handled all synthesizer configuration in our middleware program, so we did not use these two variables and gave them dummy values. We did not research at what times these strings are sent to the synthesizer device.

```
/* Delay times */
500, 50, 5, 1000,
```

These four numbers are delay times for sending data to the synthesizer. It is important that a kernel module never use the processor continually for an extended period of time as that would prevent userland programs from running and would effectively lock up the computer. The delay times are used to prevent this in modules that communicate to an external speech synthesizer, but are essentially unused in our module as we send data in large chunks to the user. As such, we left in the default values form another synthesizer.

```
/* Control variables, possibly user adjustable */
vars,
  /* Command sequence strings */
  config,
```

These two items are explained in detail below. The first is a structure defining several variables for the module, and may individually be set as user adjustable or immutable. The second is a list of control strings that are sent to the synthesizer to modify its output.

```
  /* config_map */
  0,
```

We never found out what this does. All of the other Speakup drivers have this set to 0, so we left it like that in our code.

```
  /*** The four base functions ***/

  /* Device probe */
  spk_probe,

  /* Catch up to empty the buffer */
  spk_do_catch_up,

  /* Send a single character to the synth. */
  spk_write,

  /* Check synth status */
  spk_is_alive
};
```

Finally, the four functions common to all Speakup modules are referenced at the end of the structure. These are each explained in detail below.

9.2 Speakup Module

9.2.1 *Struct vars*

```
Sttic struct spk_variable vars[] {
  {"flush", SYNTH_CLEAR_STRING, "_", (BUILDER | MULTI_SET), "*"},
. . .
```

```
  END_VARS
};
```

This structure contains several control strings for the synthesizer. The format of each item is fairly simple, containing in order:

- The name of the string
- Default value
- Control string
- Flags
- Acceptable characters for a user-set value

The control string should have an underscore where the value is to be inserted. That value may be the default or may be set by the user via Speakup's /proc/speakup interface. The flags are defined in both of the speakup/diff directories in the file ^usr^src^linux^include^linux^speakup.h.copy. Comments in the file give descriptions of what each flag does and those descriptions are also available in our speakup_usrdev.h file. As our implementation of this section is somewhat minimal a detailed description of which flags we chose to set is not specified here.

The first four of these are common to all drivers and should be (possibly required in this order): "flush", "pitch", "caps_start", "caps_stop". The latter four are driver-specific and may be omitted; we never found a way to trigger them and as such omitted them. All of these are sent to the driver one character at a time via the `write()` funtion call. This section must be terminated with the macro `END_VARS`.

"flush" is sent whenever the buffer needs to be emptied. We set this to be a single newline character in our code for several reasons. Most drivers have their `write()` function start transmitting the buffer on a newline anyway so this seemed to be a logical choice for our buffer flush string.

We did not implement the "pitch," "caps_start," or "caps_stop" strings as we were uncertain of how best to implement them, and decided to leave them as features to implement after we have received more feedback from our users. Additionally, we are still learning some of the details of how to use Festival's speech mark-up and at the time of this writing had not experimented with altering the inflection of its output. Additionally, a simpler way of setting these is presented in the next section.

One notable item that will be added here in future versions is a user-configurable value for speech rate. As this essentially controls the rate at which the user can read from the screen, many users of speech synthesizers have this speech synthesis value set as high as they can understand. At the time of writing of this document we were still experimenting with setting this in Festival and had not yet implemented it as a feature of our code.

9.2.2 *Struct config*

```
static char *config[] {
. . .
```

The "config" structure is an array of eight strings that are used to control the output of the speech synthesizer. The first four have pre-defined meanings and must be specified in the correct order. The last four are specific to a given module; we were unable to determine the use of these strings. As these values are referenced in several places in our code they are defined as constants at the beginning of our speakup_usrdev.h file.

The use of this structure appears to overlap with the above struct vars. We never determined the exact difference between these structures as setting static values in this structure worked in a reasonable fashion.

The first string is the "flush" string. For the purpose of our driver this string was used as a signal that the buffer should be flushed to the middleware program, and was set to be the newline character for reasons described in the above Struct vars section.

Next is the "pitch" string. We never determined the exact use of this string; it appeared to be inserted into the buffered text to indicate that a word should be emphasized. This is something that we will implement in a future version of the code once we have more experience with Festival.

The following two strings are the "caps start" and "caps stop" strings. These are sent to notify the speech synthesizer of how to emphasize single characters when speaking a string of text one character at a time. Eventually these will be set in our code to control characters that can be interpreted for different speech synthesizers; for the moment they are simply set to the pronounceable statements "Capitol" and "Lower case".

9.2.3 Module functions

Speakup drivers must implement the following four functions. They each perform basic operations for communicating with the speech synthesizer and are the mechanism through which Speakup "talks" to the user.

9.2.4 Function probe

The "probe" function detects and initializes the synthesizer. Most drivers open a connection to the serial port here; ours registers the character device that we use to communicate with the middleware program. This function must return 0 on success or an error value, such as –ENODEV, on all fatal error conditions.

Our current code only works if Speakup is part of the main kernel and is not compiled as a module. This is fine at the moment as Speakup cannot yet be compiled as a module, but future versions of our code should also provide a mechanism for removing the character device created here. Additionally, a devfs interface to our character device should be added to future versions.

9.2.5 Function do_catch_up

The "do_catch_up" function is called when the character buffer is starting to become full. For most drivers this function sends characters to the synthesizer one by one, requesting a delay between each character to prevent Speakup from using too much CPU time.

Our module handles this function a bit differently; we have no hardware that must have the data sent to it one character at a time. We instead have a normal program waiting to read data from our character device, as explained in detail below. As such, our implementation of this function simply wakes the process that is waiting to read; the actual data passing is handled by the `char_read()` function explained below.

9.2.6 Function write

The "write" function takes in single characters to be interpreted and usually buffered. Our code checks to see if the character device is open and simply drops the new character if no-one is listening. It checks to see if the buffer needs to be flushed due to it becoming near full or due to a flush character having been parsed. Filtering and translation of non-alphanumeric charactes (such as !@#$%) is handled by Speakup and need not be considered in this function, however most drivers including ours filters out definitely non-pronouncable characters. After these checks are made the character is buffered by a call to `synth_buffer_add()`.

9.2.7 Function is_alive

We are not entirely certain of the purpose of this function; iot either checks the functional status of the driver or of the driver's device. As our driver is effectively always on once loaded, it only returns the status of whether or not the character device was created successfully.

9.3 Middleware Interface

We used a character device to send data from our Speakup driver to our middleware program in userland. The interface for defining a new character device as a part of the Linux kernel is well defined and documented in several books.

For simplicity, our current version only offers a normal character device. Our code uses an entire major device for itself. As there are only 256 total major numbers in the Linux kernel versions 2.4 and below, most of which have pre-defined uses, this is a rather unreasonable requirement for a device that only provides one device file. Using one of the extra minor numbers of the miscellaneous device major numbers is considerably more complex and would require our code to be approved as a part of the canonical Linux kernel, and as such is not a reasonable short-term solution. A better solution will be to add devfs support to the driver, as is our plan for a future version.

Our initial design allowed for both blocking and non-blocking access, however that was limited to only blocking access in later revisions of our code, including the current release. Through the design of our device we were uncertain as to the utility of allowing non-blocking access, and as allowing for both made the code more complex we removed that part. Our middleware only uses blocking reads, and is likely the only program that will be accessing our device.

As our character device only supports reading of data, it has four functions associated with accessing it, in addition to two functions for registering and unregistering the device. The open and close functions do what their names suggest, as does the read function; all of these use Linux kernel wait queues and a boolean variable to ensure that only one process may read from the device at once and that the read is blocking. The poll function returns information on the status of data in the device; for our device it returns either that there is data to be read in a normal fashion or that there is no available data.

9.4 Middleware

Our middleware program is the glue that connects the several parts of our project together. Although the original version of the middleware program was not written by us it was modified so extensively that it is barely recognizable. There are two main threads in out middleware program: a control and usrdev thread and a Festival interface thread. The middleware program is still lacking on some features but is a solid foundation for future development work.

The original middleware program was written by the Linux 4D project [O4] and our work is a fork of their original version. Several changes were needed from that version in order to make the code work properly with our usrdev code. The first major change was that since we were generating the data using the standard character device interface instead of by adding a system call we needed to add file operations to the code and remove the signal handler that was used to inform the program that the system call had filled the buffer provided. To allow for quicker silencing of speech the middleware program was made threaded so that the control thread could simply kill the speaker thread and then start another. This change also necessitated the adding of a mutex for a buffer and appropriate calls to pthread_mutex_lock() and pthread_mutex_unlock() throughout the code. The original code's general modularity was maintained so that by simply changing an include and the init, close, and say_word functions could, in theory at least, allow for any TTS system to be used. The middleware program upon which our version is based was used for a practical example of how to use Festival's C API and little else, but this was still of great help to the project.

The two threads of the middleware program are the control thread and the Festival interface thread. The two of them communicate through access to the buffer and the integer variables buff_read and buff_to_read, access to all of which is controlled by a single mutex. The control thread of the middleware program reads from the usrdev device and creates silence as needed. The Festival interface thread communicates with Festival, sending text and executable commands. In terms of the buffer the control thread constantly fills it with read()'s and the Festival interface thread constantly depletes it with say_word()'s. To create silence the control thread will simply reset the two shared integers. Although this threaded approach may not be strictly necessary now it was in the past and possibly in the future when support for additional TTS programs is added. The majority of the work that remains necessary is in the middleware program.

The original program was a simple interface for a different way of dealing with Speakup but it gave us a reference point for how to deal with Festival from a simplistic C program. The original program was modified somewhat and then encapsulated in what is now the Festival interface thread while the control thread was written pretty much from scratch because of the different way that we were getting the data from Speakup to userland. The middleware program has a good basic set of functionality and is quite usable, if not yet perfect.

10 Appendix C: Installation instructions for the Code

These instructions assume that you already have a good working knowledge of linux and have re-compiled the kernel successfully at least once.

First have Festival properly installed and working (see their website at http://www.cstr.ed.ac.uk/projects/festival/).

Also have a fresh (no patches applied) copy of the source to a recent linux kernel (2.4.20 preferred) and have /usr/src/linux be a symlink that points to that directory.

When listing commands to be executed we include a % to indicate the system prompt, do not use the %'s when actually entering the commands.

Then get a copy of speakup version 1.5 and put it in /usr/src

```
% cp /path/to/speakup-1.5.tar.gz /usr/src/
% cd /usr/src
% tar -xzf speakup-1.5.tar.gz
```

and add our modifications to the new directory:

```
% cp /path/to/speakup-modifications.tar.gz /usr/src/speakup-1.5/
% cd /usr/src/speakup-1.5/
% tar -xzf speakup-modifications.tar.gz
```

then compile the middleware program for use later:

```
% cd middleware
% make
```

then patch the needed changes into speakup:

```
% cd /usr/src/speakup-1.5/speakup
% patch <speakup.patch
```

then go and patch the kernel:

```
% cd /usr/src/speakup-1.5
% ./install
```

then configure and install the kernel as per the Kernel-HOWTO (when configuring it is important to remember to include the usrdev console driver and to choose "usrdev" as the default synth):

```
% cd /usr/src/linux
% make config
% make dep
% make bzImage
% make modules
% make modules_install
```

Next copy the boot image to /boot, update your bootloader and reboot (this may vary based on the details of your system).

```
% cp arch/i386/boot/bzImage /boot/festival-enabled
% pico /etc/lilo.conf
% lilo
% shutdown -r now
```

Then see what major number was given and create an appropriate entry in /dev.

```
% mknod /dev/usrdev c 252 0
```

Note that the major/minor numbers are currently hardcoded in at compile time. To change the major number being used just change CHAR_REQUESTED_MAJOR_NUMBER in speakup_usrdev.h

Finally (and optionally, but recommended) add the Festival server and middleware program to your init scripts and when you reboot your computer should start talking right away.