A General Platform and Markup Language for Text to Speech Synthesis

by

Jordan Matthew Slott

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

Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 24, 1996

© Massachusetts Institute of Technology, 1996. All Rights Reserved.
A developer is faced with two challenges when writing applications which work with every commercially available text to speech synthesizer. The first challenge is controlling "presentation of speech" or the manner in which text is spoken. This includes how particular words, phrases, or constructs are spoken. A collection of "experts" identify and process these particular constructs so they are spoken in a natural manner. A markup language allows applications to control this process. The second challenge exists because each specific synthesizer presents its own API for which the developer cannot "special-case" code. A "Speech Manager" provides a single API to applications and access to many synthesizers.
To Mom, Dad, and Jon

and To, Aliza
Acknowledgments

Christopher Schmandt is the Principal Research Scientist of the Speech Interface Group at the M.I.T. Media Laboratory. I've worked with Chris for five years, beginning my first year as an undergraduate at M.I.T. He is responsible for all of my knowledge about speech and audio and a good deal of my computer science knowledge as well. One highlight of my five years at M.I.T. was discussing speech and audio architectural issues with him and solving the challenging technical problems which arise in computer science research. I thank Chris for the investment he placed in my future and the respect he has shown for my technical and scientific abilities.

Barry Arons graduated from the Speech Group with his Ph. D. in December 1992. I was fortunate to work with Barry the summer following my freshman year. During that summer we made improvements to our audio server, from which I learned a great deal about client-server computing. I have learned more about the art and science of software engineering from Barry than all of my classwork at M.I.T. Although long since graduated, this thesis would not have been possible without the software tools written by Barry.

Eric Baatz was extremely influential to the course of this thesis work, especially when it was only in its embryonic stages. Eric is an engineer for the Speech Application Group at Sun Microsystems Laboratories. I thank him for the insightful comments he has provided.

Jim Wright and Jamison Gray provided valuable insight, knowledge, and resources for this thesis. Jim Wright is the manager of the Computer-Telephony group at Sun Microsystems Computer Corporation. Jamison Gray is an engineer in the same organization who specializes in speech platforms and technologies. Much of Jamie's work parallels my own and I greatly appreciate the access he has provided me to his work.

Lisa Stifelman, my fearless officemate, has lent invaluable advice from her commanding knowledge of speech. Her enthusiasm is endless, and has been an inspiration for me in my work. Admittedly, at times, I make a quiet and nonresponsive officemate, and I thank her for her tolerance. I will certainly miss the entertaining war stories from the hockey rink.

There are a number of others who I wish to thank for their valuable contributions to my work in varying forms: Andy Breen, Earl Johnson, Manoj Muzumdar, Douglas Schreiber, James Tauber, Nicole Yankelovich.

This work was sponsored by Sun Microsystems, Inc.
Table of Contents

Acknowledgments 4

Table of Contents 5

Chapter 1. Introduction 9

1.1. Statement of the Problem ................................................................. 11
  1.1.1. The Presentation of Synthesized Speech ............................................ 11
  1.1.2. A Uniform API for Speech Synthesis ............................................... 15
1.2. Applications which use Text to Speech Synthesis ............................... 18
1.3. An Outline for the Remainder of this Thesis ....................................... 19
  1.3.1. Chapter Summaries ........................................................................... 19

Chapter 2. Speech Presentation 21

2.1. Related Work to Speech Presentation .................................................. 21
  2.1.1. Jabber ................................................................................................. 21
  2.1.2. ASTER ................................................................................................. 23
  2.1.3. FRont END (FRENĐ) .......................................................................... 24
  2.1.4. The Affect Editor ................................................................................ 25
  2.1.5. Direction Assistance .......................................................................... 25
2.2. A Useful Analogy to the Problem of Speech Presentation ....................... 25
2.3. Preprocessing of Text before Synthesis ............................................... 27
2.4. Text to Speech Markup Language (TTSMML) ..................................... 29
2.5. The Markup Stage of the TTSMML system .......................................... 31
  2.5.1. What is a Markup Expert? ................................................................. 31
  2.5.2. How tags are defined within an expert ............................................. 33
  2.5.3. Defining tags using Regular Expressions ........................................... 34
  2.5.4. Defining tags using other tags ............................................................ 36
  2.5.5. Defining tags using lexicons .............................................................. 37
  2.5.6. Grouping and Operators ................................................................. 39
Chapter 3. The Speech Manager 49

3.1. Overview of Some Commercial Text to Speech Synthesizers 50

3.1.1. Digital’s DECtalk 50
3.1.2. Entropic’s TrueTalk 50
3.1.3. British Telecom’s Laureate 51
3.1.4. Rsynth 51

3.2. Related Work to Speech Synthesis Platforms 52

3.2.1. Sun Microsystem’s SunSpeak 52
3.2.2. Microsoft’s SAPI (Speech API) 52

3.3. The Speech Manager and Uniform Speech API 53

3.3.1. The Speech Library 53
3.3.2. The Synthesizer-Dependent Device Drivers 54
3.3.3. The Audio Modules 55
3.3.4. Architectural Diagram of the Speech Manager 55

3.4. The Speech Library and Speech Manager API 56

3.4.1. Initializing the Speech Library and Speech Manager 57
3.4.2. Selecting the Third-Party Text to Speech Synthesizer at Run Time 57
3.4.3. Selecting the Audio Module at Run Time 58
3.4.4. Sending Text to the Speech Manager for Synthesis 58
3.4.5. The Speech Library Queueing Model for Text 59
3.4.6. Controlling the Flow of Synthesized Speech 61
3.4.7. Interrupting the Synthesis of Speech 61
3.4.8. Inserting Index Markers into the Text Stream 62
3.4.9. In-Stream Synthesizer Instructions 62
3.4.10. Prosody and Phonetic Character Sets 65
3.4.11. Specifying User Dictionaries 66
3.4.12. Directing Audio to a Particular Output 66
3.4.13. Opening and Closing the Output Destination 67
3.4.14. Controlling the Quality of Synthesized Audio Data 68
3.4.15. A Synthesizer-Dependent Extension Mechanism 68
3.4.16. Summary of the Speech Manager API 69

3.5. The Synthesizer-Dependent Device Drivers 71
3.5.1. Implementing an Asynchronous Interface ....................................................... 72
3.5.2. Implementing the Text Input Model ................................................................. 72
3.5.3. Obtaining the Synthesized Audio from the Speech Synthesizers ............... 73
3.5.4. Returning Status and Information to the Speech Library Asynchronously ... 74
3.5.5. In-Stream Synthesizer Instructions ................................................................. 75
3.5.6. Synthesizer-dependent Extension Messages .................................................. 75

3.6. The Audio Modules ............................................................................................ 76
3.7. Summary of the Speech Manager ................................................................. 77

Chapter 4. Discussion and Future Directions 79

4.1. Implementation Status Update ...................................................................... 79
4.1.1. The TTSML Preprocessing System ............................................................... 79
4.1.2. The Speech Manager ..................................................................................... 80
4.2. Future Directions ......................................................................................... 81
4.2.1. Prosodic Controls and Phonetic Character Sets ............................................. 81
4.2.2. Support for multiple languages in TTSML ................................................. 81
4.2.3. Definition of new tags using TTSML itself ..................................................... 82
4.2.4. A Simple Language to Simplify the Process Experts ................................... 82

References 83
Chapter 1  Introduction

Text to speech synthesis is rapidly becoming a popular means by which applications provide the user with information and feedback in the audio domain. Text to speech synthesis provides computer applications with the ability to communicate information in situations where a traditional graphical display is either unavailable or unusable. Text to speech technology places no inherent limits on the kinds of textual data which may be spoken. The applications employing text to speech synthesis speak the contents of a graphical display or read personal information such as electronic mail messages or calendar entries. The latter type of speech-based applications commonly benefits users who are away from the desktop with only the telephone as an interface.

Text to speech synthesizers are widely available on the commercial market today for nearly all desktop platforms. No doubt, the abundance of text to speech synthesizers encourages its use in applications. There are several reasons which explain the growing availability of text to speech synthesizers. The first reason is their manageable price. Each text to speech synthesis package bought for this thesis cost between $500 and $1000 per seat. This number drops sharply for the higher volume personal computer market. Most of the recent cost reduction of this technology owes to exponentially increasing processing
power. No longer do text to speech synthesizers require specialized hardware, most are available as software-only development kits. An additional factor contributing to the growth of speech-based applications is the trend that audio hardware is becoming standard on most workstations and PCs.

While the availability of text to speech technology encourages its use in applications, it unfortunately presents some liabilities. Since many different text to speech synthesizers exist, application developers have extreme difficulty ensuring their applications work universally with all text to speech synthesizers. First and foremost, every text to speech synthesizer has its own Application Programmer’s Interface (API). Synthesizing unconstrained text, such as electronic mail messages or news stories, presents another liability to the already bewildered developer. English is a rich language, full of special constructs and phrases, punctuation, names, and jargon. Providing the user with the most understandable speech is of paramount importance. Applications, therefore, must ensure that all of the above are spoken correctly. These two liabilities of text to speech synthesis are the fundamental problems addressed by this thesis.

This thesis focuses on text to speech technology with respect to the application layer. Its primary goal is to provide a clearer understanding of the issues and challenges of developing speech synthesis-based applications. Equally important, it describes the design and implementation of software tools and systems developed as part of this thesis’ research. These systems alleviate the two liabilities mentioned above, and make the robust development of speech synthesis-based applications a reality.

The remainder of this chapter defines the problems mentioned above and addressed by this thesis. Section 1.1 proceeds with the statement of the two problems considered by this thesis. Section 1.2 briefly discusses current research and applications which employ text to speech synthesis. The purpose is to provide the reader with some background knowledge of the kinds of applications which benefit from this thesis. Section 1.3 concludes with an outline for the remainder of this document.
1.1. Statement of the Problem

This section focuses on the problems and difficulties associated with building speech synthesis-based applications. The two fundamental problems considered here are: 1) How does an application ensure that arbitrary, unconstrained text is spoken correctly and in the most natural manner? 2) How does an application developer write an application which works with every text to speech synthesizer on the market?

1.1.1 The Presentation of Synthesized Speech

This section contains an in-depth explanation of the problem of speech presentation. Speech presentation concerns itself with the various issues arising from correctly speaking arbitrary text. The English language contains a rich set of constructs, phrases, and words which convey different meanings. A construct is a collection of words and/or punctuation symbols which conveys some piece of information. A common example of a construct is a telephone number or electronic mail address. Text to speech synthesizers cannot possibly pronounce all of the various constructs, phrases, and words correctly for a given piece of text. This is particularly true if the text is likely to contain less frequent words and constructs, such as electronic mail messages and news stories. The difficulties text to speech synthesizers have speaking text may be classified into a number of categories. One challenge of this thesis is to identify all possible categories of constructs, words, and phrases which text to speech synthesizers typically have difficulty with. The problem of speech presentation, therefore, is best defined in terms of the following categories.

1. Proper Names. These are the most common words which are mispronounced by text to speech synthesizers. Synthesizers often allow application to specify an exception dictionary for this purpose. One example is the last name Stifelman. DECTalk, [DECTALK 1987] for example, pronounces Stifelman with a “short i” rather than properly, with a “long i”. Most often, a phonetically spelled version of a proper name corrects any mis-
pronunciations.

2. **Abbreviations.** Text to speech synthesizers sometimes have support for abbreviations. Trueta[rk, [TRUETALK 1995] for example, maintains an internal list of abbreviations. Any abbreviation which is on this list is replaced with the corresponding word expansion. This list of abbreviations cannot be modified by the application, however. Exception dictionaries, most often used for proper names, are used for abbreviations as well. If a text to speech synthesizer cannot decipher the abbreviation, it often speaks the abbreviation letter-by-letter or attempts to pronounce the abbreviation as a normal English word. For example, if the speech synthesizer does not recognize `mtg.` as `meeting`, it may incorrectly pronounce it as `emm tughh`.

3. **Acronyms.** This category of word type is closely related to abbreviations. Often text to speech synthesizers maintain lists of acronyms as well. How acronyms are properly spoken varies, however. Expanding an acronym into its constituent words is not always the correct speaking of the acronym. For example, `A.M.` is never pronounced `ante meridian`. Yet, the acronym `nafta` is rarely pronounced letter-by-letter as `n a f t a`. Rather, it is commonly pronounced as an English word. Sometimes application may desire that is be spoken as `North American free trade agreement`. Because speech presentation is focused on the most natural way to speak words, these minor details are significant.

4. **Contractions.** This category is similar to both abbreviations and acronyms. With respect to the presentation of speech, there are two ways to present a contraction. They may be spoken either as English words or they may be expanded into two words. For example, `can’t` may be spoken as `kant` or `can not`. Applications desiring more formal speech will chose the later. This assumes that speech synthesizers are able to recognize `can’t` as a contractions to begin with.

5. **Compound Words.** Compound words are often spoken correctly by separating them into two or more separate words. Many compound words may be detected by simply
searching for capitalized letters or punctuation which separates the word into two distinct words. For example, \textit{i-net} and \textit{OS/2} are such words. Often a compound word consists of a word concatenated with a number, such as \textit{12th}.

6. \textbf{Ambiguous Abbreviations}. Some abbreviations have different meanings depending upon their contexts. A notorious example is the \textit{St.} abbreviation which either means \textit{Street} if it follows a proper name, or means \textit{Saint} if it is followed by a proper name. The abbreviation \textit{Dr.} is another example. These special abbreviations are kept in a separate category because handling them are far more complicated. These abbreviations may be pronounced as \textit{S T period}, for example, however this is not the most natural way to speak them.

7. \textbf{Common English Constructs}. This category includes dates, time of day, telephone numbers and other constructs which are common in English text. Each type of construct has its own syntax, and often there are numerous ways to represent the same information. The following are all the same dates: \textit{January 1, 1990; Jan 1 1990; 1 jan 1990; 01-01-90; 01/01/90; 1/1/90}. Pronouncing all of the punctuation explicitly is clearly not natural. The majority of these constructs can often be described by a catalog of patterns.

8. \textbf{Mathematical Expressions}. Fractions, equations, and dollar amounts are often present in text. Mathematics presents its own rich syntax for conveying information. Most synthesizers fail to pronounce mathematical expressions correctly, resulting in their pronunciation letter-by-letter. Truetalk [TRUETALK 1995] is unique and has a special “math mode” which detects various forms of mathematical expressions.

9. \textbf{Domain specific jargon}. Each profession contributes its own jargon and constructs. The computer industry has complicated constructs such as electronic mail addresses and URLs. Although electronic mail addresses are common now, they are still mispronounced by our version of DECTalk [DECTALK 1987] and Entropic’s Truetalk [TRUETALK 1995]. In addition to constructs, it contributes its own jargon words such as
10. **Plural Words.** Text to speech synthesizers handle most plural words correctly. However, such items as proper names and abbreviations are pluralized. In this case, the correct plural ending must be added to the end of the word. For example, if a proper name is translated into its phonetic spelling, the type of plural ending added depends upon the phonetic spelling of the pluralized word. Here, the mechanism to handle plural words must be coordinated with the handling of other types of constructs.

11. **Possessive Words.** Possessive words are handled similarly to plural words. The pronunciation of possessive words must also be coordinated with the mechanism which determines how proper names are spoken, for example.

12. **Punctuation.** Most often, punctuation is handled correctly by text to speech synthesizers. Often, excessive use of punctuation is used as delineators. Many text to speech synthesizers pronounce these excess punctuation symbols explicitly. Listening to a speech synthesizer pronounce eighty exclamation points is unbearable. Excessive punctuation can be reduced to a single punctuation mark before causing the speech synthesizer convulsions.

13. **Parts of Speech.** Certain words have two different pronunciations depending upon their parts of speech in the context of the sentence in which they appear. For example, *Record*, in the sentence, *Record your message now* is used as a verb. The DECTalk synthesizer pronounces *Record* as a noun (as in phonograph record). This is clearly wrong.

It is hopefully clear by now the many issues the problem of speech presentation concerns itself with. Developing successful applications which synthesize text depends upon the correct pronunciation of all parts of the spoken text. Users expect to hear the text as humans would naturally speak the text. The system developed to address the problem of speech presentation is presented later in this thesis, detailing the mechanisms by which these various issues mentioned above may be solved.
1.1.2 A Uniform API for Speech Synthesis

It is not only unrealistic, but nearly impossible for application developers to write applications which work with every text to speech synthesizer. One might believe that carefully “special case-ing” code for the various speech synthesis API’s is all that is needed. There are, however, much deeper issues which prevent this immediate solution to the problem. This section carefully examines the various differences among text to speech synthesizers which currently makes synthesizer-independent application development extremely difficult. The goal, ultimately, is to provide a single Application Programmer’s Interface (API) and system for text to speech synthesis which unifies the disparities discussed below.

1. **Varying Functional Interfaces (API’s).** Each individual text to speech synthesizer has its own set of interface function calls. Applications use these function calls to, for example, speak text, control the flow of speech, set parameters related to the synthesis, etc. On a simplistic level, a common function name is needed for similar functions. For example, one synthesizer may have the function `tts_speak_string()` to speak text, while another has `speak()` for speaking text. A standard nomenclature and naming convention is needed on this fundamental level.

2. **Synchronous vs. Asynchronous Programming Model.** Text to speech synthesizer API’s present not only a set of function calls to the application, but also either a *synchronous* or *asynchronous* programming model. In a *synchronous* programming model, function calls do no return until they complete their tasks. The Rsynth [Ing-Simmons 1995] and Laureatte [Ashworth 1995] text to speech synthesizers provide such a synchronous interface. The “speak” commands, for example, do not return until all of the text has actually been spoken. An *asynchronous* programming model implies that functions return immediately before their task is complete. The X Windows Instrinsics Toolkit [Nye 1990] is a common example of a non-speech-based asynchronous API. Both DECTalk [DECTALK 1987] and Truetalk [TRUETALK 1995] present an asyn-
chronous programming model to the application. These synthesizers often include a stand-alone server executable and an asynchronous API library to communicate with the server.

3. **Varying Functionality.** In most cases, text to speech synthesizers not only have different sets of function calls (item 1), they have different functionality. For example, one synthesizer may return the input text as a string of phonemes in addition to the audio, while others may not. Another synthesizer may not be able to pause the output of speech in the middle of an utterance while most are capable of this. Application developers require a standard specification for the functionality of the underlying text to speech platform.

4. **Extension Mechanism.** In light of item 3, a uniform text to speech API should not completely restrict applications from the functionality of any particular text to speech synthesizer. As a means of last resort, applications should possess the ability to send synthesizer-specific messages to the text to speech synthesizer currently in use. While only that particular synthesizer will understand the message, it is a more attractive option than not allowing applications access to that functionality at all.

5. **Text Input Model.** Each text to speech synthesizer presents to applications a different model for text input. The *model for text input* means: 1) the amount of text input accepted at once; and 2) the boundary on which text input is accepted. The amount of text which may be handled by speech synthesizers at once may vary between one thousand bytes to tens of thousands of bytes. The boundaries on which text is accepted is often along either word, phrase, or sentence boundaries. In order to develop an application independent of any text to speech synthesizer, a single text input model is needed. Also, a well-defined error mechanism must also exist when the application violates the specified text input model.

6. **Different Destinations to Send Audio.** Each text to speech synthesizer has its own way to handle the synthesized audio. Some synthesizers output audio directly to a telephone
line or line output plugged into a hardware unit [DECTALK 1987]. Others only output the audio to the default audio device on the workstation or to a file on disk. Many text to speech synthesizers provide a copy of the audio data to the application. An application developer needs flexibility in the number and types of destinations it can send the audio data. For example, screen reading applications mentioned later output audio to the desktop audio device, while the telephone-based electronic mail readers output audio to an analog telephone or ISDN line. Restricting the places an application can send the synthesized audio is one of the most serious limitations faced by speech application developers.

7. **Different Mechanisms to Handle Audio.** In addition to item 6, each workstation may have a different mechanism or platform to handle audio. On some workstations, audio is played via an interface to an operating system audio driver. On other workstations, a more elaborate audio platform or server is used to play audio data. Examples of such audio servers include Digital Equipment Corporation's AudioFile [Levergood 1993] and the Média Lab's Audio Server. [Arons 1992][Slott 1996] Applications should use the particular audio handling mechanism in use on their workstation.

8. **Varying Types of Status Notification.** Applications need some way of knowing when speech output has begun and completed. Each text to speech synthesizer has its own way of notifying the application of such status changes. The manner in which applications are informed of this information also depends upon whether the synthesizer presents a synchronous or an asynchronous programming model (item 2).

9. **Synchronization.** At times, applications synchronize the output of synthesized speech with other forms of feedback. For example, applications may want to update a graphical user interface when certain portions in the text are spoken. Each text to speech synthesizer has its own mechanism for synchronization, if at all.

10. **In-Stream Synthesizer-Specific Instructions.** Text to speech synthesizers permit applications to control a number of parameters which affect qualities of the voice out-
put. For example, the speaking rate (i.e. words per minute) and speaking voice (i.e.
man or female) are two such parameters. Other examples include placing phonetic
spellings for words within the text, adding pauses, adding emphasis to certain words, or
other prosodic or rhythmic controls. Often these parameters are controlled via in-
stream synthesizer instructions. These in-stream instructions often take the form of
some escape sequence placed within the text stream. These instructions set the value
for that particular parameter for all subsequent text. Each text to speech synthesizer has
its own, often unique, set of escape sequences. Applications can not robustly account
for all possible variations in the in-stream synthesizer instructions.

11. User Dictionaries. Almost all text to speech synthesizers allow applications to provide
an exception dictionary. These exception dictionaries consist of words paired with their
phonetic spelling and allow applications to correct the pronunciation of hard-to-pro-
nounce words. Often, words in exception dictionaries are proper names. Each synthe-
sizer has its own dictionary format. An application can not reasonable supply and
maintain a separate exception dictionary for every possible text to speech synthesizer.
In fact, this item is related to the first problem considered by this thesis, and is dis-
cussed further in that context.

1.2. Applications which use Text to Speech Synthesis

Two types of speech synthesis-based applications receive a fair amount of recent research.
The first genre of applications synthesize information stored as text. Speech is common-
place in applications which read telephone numbers in a directory assistance application,
frequent flyer numbers in an airline application, or account balances in a banking applica-
tion. However, these systems often rely upon the concatenation of pre-recorded human
speech, speaking only a small domain of words and phrases. The most interesting use of
text to speech synthesis is speaking unconstrained text such as electronic mail messages,
news stories, or weather information. The second class of applications are screen readers.
The primary beneficiaries of this technology are those users with visual disabilities. Screen readers are also applicable in “hands-eyes busy” situations.


A number of systems read information from a graphical display. EmacsSpeak [Raman 95] is a system which reads everything in an emacs buffer while the user navigates through the text. The Mercator project [Edwards 1994] is a general screen reader, presenting a speech-based nonvisual “view” of the computer terminal. It creates and uses extensions to the X Windows System to obtain the needed textual information before lost to bitmap form.

1.3. An Outline for the Remainder of this Thesis

This introduction described the two fundamental problems addressed by this thesis with respect to text to speech synthesis: speech presentation and a uniform API. The remainder of this thesis is dedicated to further elaboration on these two problems and the description of the design and implementation of software systems which solves them. A summary of the remaining chapters in this thesis is found below.

1.3.1 Chapter Summaries

*Chapter 2: Speech Presentation* describes a flexible and extensible system
which analyzes and processes text before it is sent to a text to speech synthesizer. A markup language, text to speech markup language (TTSML) is introduced, used by both the system and applications to indicate how and which parts of input text is processed.

*Chapter 3: The Speech Manager* describes the design of a uniform text to speech synthesis API. A system called the Speech Manager integrates multiple third-party text to speech synthesizers for use by applications. This chapter discusses the solutions and compromises used by the Speech Manager as they pertain to the issues mentioned above.

*Chapter 4: Discussion and Future Directions* discusses applications built around the systems described in this thesis as well as a concluding discussion on the work presented within and remarks about some future directions.
Chapter 2 Speech Presentation

This Chapter discusses the second major problem considered by this thesis: Speech Presentation. As stated in Chapter 1: Introduction, speech presentation concerns itself with two issues: 1) how applications ensure that text is spoken properly, and 2) how applications control the way in which text is spoken. Section 2.1 overviews related work. Section 2.2 presents an analogy to speech presentation from the domain of text formatting and text presentation. Section 2.3 introduces the system developed by this thesis to address the problem of speech presentation. The remaining sections of this chapter provide the details to this system.

2.1 Related Work to Speech Presentation

This section presents previous work done in the area of preprocessing and formatting text for the purpose of text to speech synthesis or some form of audio presentation of the text.

2.1.1 Jabber

Jabber [Lee 1994] is the preprocessing system for text to speech on which this section of the thesis work is based. Jabber addresses many of the speech presentation issues dis-
cussed here. In some ways, this work is an extension of Jabber. Much of the knowledge about the regular expressions descriptions for different types of text constructs is borrowed from Jabber.

The Jabber system uses the lex/yacc tools [Levine 1992] to identify parts of the input text which require special attention. In lex, a number of tokens are defined: words associated with regular expressions. These tokens are similar to the tags discussed in this thesis. The tokens in lex are then combined in different ways to form rules in yacc. The yacc parser, along with lex, analyzes the input text stream. When yacc matches one of its rules to some portion of the input text stream, it calls some C function. Each rule in Jabber has its own C function. The purpose of the C function is to process the input text portion in such a way that it subsequently gets pronounced properly by a text to speech synthesizer.

We will see shortly that the high level organization of the preprocessing system built for this thesis resembles this rule-action structure in a simple way. While the Jabber system performed adequately for a few years, it has several limitation and drawbacks. The lex/yacc tools used for Jabber is one source of its inflexibility. In order to add new rules, one must recompile the entire system. This may not be possible if Jabber is distributed to a remote location. New rules may not be added while the system is running. Rules inside Jabber are contained within one linear list. One major difficulty encountered with Jabber is the interaction between these rules often causes unexpected results. If a new rule is added to jabber, care must be taken as to exactly where in the list of rules this new rule is inserted. In many cases, a new rule A is inserted before an old rule B. The regular expression for rule A matches some text that normally rule B would and should match. The text is processed in a different way by rule A from which it should be processed by rule B. These types of bugs in the ruleset are often difficult to find. Consider the following example. The phrase MS-DOS which used to be pronounced M S DOS is now incorrectly pro-
nounced as *Mississippi hyphen DOS*. The addition of rule 3 causes this error.

<table>
<thead>
<tr>
<th>Old ruleset:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 1: speak &quot;MS-DOS&quot; as &quot;M S DOS&quot;.</td>
</tr>
<tr>
<td>Rule 2: speak all two letter state abbreviations explicitly.</td>
</tr>
<tr>
<td>&gt; &quot;MS-DOS&quot; is spoken as &quot;M S DOS&quot; (by rule 1)</td>
</tr>
<tr>
<td>&gt; &quot;I live in MS&quot; is spoken as &quot;I live in Mississippi&quot; (by rule 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New ruleset, adding rule 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 3: break hyphenated words into separate words, and speak the hyphen.</td>
</tr>
<tr>
<td>Rule 1: speak &quot;MS-DOS&quot; as &quot;M S DOS&quot;.</td>
</tr>
<tr>
<td>Rule 2: speak all two letter state abbreviations explicitly.</td>
</tr>
<tr>
<td>&gt; &quot;MS-DOS&quot; is spoken as &quot;Mississippi hyphen DOS&quot; (by rule 3, then rule 2)</td>
</tr>
<tr>
<td>&gt; &quot;I live in MS&quot; is spoken as &quot;I live in Mississippi&quot; (by rule 2)</td>
</tr>
</tbody>
</table>

Figure 1. Example of the side effects of a misplaced rule in Jabber.

Another limitation of Jabber is applications cannot “hint” to the system if it knows certain portions of the text which require special processing. Only that text which is identified by some Jabber rule is subsequently processed.

2.1.2 *A₅TₑR*

In his Ph. D. dissertation, T. V. Raman describes a system which speaks text written using LaTeX. His system, *A₅TₑR*, [Raman 1994] interprets the LaTeX markup within the document to identify and determine the structure of the document. Examples of specific structural objects include paragraphs and words, lists, tables, and mathematical equations. *A₅TₑR* specializes in reading technical documents which often contain numerous mathematical expressions. Raman states that *A₅TₑR* is based upon the premise that a document is composed of a number of objects. The effectiveness of *A₅TₑR* is tied to how well a document’s structure is identified. In the absence of LaTeX markup, *A₅TₑR* identifies only basic paragraph breaks and quoted passages.

Once a document’s structure is represented internally, *A₅TₑR* “renders” this document to audio using text-to-speech synthesis. This is accomplished using AFL or Audio Format-
ting Language, a rule-based mechanism to decide how text is spoken. Parameters which
determine how text is rendered, such as the speaking voice and rate, are embedded within
these AFL rules. AFL also provides a means to define “reading rules,” Common Lisp
functions which provide some sort of transformation of the textual data. This is especially
useful for equations, which often need preprocessing to convert the mathematical sym-
bols, quantifiers, and operators into words before spoken.

2.1.3 FRont END (FREND)

FREND is the preprocessing stage for the AT&T Bell Laboratories Text-to-Speech
system. [Pavlovck 1991] This later became Entropic’s TrueTalk synthesizer. [TRU-
ETALK 1995] The FREND preprocessing stage modifies text before synthesized into
audio and increases the accuracy of the pronunciation of text. It is of importance as related
work to speech presentation since the system presented in this thesis does many similar
tasks as FREND.

Generally, FREND applies a set of rules to incoming text and expands parts of it.
Often these “parts” are abbreviations or mathematical symbols which are expanded into
full words, or proper names which are expanded into their phonetic spellings. FREND cat-
egories these parts of text into “text classes.” For example, telephone numbers, times, and
dates are all separate text classes. Each text class has a set of rules which provide regular
expressions to identify text which belongs in a text class. FREND understands a special
language, PARS (Parse Action Rules System) which instructs the system using a pattern
matching scheme on how to expand text classes. A PARS compiler is provided to convert
the source into a format understood by FREND. The FREND system is similar to the one
presented in this thesis in the way it acknowledges the importance of the order in which its
text classes are processed. It is also significant because it distinguishes between the stage
of preprocessing where text classes are identified and where these text classes are pro-
cessed.
2.1.4 The Affect Editor

The Affect Editor, a system created by Janet Cahn for her Master's Thesis, adds emotional affects to synthesized speech. [Cahn 1990a,b] Synthesized speech has a reputation for poor pronunciation and a monotonic, awkward voice. Cahn's system adds prosody or rhythm to synthesized speech. This enhances its understandability and naturalness, and conveys additional meaning in synthesized speech beyond just the words themselves. Based on research into the acoustical characteristics of speech with emotion, Cahn parameterizes the various desired effects. These parameters are controlled independently and may be loosely categorized as "pitch, timing, voice quality, and articulation." [Cahn 1990b] The Affect Editor system transforms a high-level description of an emotion to a corresponding set of values for her parameters. This system uses DECTalk [DECTALK 1987] for its text to speech synthesis.

2.1.5 Direction Assistance

Direction Assistance [Davis 1988] provides a user with spoken driving directions between two locations over the telephone. Using information manually embedded within the description of routes, Direction Assistance adds intonational information to the synthesized speech. Based on the type of direction being given, the Direction Assistance system will vary such parameters of the synthesized speech as intonational contour, pitch range, pause, and final lowering to add additional meaning to the utterance.

2.2 A Useful Analogy to the Problem of Speech Presentation

The problem of speech presentation is not without analogies. Before describing a solution to the speech presentation problem described above, the problem of text formatting or text presentation is outlined. [Sproull 1995] Text formatting serves as a useful analogy to speech presentation because the process of each is similar. Text formatting is the problem of taking plain text and presenting it to the user graphically, in an easy-to-read manner.
This problem is the basis for advanced word processors, which display text in different fonts, with varying margins, spacing, indentation, etc. Recently, more advanced software allows text to appear in two columns with headers and footers, flowing around an inset of a picture or graphic. The idea underlying the need for these applications is this: the readability of a document is based, to a large extent, on how that document is presented. Not only do the words of a document carry meaning, so does the way in which the words are presented. The analogy here is the relationship between the presentation of text and the presentation of speech. The presentation of speech is critical to the understandability of the spoken text, just as the presentation of text is critical to the understandability of the written text.

The process by which a user interacts with a text formatting system provides a useful analogy to the problem speech presentation. The most popular tools for text formatting in the UNIX Operating System are FrameMaker by Frame Corporation, Inc. [FRAME 1991], LaTeX [Lamport 1986] and its parent, TeX [Knuth 1984][Knuth 1986]. When users present text using one of these tools, a three step process is undertaken. The first step involves entering the raw text into electronic form. During the second step, users identify certain portions of the raw text which are of a certain type. For example, portions of the raw text might be “headlines”, “items in a list”, “footnotes”, or “emphasized terminology”. Note that this identification says nothing about what the text is supposed to look like (although users probably have an idea of what each item should look like). The second step only identifies certain portions of the text as belonging to a certain type. The third step defines how each type of text looks. Often, this definition is in terms of a number of well-known parameters such as: font, point size, angle (italics), weight (bold), inter-line spacing, inter-word spacing, etc. Each portion of text of a certain type is presented according to

1. I first heard the topic of this section of my thesis described as “Speech Presentation” by Robert Sproull, Director of Sun Microsystems Laboratories East. He also was the first to suggest the analogy with text formatting. It caused me to analyze the process of text formatting and make the analogy with the design principles of the preprocessing system described by this thesis. He probably doesn’t realize he made such a contribution, however I cite the personal communication regardless.
how that type is defined in terms of these parameters. Changing the definition of a text type affects all text labelled with that type. The key to this process is the separation of identifying text to be of a certain type and defining what that type is supposed to look like. Often these text formatting tools have some sort of "language" to describe text types. This central idea to the text formatting process is applied to the speech synthesis domain as discussed later.

In FrameMaker [FRAME 1991] users enter text into the application’s window. Using the mouse, users position the cursor in a particular paragraph or highlight certain words. The user then selects a "paragraph type" or a "character type" from a list. The paragraph or characters assume the type selected and is presented according to its definition. In order to change the appearance of a paragraph or character type, the user chooses the "Format, Paragraph" or "Format, Character" option and selects the values for a number of well-defined parameters. The "language" by which each paragraph or character type is described is not made explicit to the user, however; some representation is used by the FrameMaker file format. We see a similar process using LaTeX. [Lamport 1986] Text is "tagged" using the "\begin{tag}" and "\end{tag}" delimiters. Then each tag is defined with a "macro," using the language exported by LaTeX to users to describe different types of text. Since users edit their text in LaTeX using only an ASCII-based text editor, they do not see a real-time presentation of their document.

2.3 Preprocessing of Text before Synthesis

This thesis describes a text preprocessing system. Text is preprocessed before sent to a text to speech synthesizer. This preprocessing stage tackles the various problems associated with the proper speech presentation of text. This system, viewed as a black box, accepts text from the application as input and generates a modified form of this input text as output. This modified text is altered in such a way that words or phrases that would normally be mispronounced by a text to speech synthesizer are now in some synthesizer-inde-
pendent representation. This representation causes the text to speech synthesizer to pronounce this word or phrase correctly. Chapter 1: Introduction discussed the difference types of words, phrases, and constructs in the English language which might require special preprocessing.

Most text to speech synthesizers perform their own preprocessing of input text. The preprocessing stages buried inside these synthesizers are concerned with the same speech presentation issues discussed earlier. FREND, [Pavlovcik 1991] for example is TrueTalk’s [TRUETALK 1995] preprocessing system. The Jabber project provided additional processing to correct any mistakes made by the DECtalk [DECTALK 1987] synthesizer specifically. It no longer makes sense to hide the processing stage from the application as most synthesizers do. Applications know a great deal of context about the text which they want spoken. Text to speech synthesizers can use this context to aid in their pronunciation of the text. Speech synthesizers can speak email addresses, for example, more naturally if they know the text is an email address to begin with. An application which announces a person’s electronic mail by speaking the sender knows that the text it sends to a synthesizer is some sort of email address. The ability for an application to provide context to a speech synthesizer is a primary motivation for building the preprocessing system in this thesis. Another motivation for this preprocessing system is the end-to-end argument [Saltzer 1984]. Because many speech synthesizers exist today, the preprocessing system ensures that text is spoken similarly across all synthesizers. In this way, its applicability far exceeds Jabber.

The architecture of this preprocessing system naturally divides into two parts. The first part attempts to identify those words or phrases which might need special processing in order to be pronounced correctly. For reasons which become apparent later, this first stage of the preprocessing system is called the Markup Stage. The second stage takes these specially-identified phrases or words from the Markup Stage and transforms them into the synthesizer-independent representation discussed above. This second stage is called the
**Process Stage.** This two-stage architecture is also motivated by the argument in the previous paragraph. Since applications may provide context for certain portions of the input text, these portions may bypass the *Markup Stage* of this system. Note that our analogy with text formatting breaks its processing along the same boundaries. The diagram below depicts these two stages of the preprocessing system. In this example, the input text is transformed using a combination of phonetically spelled words and the explicit spelling of numbers.

![Diagram of two stages of preprocessing: Markup and Process](image)

**Figure 2. The two stages of preprocessing: Markup and Process**

There is one element yet to be discussed from this picture. How does the Process Stage know about the specially-identified words, phrases, or constructs from the Markup Stage? As the name implies, the Markup Stage will *tag* the text items it identifies. In the example above, the person’s name *Atty* is tagged with a *person* tag. The five-digit telephone extension *x5-1212* is tagged appropriately as well. The collection of tags used by the Markup Stage is called the Text to Speech Markup Language (TTSML) and is the focus of the next section.

### 2.4 Text to Speech Markup Language (TTSML)

The Text to Speech Markup Language borrows many ideas from SGML (Standard,
Generalized Markup Language) [Smith 1992], the syntax of each tag in particular. HTML [HTML 1996] is a popular deviant of SGML as well. TTSML is composed of a set of tags. Each of these tags identifies a portion of text as being of a certain type. For example, there may be (and is) a tag for electronic mail addresses. This tag is named email. Postal addresses, times of days, dates, URLs, and proper names are more examples of types of text items which have tags in TTSML. These tags, as in SGML, are meant to only identify certain portions of text. In this way, these tags are descriptive, rather than procedural. A TTSML tag does not say what should be done with text, only what type of text it is.

When a portion of text is identified by a tag, it is said to be marked-up. Marked-up text is identified by both a beginning and ending tag of the same type. The structure of marked-up text comes directly from SGML. For example, a marked-up telephone number is: 

\[ <phone><prefix>555</prefix>-1212</phone> \]

Ending tags includes a forward slash before the tag name. Each beginning tag must have a corresponding ending tag in TTSML. Tags are permitted to be nested inside of other tags, however, this nesting must be proper. In other words, 

\[ <a>Hello. <b>Bye</b></a> \]

is properly nested, while 

\[ <a>Hello. <b>Bye</b></a> \]

is not properly nested. Often, nesting is used by TTSML to further identify the specific structure within tag, as exemplified by the phone tag above. This feature is useful when tags are composed of discrete, identifiable components. For example, each particular item in an electronic mail address may be identified as follows: 

\[ <email><username>hordack</username>@<netname>mit</netname>.<netname>edu</netname></email> \]

The email tag contains the username and netname tags.

The architectural diagram of the TTSML preprocessing system depicted above is now complete. The Markup Stage uses TTSML to communicate to the Process Stage which portions of text require special attention. Hopefully, it should be now obvious why the first stage is called the Markup Stage. One powerful attribute of this system is that nothing restricts applications from using these TTSML tags themselves, often to “hint” to the system when they believe a certain portion of text requires special attention. TTSML is the
mechanism which exposes the preprocessing stage to the application.

2.5 The Markup Stage of the TTSML system

The Markup Stage takes raw input text and tags certain portions of the text. These marked-up portions identify the text as being of a certain type: email addresses, postal addresses, proper names, dates, fractions, etc. The Markup Stage is only concerned with the identification of these text items. It is not concerned with how these items are to be spoken. This distinction is an important one. The Markup Stage attempts to identify those types of text which it knows about. All of the different types of text it knows about are represented by the collection of all the tags in TTSML and the knowledge contained within its Markup Experts.

2.5.1 What is a Markup Expert?

The Markup Stage is further organized into Markup Experts. Each expert has specific knowledge about particular types of text. For example, a mathematical expert knows what fractions, equations, roman numerals, etc. look like. A computer experts knows what an electronic mail address, URL, and certain computer-related jargon looks like. These experts analyze an input text stream and identify portions of the text which fit the descriptions it knows about. The markup expert then surrounds the identified part of text with the appropriate identifying TTSML tag. The diagram below depicts this process with some
example experts.

![Diagram of Markup Experts in action]

**Figure 3. An example of Markup Experts in action**

The first observation about the markup stage is there is some order in which markup experts are queried for their knowledge. In fact, the order in which experts are queried is important and not arbitrary. Intuitively, there are some types of text we want to identify first, especially if different types of text are imbedded within one another.

In the language of object-oriented programming systems (OOPS) [Liskov 1986], each expert is an *object*. In the simplest terms, an object is a self-contained entity which possesses an *interface* and *state*. The state for each expert is the knowledge it contains about its tags. The second component of an object is an interface, a set of functions which permit the world to access this object. In terms of markup experts, their interface allows the world to query whether it can identify portions of the input text which match its tags. More complicated definitions of OOPS allow *inheritance*--objects borrowing functionality from other objects. Markup Experts do possess some form of inheritance--the ability to make use of another expert's knowledge about certain tags.

The primary advantage of organizing the markup stage into distinct experts is one of
maintainability and expandability. One major problem with the Jabber system is the addition of new rules. When adding new rules to Jabber, care is needed so that the rule is placed in the proper position of some one-hundred rules. The placement of a new rule is arbitrary and the possible side effects are discovered only later. Discovering the cause for these side effects is nearly impossible. To repeat an example used previously, a misplaced rule caused MS-DOS to be pronounced as Mississippi DOS. Markup experts allow knowledge about text items to be categorized by domain. Each domain of knowledge may be kept separate and maintained separately. If the markup system requires knowledge about a completely new domain of text, a third-party may author the necessary expert.

Markup experts are integrated into the system by listing them in a configuration file. The order in which markup experts are listed determines the order in which they are consulted during the Markup Stage. The following is an excerpt from an example configuration file. The first field is the name of the markup expert, and the second field is the full path name of the markup expert.

<table>
<thead>
<tr>
<th>Name</th>
<th>Module Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>computermarkup.expert</td>
</tr>
<tr>
<td>political</td>
<td>politicalmarkup.expert</td>
</tr>
<tr>
<td>compound</td>
<td>compoundmarkup.expert</td>
</tr>
<tr>
<td>possessive</td>
<td>possessivemarkup.expert</td>
</tr>
<tr>
<td>plural</td>
<td>pluralmarkup.expert</td>
</tr>
<tr>
<td>people</td>
<td>peoplemarkup.expert</td>
</tr>
</tbody>
</table>

Figure 4. Example of a configuration file

2.5.2 How tags are defined within an expert

Defining a tag within an expert requires listing the name of the new tag, an expression defining this type of text, and whether the tag is exported or nonexported, each on separate lines. A markup expert, therefore, is a disk file which contains any number of these tag definitions which are categorized under one domain. A markup expert file resembles the
portion of the telephone markup expert shown below.

```plaintext
# The Markup Expert for telephone numbers
exchange
[0-9][3][-]+  NONEXPORTED

phone
<exchange>[0-9][4]  EXPORTED

extension
x[0-9][-][0-9][4]  EXPORTED
```

**Figure 5. Example of a Markup Expert.**

What it means for a tag to be exported versus nonexported is discussed later. The most important part of a tag definition is defining what text of the tag’s type looks like. For example, the tag for a five digit telephone extension is defined as follows: “the letter x, followed by a digit between zero and nine, followed by a minus sign, followed by a four digit number. What language does the author of an expert use to describe such a tag? The next few subsections provide a rich set of building blocks to describe what text of a tag’s type looks like. The second line of each tag definition contains an expression using these building blocks. These building blocks are based primarily on regular expressions, but contain other powerful constructs.

### 2.5.3 Defining tags using Regular Expressions

A regular expression is a pattern to which input text is matched. There are a number of rules, which are not discussed in-depth here, which dictate how regular expressions are constructed, including the operators which may be applied to regular expressions and the operations which allows the combination of regular expressions. For further background on regular expressions, consult [Levine 1992][Martin 1991][Sipser 1996]. The Markup Stage uses the GNU regular expression package to perform regular expression matching. [Hargreaves 1992]. Its regular expression language is based upon the POSIX regular
expression language specification. For each regular expression, the regular expression parser identifies the leftmost and longest string which matches the pattern described by the regular expression.

As one brief example of a regular expression, consider the tag which describes a five-digit telephone extension. The tag definition for a five-digit telephone extension is:

```plaintext
# The definition for a five digit telephone extension.
extension
x[0-9][-][0-9]{4}
EXPORTED
```

Figure 6. The tag definition for the five-digit telephone extension.

The extension “x5-1212” would match this regular expression, however “x1212” would not. The later might be identified by a tag which describes a four-digit telephone extension. In the definition the expression “[0-9]” represents the single choice of any one-place digit. The “{4}” operator following the second “[0-9]” means to chose four digits between 0 and 9.

A large number of tags can be defined in terms of just regular expressions. The Jabber system uses only regular expressions to describe types of text. There are two limitations which arise from using only regular expressions. First, if a tag contains a number of identifiable “parts”, the regular expression for each part must explicitly be present in the tag’s definition. If a number of tags contain the same “parts”, the definitions become repetitious. Also, if the definition of such a “part” changes, it must explicitly change in all tags which describe that “part.” Second, tags are often not described by a regular expression, but by a list of words. For example, a `propername` tag is defined by the words which exist in some lexicon (i.e. dictionary). It it possible to list each of these words inside the tag’s definition as a regular expression, each separated by an OR (“|”) operator. However, this method to construct a list of words is cumbersome. The following subsections enhance the toolbox which authors use to define new tags. The two problems associated with regular expressions mentioned in this paragraph are solved below.
2.5.4 Defining tags using other tags

As mentioned above, markup experts can be thought of in terms of objects in an OOPS. One feature of objects is a new type of object may inherit or borrow the functionality of pre-existing objects. Similarly, a markup expert may borrow the knowledge of another expert. More specifically, not only may a tag within an expert be described in terms of regular expressions, it also may be described in terms of other tags. In that way, tags not only serve to identify self-contained portions of text, they may also be used as building blocks to describe larger constructs within text. In order to use a tag within another tag’s definition, the name of the tag is enclosed in angle brackets (‘<’ and ‘>’) and used in the new tag’s definition.

Consider a tag which describes a common, local telephone number. A telephone number is broken into two parts: the exchange and the number. The phone tag is described in terms of an exchange tag below. In this instance, the exchange tag does not make sense by itself, it only makes sense within the context of a telephone number. (Incidentally, this is the reason why the exchange tag is not exported.) Exported versus nonexported tags are discussed later. Tags included in other tags’ definitions may either be exported or nonexported.

```plaintext
# The tag definition for a common telephone tag
exchange
[0-9]{3}[ -]+ NONEXPORTED

phone
<exchange>[0-9]{4} EXPORTED
```

Figure 7. The telephone tag in terms of the exchange tag.

The definition of an exchange is simply a three digit number, “[0-9]{3}”. A telephone number is composed of an exchange followed by a hyphen, followed by a four digit number: “<exchange>-[0-9]{4}”. The following is an example of text marked-up with a phone
A tag need not be defined within the same expert to be used by another tag. If more than one tag exists with the same name, then the first tag found is used. There are several advantages to defining tags using other tag definitions. First, each tag definition becomes more readable. It is easier to decipher the meaning of a list of tags without deciphering a complicated regular expression. In a similar vein, when an author creates a new tag, he/she does not have to worry about the regular expressions underlying the tags included in the new definition. This mechanism allows a database of abstract text objects to be created, and the system becomes more easily extensible and powerful as a result. Underlying tags can be improved in the accuracy of their descriptions, and all tags which use the underlying tags improve automatically as a result. Also, a complicated tag which is marked-up using its component subtags turns out to be much easier to process, as becomes apparent during the discussion of the Process Stage.

2.5.5 Defining tags using lexicons

A lexicon is merely a collection of words which have some alternative representation associated with them. For example, lexicons in text to speech systems contain hard-to-pronounce words and their corresponding phonetic spelling. In a number of instances in the Markup Stage, it is easier to provide a list of words to describe a tag rather than a regular expression. These lists of words can be described by a regular expression--each word is separated by the OR ("|") operator. However, tag definitions as described by the previous subsections become cumbersome if many of the tags are described in this manner. Lexicons, therefore, are a mechanism primarily of convenience. A tag definition may now simply refer to this lexicon, rather than explicitly listing all of the lexicon entries in the tag’s definition.

A tag is defined in terms of entries in a lexicon by simply using the lexicon tag, "<lexicon>" as the tag’s definition. If a tag refers to a lexicon, then its definition must only con-
tain the lexicon tag, it cannot be defined in terms of other tags or regular expressions. This
does not prevent the tag itself from being used in other tags’ definitions. The tag which
describes two-letter U.S. state abbreviations is defined in terms of a lexicon. Its tag defini-
tion is below.

```
# The tag definition for the two letter US state abbreviation
state-2
<lexicon>
EXPORTED
```

Figure 8. The definition of the two-letter U.S. state abbreviation in terms of an external lexicon.

Each markup expert may have one associated lexicon. Each lexicon is divided into
sections. Each section of the lexicon corresponds to a single tag. The two-letter U.S. state
abbreviation tag, `state-2`, refers to that section within its expert’s lexicon. In this way, there
can be more than one tag within an expert which refers to the lexicon. Each of these tags
may refer to a separate and distinct section of the lexicon. For example, the `state-4`
tag of
four-letter U.S. state abbreviations has its own section in the same lexicon. A lexicon sec-
tion begins with the `.SECTION` flag and ends with the `.ENDSECTION` flag. Each lex-
icon entry contains three fields: the entry’s word, its translation type, and its translation. A
word’s translation is a synthesizer-independent representation of its correct pronunciation.
The translation type can either be `"^verbatim"`, `"^spell"`, or `"^phonetic"`, describing what
kind of translation is provided. Since the Markup Stage is only concerned with identifying
special text items, it is only concerned with the first column of a lexicon entry. A portion
of the `state-2` lexicon section is provided below. Lexicons are found under the same direc-
tory as the markup experts themselves, and are always named `<expert>.lexicon`, where
<expert> is the name of the markup expert.

<table>
<thead>
<tr>
<th># Two letter state abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTION state-2</td>
</tr>
<tr>
<td>AK ^verbatim alaska</td>
</tr>
<tr>
<td>AL ^verbatim alabama</td>
</tr>
<tr>
<td>AR ^verbatim arkansas</td>
</tr>
<tr>
<td>AZ ^verbatim arizona</td>
</tr>
<tr>
<td>CA ^verbatim california</td>
</tr>
<tr>
<td>CO ^verbatim colorado</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>. ENDSECTION ^xxxxxxxxxx x</td>
</tr>
</tbody>
</table>

Figure 9. An example of a lexicon section for the two-letter U.S. state abbreviation.

Lexicon tags are matched with the input text in a different way than regular expressions. When searching for a tag within text which uses a lexicon, the text is first searched as-is. An entire second pass of the text is performed immediately following the first pass. The input text is converted to all lowercase for this second pass. Suppose a proper name like Jim is listed in a lexicon as “jim”. Input text of the format “Jim”, “JIM”, or “jim” will match this lexicon entry. In this case, the second pass through the lexicon performs a case-insensitive check without having to list all three: “jim”, “Jim”, and “JIM” in the lexicon. Now suppose the lexicon entry is for some reason, “JIM”. Only input text which looks like “JIM” will match this lexicon entry. “Jim” and “jim” will not match the “JIM” lexicon entry. To summarize, the special way in which text is searched with a lexicon entry provides flexibility in allowing words to exist upper-cased without listing all combinations of capitalizations. It also permits words to match only if they display the required capitalization.

2.5.6 Grouping and Operators

A toolkit for describing different types of text has been presented including regular expressions, tags, and lexicons. There are a fair number of English constructs which require a still more powerful toolbox. For example, consider the email tag, which
describes an electronic mail address. Suppose the following building-block tags exist: username, atsign, netname, and period.

```plaintext
# Building blocks for the email tag
username
[a-zA-Z]+
NONEXPORTED

netname
[a-zA-Z]+
NONEXPORTED

atsign
\@
NONEXPORTED

period
\.
NONEXPORTED
```

Figure 10. Building block tags for the definition of the email tag.

One possible construction of the email tag is `<username><atsign><netname><period><netname>`. This tag matches electronic mail addresses of the form: hordack@mit.edu or jon@gte.net. It does not identify the more complex electronic mail addresses such as hordack@media.mit.edu. Ideally, the definition of the email tag should indicate that the construct `<netname><period>` may be repeated any number of times. This is accomplished by applying a repetition operator to that part of the email tag definition.

Before a repetition operator is applied to a certain portion of a tag’s definition, that portion must be grouped together. Surrounding parts of a tag’s definition with parentheses indicates a grouping. In the email tag example above, the new expression with the proper grouping looks like: `<username><atsign>(<netname><period>)<netname>`. Both tags and regular expressions may be grouped together. Supposing the period tag did not exist, the grouping would look like: `(netname)\)`, where “\.” is actually a regular expression.

Once the desired part of a tag’s definition is grouped, one of three repetition operators may be applied to that grouping. An operator is applied to a group by placing the symbol
for the operator immediately following the right parenthesis of the grouping. The three
operators are: ‘*’ which matches zero or more instances of the group, ‘+’ which matches
one or more instances of the group, and ‘?’ which matches zero or one instances of the
group. These operators possess the same definition as they do for normal regular expres-
sions. [Hargreaves 1992] has a detailed discussion. The definition for the email tag is now
complete: <username><atsign>(<netname><period>)<netname>. At least one
instance of the <netname><period> pattern is matched.

Before concluding the discussion on the building blocks of tag definitions, one techni-
cality is left unmentioned. The Markup experts impart special meanings on several sym-
bols. Specially, the “<”, “>”, “(“, “)””, “+”, “?” and “*” symbols possess special meanings
in tag definitions. In order to use these characters verbatim, they must be preceded with an
escape character. For example, suppose we want to match an actual left parenthesis in a
regular expression. The expression must be “\(“, otherwise the left parenthesis is inter-
preted as the start of a new grouping. Therefore, if one of the above symbols is preceded
by the backslash (“\”), it is not interpreted specially within a tag definition.

2.5.7 Exported and Nonexported tags within an expert

The Markup Stage encourages the definition of tags in terms of other tags. Organizing
the definition of tags in this hierarchical manner simplifies the Process Stage. There are
some tags, however, that are internal tags. These tags only make sense in the context of its
containing tag. Consequently, the Markup Stage does not want to identify self-contained
text items which are described by this tag, because the markup would be incorrect and
have no meaning. One example mentioned previously, the exchange tag, only makes
sense within the phone number tag. Recall that the exchange tag is defined as “[0-9]{3}”.
It would not make sense for all three digit numbers not labelled as anything else to be
labelled as exchanges. While this incorrect markup may be harmless in this instance, it
may cause text to be processed incorrectly, and therefore spoken incorrectly, in other
instances.

To solve this small hitch, some tags are declared as exported and some are declared as nonexported. The third line of a tag's definition reflects whether or not a tag is exported. Both exported and nonexported tags may be used in the definitions of other tags regardless of which experts they are in. Exported tags, however, are tags which the Markup Stage tries to identify on their own. The Markup Stage does not attempt to find instances of nonexported tags by themselves in the input text. The exchange tag, therefore, gets a nonexported designation, while the phone tag gets an exported designation.

2.5.8 Summary of the Markup Stage

At the beginning of this section, the Markup Stage is described as a "collection of experts". Each expert possesses knowledge about a certain domain, and the appearance of different types of text items within that domain. Experts are consulted the order in which they appear within a configuration file, as previously described. Within each expert, the Markup Stage searches for text which match the description of tags within the expert. The tags are checked in the order in which they appear inside the expert. Tags are defined in terms of regular expressions, lexicons, or other tags. Only those tags which are exported are searched for.

Instead of a "collection of experts", the Markup Stage is more accurately described as a "web of information." Entry points into this web are represented by an exported tag. As text is searched to match the description of a tag, the Markup Stage searches through the web to find knowledge about constituent tags. In this way, knowledge is decentralized and distributed throughout this web. More precisely, this "web" is a directed, acyclic graph. Since no tag may indirectly (or directly) refer to itself, the graph is acyclic. The entry points into this graph are exactly the set of tags which are exported. When matching a tag, the graph is searched in a depth first pattern according to the tag's definition. Certain nodes in the graph may be repeated if one of the repetition operators is used. The follow-
ing is a pictorial representation a complete view of the Markup Stage with some sample
tags and experts. The shaded nodes represent entry points (exported nodes) into the graph
in the order of their heights.

![Diagram of Markup Stage]

Figure 11. A Graphical Representation of the Markup Stage

2.6 The Process Stage of the TTSML system

The *Process Stage* of the TTSML system accepts marked-up text as input and converts
the marked-up text into a form which is read correctly by any text to speech synthesizer.
Text is marked-up using TTSML by either applications or the Markup Stage. Once text
leaves the Markup Stage, no more analysis is performed on the text. The Process Stage is
concerned only with transforming the marked-up text into some synthesizer independent
representation which is read correctly. Each tag in TTSML possesses a unique definition,
so the Process Stage is not concerned with interpreting the meaning of marked-up text.
The process of transforming text into a synthesizer-independent form may involve one or
several of the following actions: replacing a word or phrase with its lexicon meaning
(which may be a phonetic translation), expanding punctuation into words, expanding num-
bers in words, inserting spaces between letters, or removing extra punctuation.
2.6.1 What is a Process Expert?

A Process Expert, much like a Markup Expert, possesses knowledge about a particular domain: electronic mail addresses, telephone numbers, etc. Each Markup Expert corresponds to a Process Expert. Each Process Expert contains the knowledge of how to convert text labelled with a certain tag into a synthesizer-independent representation. The example below uses the phrase *Bob Dole, a* `<political> R-<state-2> KS</state-2> </political>` as input to the Process Stage, where both the *political* and *state* experts are asked to convert the marked-up text into a form which represents its correct speech presentation. The *political* tag is used for political designations in the United States. In this case, the process experts consult lexicons, expand words, and remove punctuation to transform the phrase into *Bob Dole, a republican from Kansas*. Although the diagram shows the tags still attached to the fully processed tags, they are remove before the final version of the text is produced.

![Figure 12. Overview of the Process Stage](image)

2.6.2 The Processing of Marked-up Text

As the example above illustrates, "marked-up" text contains properly nested TTSML tags. Of course, applications may pass plain text into the Process Stage with no effect. The Process Stage first parses the marked-up text to check that it is properly nested. The parsed text is placed in a tree data structure. While describing the implementation of the TTSML
system is not a focus of this thesis, it is useful to think of marked-up text as a tree in the context of the Process Stage. The diagram below illustrates the example of marked-up text above in tree form. Each internal node in the graph corresponds to some tag, where the text associated with its subtree is labelled by that tag. Only leaf nodes correspond to the actual input text. The tree structure naturally represents the nested and hierarchical structure of TTSML marked-up text. Internal nodes, which correspond to tags, are shaded. Leaf nodes, which correspond to text, are represented by solid white nodes. The head node in the tree, which may or may not have a tag is solid black. In this case, the head node does not have a tag.

![Tree representation of Marked-up Text](image)

**Figure 13. Tree representation of Marked-up Text**

The Process Stage transforms marked-up text represented as a tree as follows. The Process Stage visits each node in a depth-first manner. The reader may consult [Cormen 1990] for an introduction to trees and depth-first searches. When the search encounters an internal node whose children are only leaf nodes or internal nodes which already have been processed, it passes the entire subtree of that node to the Process Expert knowledgeable about that tag. The most nested tags are processed first. The Process Stage need not pass any additional information beyond the subtree for the node to the process expert. Since each tag is unique has a unique, well-defined presentation, only the text which is tagged is needed. The numbers above the internal nodes indicate the order in which their corresponding subtrees are passed to process experts. The text corresponding to the state-
2 tag is processed first, the text corresponding to the political tag is processed second.

2.6.3 The Design of a Process Expert

A process expert is a UNIX shared object with a C-language interface. Each expert exports the same interface, as described below. Making each process expert a shared object allows new experts to be created, compiled and integrated into the TTSML system without any other recompilation. Also, changes to an expert may be seen during run-time. Complicated manipulations are required during the processing stage which require the full capabilities of a general programming language such as C. Each processing expert has at its disposal a number of utility functions, such as the ability to search lexicons. Although processing experts have the entire power of C available to it, it is generally more difficult to author a process expert than a markup expert.

Each process expert must export the following two C-language functions: querytags() and process(). The Process Stage invokes these function calls in order to communicate with the process expert and query its knowledge. The querytags() function informs the Process Stage of the tags supported by that expert. The process() function processes input text passed to it. The process() function must be capable of processing all tags which it declares via the querytags() function. Authoring a new process expert involves implementing these two functions. The remainder of this section, therefore, is devoted to describing these functions in detail and the utilities available to them.

2.6.4 The querytags() function

The querytags() function has the following prototype:

```c
void querytags(llist *ll);
```

Figure 14. The function prototype for querytags().
Its single argument is an empty linked list. The `querytags()` function adds to this list all tags which are supported by the process expert. The linked list utility is provided with the TTSML system. The `querytags()` function for the state process expert is given below. The `LlInsertAtHead()` function inserts a new element at the head of the linked list. The state process expert supports two tags: `state-2` and `state-4`.

```c
void querytags(llist *ll)
{
    LlInsertAtHead(ll, strdup("state-2"));
    LlInsertAtHead(ll, strdup("state-4"));
}
```

**Figure 15. The querytags() function for the state expert.**

### 2.6.5 The `process()` Function

The `process()` function performs the processing of marked-up text. The Process Stage passes the `process()` function a subtree. This subtree's tag is one which is supported by the process expert. The prototype of the `process()` function is given below.

```c
void process(LexiconManager *lm, TtsmlNode *node);
```

**Figure 16. The function prototype of process().**

The first argument, `lm`, is the *Lexicon Manager*. The Lexicon Manager is an example of a utility provided to the process experts. Process experts use the lexicon manager to access entries inside a lexicon. For tags defined in terms of a lexicon, such as `state-2`, the state process expert queries the `state-2` section of the state lexicon for the translation of the matching entry. The second argument, `node`, is subtree corresponding to the tagged text. The *TtsmlTree* abstraction is another utility provided to process experts. The *TtsmlNode* structure represent a node in this tree, and there are utility functions to traverse this tree available to the process expert. The process expert modifies this subtree directly when pro-

---

1. Barry Arons developed the linked list utility.
2.6.6 An Excerpt from an example process expert

The best means to illustrate the construction of a process expert is by example. This example comes from the state expert, particularly the sections which process the state-2 tag. This excerpt exemplifies the use of the LexiconManager to lookup and return the translation for an entry and the navigation of a subtree. The TtsmlFirstChild() function returns the first child of a node. The TtsmlNextChild() function returns the child of a node following the child provided in the function call. The LEXFetchLexicon() function obtains the specified state lexicon. The text tagged by the state-2 tag is looked-up in the state-2 section of the lexicon. The original text is replaced with the value returned by the lookup.

```c
void querytags(llist *ll)
{
    LlInsertAtHead(ll, strdup("state-2"));
}

void process(LexiconManager *lm, TtsmlNode *node)
{
    char *tag = node->text;
    if (strcmp(tag, "state-2") == 0) {
        Lexicon *lexicon = LEXFetchLexicon(lm, "state");
        TtsmlNode *child = TtsmlFirstChild(node);
        char *text = child->text;
        char *value;
        if ((value = LEXLookup(lexicon, tag, text)) == NULL) {
            return;
        }
        free((char *)child->text);
        child->text = strdup(value);
    }
}
```

Figure 17. Portions of the state process expert.
Chapter 3  The Speech Manager

This chapter describes the Speech Manager in detail. The Speech Manager addresses the second problem presented in Chapter 1: Introduction. Each text to speech synthesizer exports its own interface. The Speech Manager specifies a uniform API (Application Programmer's Interface) for text to speech synthesis. It also provides a mechanism to integrate any number of third-party speech synthesizers. The Speech Manager permits applications to use any one of these third-party synthesizers through this uniform API. Applications may switch between synthesizers during run time.

This chapter is organized as follows. Section 3.1 briefly describes the commercial text to speech synthesizers used in this thesis. Section 3.2 discusses related work in speech synthesis API’s. Section 3.3 presents the primary components of the Speech Manager and its overall architecture. Sections 3.4, 3.5, and 3.6 describe the components of the Speech Manager in detail. Section 3.4 examines the Speech Library and the uniform speech API. Section 3.5 discusses the synthesizer-dependent Device Drivers and Section 3.6 describes the Audio Modules.
3.1. Overview of Some Commercial Text to Speech Synthesizers

This section describes various text to speech synthesizers used in this thesis. This brief discussion provides a very informal survey of several of the most popular commercially available text to speech synthesizers. The differences between the synthesizers illuminate the need for a single speech synthesis API and provides the basis for a more in depth future discussion about the specific differences between synthesizers.

3.1.1 Digital's DECTalk

Digital Equipment Corporation's DECTalk [DECTALK 1987] is one of the earliest and most common text to speech synthesizers. DECTalk comes both as a hardware peripheral and as a software-only library. The DECTalk available for this thesis is the hardware peripheral, which includes a software interface library to the hardware.

The DECTalk hardware integrates well with telephony. It interfaces to an analog telephone line. It performs basic call control, such as detecting incoming calls and answering them. DECTalk also detects DTMF tones (touch-tone) during active telephone calls. The DECTalk hardware outputs synthesized audio exclusively to the analog telephone line or line output ports. It allows applications to use one of a number of distinct voices including male, female, and child voices. It permits applications to set the speaking rate as well. Although DECTalk facilitates the development of touch-tone and speech-based telephony applications, its tight integration with telephony goes beyond that of standard text to speech synthesizers. The DECTalk interface library to the hardware is asynchronous. DECTalk supports index markers and uses its own Arpabet-based phonetic character set. Both index markers and phonetic character sets are discussed later.

3.1.2 Entropic's TrueTalk

Entropic's TrueTalk text to speech synthesizer [TRUETALK 1995] is based upon tech-
nology developed at AT&T Bell Laboratories. The TrueTalk synthesizer provides a stand-alone server executable and an asynchronous interprocess communication library which issues commands to the server. TrueTalk directs the audio output back to the application upon request in many different audio encodings. TrueTalk has its own single-character phonetic character set based upon Arpabet as well as a rich language to control prosody based upon the Pierrehumbert intonational model [Hirschberg 1986][Pierrehumbert 1980]. It returns synthesized audio data via asynchronous callbacks. However, applications must periodically query the server to obtain synthesized audio data. TrueTalk indicates when it has completed synthesizing text and supports index markers.

3.1.3 British Telecom's Laureate

British Telecom’s Laureate [Ashworth 1995] uses concatenative speech synthesis, rather than formant based speech synthesis (or synthesis by rule). Concatenative synthesis combines previously recorded snippets of human spoken speech. All other synthesizers used in this thesis are formant synthesizers. Formant synthesizers model the human vocal tract and control a number of parameters in their model. Laureate is a synchronous library with a single “speak” command which returns audio in many different audio encodings. The most distinctive feature of Laureate (at least to American English speakers) is its British sounding voice. Because Laureate provides only a synchronous interface, an asynchronous interface is achieved by spawning Laureate into its own process.

3.1.4 Rsynth

The Rsynth text to speech synthesizer [Ing-Simmons 1995] is based on the Klatt [Klatt 1980] synthesizer and is unique is one way--it is in the public domain. The quality of the speech output for the Rsynth synthesizer is inadequate for commercial applications. However, it provides a readily-available example of a speech synthesizer which has a synchronous, library-based interface. It is this reason why this thesis takes a slight interest in
Rsynth. It returns both the phonetic representation of text and the corresponding synthesized audio. It permits applications to provide an exception dictionary with phonetically spelled words based on an ASCII representation of IPA [IPA 1996].

3.2. Related Work to Speech Synthesis Platforms

This section discusses related work to the Speech Manager. Each of these related projects provide a uniform API for text to speech synthesis and an integration mechanism for multiple third-party text to speech synthesizers. These descriptions highlight important issues which are either not mentioned in these works, or not considered by the Speech Manager.

3.2.1 Sun Microsystem's SunSpeak

SunSpeak [Gray 1995] provides a uniform C++ API for text to speech synthesis. It also provides an SPI (System Programmer’s Interface) to integrate new text to speech synthesizers into SunSpeak. Applications instantiate instances of the SpeakSynthesizer class. Its methods provide a uniform interface to text to speech synthesis. Its virtual methods are overridden by the application for event notification. The SunSpeak library is meant to integrate well with Sun’s XTL Teleservices Platform. [XTL 1994] Towards this end, SunSpeak accepts a file descriptor to which it outputs audio. When using XTL in conjunction with SunSpeak, this file descriptor corresponds to the active B-channel of the ISDN telephone line. The synthesizer must be idle before applications may send next text. An error is generated if new text arrives while old text is still being synthesized. New text to speech synthesizers are integrated into SunSpeak by subclassing the SpeakEngine class. The various methods are overridden and implemented for the specific speech synthesizer.

3.2.2 Microsoft’s SAPI (Speech API)

SAPI is Microsoft’s synthesizer independent API for text to speech synthesis. [SAPI 1995] Much of the following analysis is provided by Jamison Gray of Sun Microsystems
Computer Corporation through personal communications. [Gray 1996] The Microsoft SAPI provides both an API for application developers and an SPI for integrating new text to speech synthesizers. It is based upon OLE, Microsoft’s object oriented development framework. Applications create instances of OLE objects and invoke their methods. SAPI has methods to control the flow of synthesized speech such as pause and resume. SAPI queues all incoming text until all previous text has finished being synthesized. SAPI supports the International Phonetic Alphabet (IPA) [IPA 1996] as its phonetic character set, using UNICODE instead of the more common ASCII representation for text. The UNICODE representation of text allows for easier internationalization.

Most notably, Microsoft’s SAPI allows synthesized speech to be sent to a number of “audio destinations.” These audio destinations are uniform interfaces to different kinds of audio output devices. Microsoft’s audio destinations are similar to Audio Modules presented in this thesis. Also present in SAPI is “tagged text,” a number of embeddable escape sequences which allows applications control of such attributes as speaking rate, volume, and some basic prosodic controls. Interestingly, applications may request a synthesizer via a set of attributes, such as “language, dialect, ..., speaking style, and age.” [Gray 1995] SAPI attempts to pick a text to speech synthesizer which most accurately matches the selected attributes.

### 3.3. The Speech Manager and Uniform Speech API

The Speech Manager has three component parts: the Speech Library, the synthesizer-dependent Device Drivers, and the Audio Modules. Figure 18 depicts the architecture of the Speech Manager with respect to these three components. A brief overview of each module is given below. Future sections describe each component in detail.

#### 3.3.1 The Speech Library

Applications link the Speech Library at compile time. It is the only component of the
Speech Manager to which applications directly interact. The Speech Library presents the uniform C language speech synthesis interface (API) of the Speech Manager to applications. The Speech Library API is asynchronous, composed of *request* and *events*. Requests are messages sent from the application to the Speech Manager, where events are messages sent from the Speech Manager to the application. Applications register C language function callbacks which are called when the Speech Manager sends an event to the application. The API provided by the Speech Library contains a “main loop” function, which is called only once by applications. This “main loop” function never returns and handles the delivery of requests and the dispatch of events. The asynchronous programming model employed by the Speech Manager is similar to the X Windows X Toolkit Instrinsics. [Nye 1990]

The Speech Library API allows applications to select which particular text to speech synthesizer to use. Its interface includes requests to synthesize text into audio and control the output of this audio. The Speech Library API also provides applications with accurate status and information events, such as when the synthesis of text has completed. The Speech Library communicates with the synthesizer-dependent device drivers, sometimes across process spaces. The actual Speech Library API is presented in Section 3.4. At that time, the complete functionality of the Speech Manager becomes apparent.

### 3.3.2 The Synthesizer-Dependent Device Drivers

Third-party text to speech synthesizers are integrated into the Speech Manager via *device drivers*. Each text to speech synthesizer must have its own device driver. Once a device driver is written for a text to speech synthesizer, applications use it via the Speech Library API. Each device driver may either be a stand-alone executable which communicates with the Speech Library via RPC [Arons 1992] or a shared object which communicates with the Speech Library via a C language interface. This receives elaboration in Section 3.5. Device drivers implement the primary functionality of the Speech Manager.
They provide an interface between the uniform Speech Library API and each speech synthesizer-dependent API. Device drivers are responsible for translating requests (C language calls) from the Speech Library into synthesizer-dependent API function calls. They also route synthesized audio data to the application or to the Audio Modules.

Section 3.5 describes device drivers in depth, including the means by which a systems programmer integrates a new text to speech synthesizer into the Speech Manager.

3.3.3 The Audio Modules

Audio Modules provide the means by which applications route the synthesized audio data to a number of different types of destinations. These destinations include an audio file, an audio device (e.g. workstation’s speaker), a file descriptor, or the application itself. Also, Audio Modules provide a means by which applications select the mechanism which handles the output of audio. Different audio output mechanisms (i.e. different audio servers or libraries to play audio data) require their own audio module. Applications select at run time which audio module to use. Audio Modules are dynamic shared objects which communicate with device drivers via C language functions.

Section 3.6 describes audio modules in detail, including the means by which a systems programmer writes new audio modules and integrates them into the Speech Manager.

3.3.4 Architectural Diagram of the Speech Manager

The diagram below shows how the three components fit together to form the Speech Manager. The solid lines show the flow of textual data, while the dashed lines show the flow of audio. The application uses speech synthesizer ‘A’ via the uniform speech synthesis API. Its requests are sent to the device driver for synthesizer ‘A’ where they are translated into synthesizer ‘A’-dependent API calls. The device driver may either be a stand-alone executable or a shared object, as discussed in Section 3.5. The device driver obtains the synthesized audio, if possible, and routes it to either the application or an audio mod-
ule. Because the application wants to use AudioFile [Levergood 1993] to play audio data, it has selected its corresponding audio module.

![Diagram of the Architecture of the Speech Manager](image)

**Figure 18. Overview of the Architecture of the Speech Manager**

### 3.4. The Speech Library and Speech Manager API

This section discusses the Speech Library in detail, and in particular, the API of the Speech Manager which is presented to applications via the Speech Library. The Speech Library presents to applications a set of C language function calls which permit the application to send requests to the Speech Manager. It also allows applications to register C language function callback which are called when the Speech Manager sends events to the application.

The Speech Library communicates only with the device driver currently in use. If the device driver is a stand-alone executable, the communication mechanism is a UNIX socket-based remote procedure call mechanism (RPC) named the Byte Stream Manager/
Socket Manager (BSM/SM for short). [Arons 1992]. If the device driver is a shared object, the Speech Library communicates with the device driver via C language function calls and using dlopen() and dlsym(). Naturally, device drivers have their own internal interface for communication with the Speech Library. In order the use the Speech Manager API, applications must simply link the Speech Library during compile time. The following sections describe the Speech Manager API in detail. Each C language API function call begins with the speech_ prefix.

3.4.1 Initializing the Speech Library and Speech Manager

The speech_initialize() function initializes the Speech Library and Speech Manager. It must be called before any other invocation to the Speech Manager API. The speech_initialize() function takes no arguments and returns SPEECH_SUCCESS upon success, otherwise it returns SPEECH_ERROR.

3.4.2 Selecting the Third-Party Text to Speech Synthesizer at Run Time

Before applications synthesize text, they must select a particular third-party text to speech synthesizer to use. This is accomplished via the speech_engine() function call. Each text to speech synthesizer available in the Speech Manager is listed in the configuration file named drivers.conf. An example drivers.conf file is given below. The first column of each entry in the configuration file lists the name by which each text to speech synthesizer is referred to by applications. For example, Digital's DECtalk is called "Dec-talk" while Entropic's TrueTalk is called "Truetalk." The second column gives the full path name of the device driver's location. The third column describes whether the device
driver is implemented as a stand-alone executable or a shared object.

<table>
<thead>
<tr>
<th># The drivers.conf Configuration File</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dectalk /speech/tts/bin/tts.dectalk.so object</td>
<td></td>
</tr>
<tr>
<td>Truetalk /speech/tts/bin/tts.entrpic.so object</td>
<td></td>
</tr>
<tr>
<td>Rsynth /speech/tts/bin/tts.rsynth.so executable</td>
<td></td>
</tr>
<tr>
<td>Laureate /speech/tts/bin/tts.laureate.so executable</td>
<td></td>
</tr>
</tbody>
</table>

Figure 19. Example of the driver.conf configuration file.

The speech_engine() function takes a single argument, the name of the synthesizer desired as listed in the drivers.conf configuration file. This function returns SPEECH_SUCCESS if the device driver for the selected synthesizer starts successfully, otherwise it returns SPEECH_ERROR. Applications select new text to speech synthesizers at any time. However, the old synthesizer must have completed servicing all outstanding requests from the application. An application must select a synthesis engine with speech_engine() before any further call to the Speech Manager API.

3.4.3 Selecting the Audio Module at Run Time

Similar to selecting a synthesizer, applications select which audio module to use. Another configuration file, audio.conf, lists all of the audio modules available in the Speech Manager. Each corresponds to a different audio handling mechanism. Like device drivers, each audio module has a name. Application use the speech_audio_module() function call to set the audio module. This call takes a single argument: the name of the audio module. If the application selects no audio module, a default is used. Applications may switch between audio modules, provided the Speech Manager is currently idle.

3.4.4 Sending Text to the Speech Manager for Synthesis

Applications use the speech_speak() function to send text to the Speech Manager. This function takes a single argument, a pointer to the input text buffer. If it is able to
speak the text, `speech_speak()` returns `SPEECH_SUCCESS`. Otherwise, it returns `SPEECH_ERROR`. If the return value is `SPEECH_ERROR`, applications must issue a second `speech_speak()` function for the same text. The behavior of `speech_speak()` is "all or nothing"—either all of the text is accepted and synthesized or none of it is accepted. The `speech_speak()` function returns an error because of a number of reasons. One such reason is if the internal text queueing buffer of the Speech Manager is full, or if the input text ends along the incorrect boundary. The following section on the queueing model of the Speech Manager describes what the Speech Manager expects in terms of input text buffers.

### 3.4.5 The Speech Library Queueing Model for Text

Different text to speech synthesizers implement different "queueing" models. Queuing models dictate how much input text data is accepted at once, on what boundaries the input text must lie (word, phrase, or sentence), and whether or not new input text overwrites old, yet-unspoken text. Text to speech platforms must define the specifics of the model it implements for applications. It is extremely important for uniform text to speech API's to present a well-defined text to speech queueing model, so applications expect some constant, well-defined behavior.

The Speech Manager queues all new, accepted input text provided by the application. Therefore, new input text waits until the Speech Manager finishes speaking old input text (unless explicitly flushed by the application). This behavior contrasts with another common behavior where new input text interrupts any old input text still being spoken. The
Each input text buffer must lie along sentence boundaries. If the Speech Manager detects that an input buffer does not lie along a sentence boundary, it appends a period onto the end of the input text. The Speech Manager will queue up to 50 Kilobytes of input text. If new input text causes this limit to be exceed, the new input text buffer is rejected in its entirety and an error is returned.

The Speech Manager also implements a “speak as soon as possible” policy. For example, suppose buffer A is sent to the Speech Manager for synthesis at time 0. Buffer A takes 6 seconds to speak. At time 3, the application sends buffer B to the Speech Manager. With the “speak as soon as possible” policy, buffer B is spoken immediately after buffer A completes. An alternative approach is for the Speech Manager to pause 3 seconds before buffer B is spoken after the completion of A. In other words, the relative timing of when successive speech_text() commands are invoked is irrelevant to the Speech Manager. They have no bearing upon the relative timing of when the successive input text buffers are spoken. In order to ensure the relative timing between spoken phrases, applications must insert pause commands into the text stream. The pause instruction is discussed later in section 3.4.9. The diagram below illustrates these two possible policies; the Speech Manager
implement the first.

![Diagram of text buffers and timing]

**Figure 21. Two different policies for synthesis timing**

### 3.4.6 Controlling the Flow of Synthesized Speech

Applications may temporarily stop the output of synthesized speech. They then may instruct the Speech Manager to continue synthesizing speech in exactly the spot it left off. The Speech Manager API functions `speech_pause()` and `speech_resume()` are for these purposes, respectively. Any text waiting for synthesis is not lost when `speech_pause()` is called. The Speech Manager pauses and resumes synthesis as accurately as it can. Synthesized audio is sent by the Speech Manager to an audio module which may internally queue several seconds of audio data. Therefore, the pause and resume commands depend upon the accuracy of the audio module’s own pause and resume capabilities. This issue is discussed more in depth in Section 3.6.

### 3.4.7 Interrupting the Synthesis of Speech

At times, applications interrupt the output of synthesized text. For example, applications which read long text files allow users to interrupt this output. Applications use the `speech_flush()` command to halt the speech output. It also clears all internal text buffers in the Speech Manager and attempts to flush all internal buffers of the audio mod-
ule. Once again, this depends upon the accuracy of the flushing mechanism of the audio module in use. This issue is discussed in more detail in Section 3.6.

3.4.8 Inserting Index Markers into the Text Stream

Applications want specific notification when a particular word or phrase is spoken. This information is useful in instances where a graphical interface is updated according to when certain words or phrases are spoken. One example is DECface [Waters 1995], which displays a computer-generated image of a human face which mimics the mouth movements of a human speaker.

For each place in the input text stream it wants notification, applications insert an in-stream synthesizer instruction into the data stream. In-stream synthesizer instructions are discussed generally in the next section. The in-stream instruction for index markers is “<marker=n>”, where $n$ is a positive, non-zero integer. Each time the Speech Manager “speaks” one such index marker, it notifies applications via an asynchronous notification event. Applications must tell the Speech Manager which function to call when one of these index marker events is generated. This is accomplished via the speech_index_marker_cb() function. This function takes a single argument, a pointer to the desired callback function. The callback function has the following prototype:

```c
void indexcb(int marker);
```

where `marker` is the non-zero, positive integer corresponding to the index marker just spoken. The accuracy of when the index marker notification event is delivered to the application, once again, depends upon the accuracy of the audio module in use. For more details, see Section 3.6.

3.4.9 In-Stream Synthesizer Instructions

Applications often provide information to the speech synthesizer in addition to the
plain input text. This additional information may set the value of an attribute of synthesized speech. Such attributes include the speaking rate, the speaking voice (e.g. male or female), voice pitch, or emphasis. Also, applications may instruct synthesizers to pause for some amount of time, spell out certain words, or speak a word which is spelled phonetically. All of this extra information is communicated by applications via *in-stream synthesizer instructions*. In-stream synthesizer instructions take two forms. One form is “<instruction=value>”, where *instruction* is the name of the instruction, and *value* is the value assigned its parameter. The new value takes effect immediately and lasts until its value is changed with a future instruction. The second form of instruction requires both a beginning instruction, “<instruction>”, and an ending instruction, “</instruction>”. All text contained between these two “tags” are affected by the instruction. Spelling words phonetically requires this second form of instruction, for example.

- **Speaking Rate.** The speaking rate is the number of words per minute at which the text is spoken. The typical speaking rate for humans is 180 words per minute and is the default speaking rate for the Speech Manager. To change the speaking rate, use the “<rate=n>” instruction where *n* is the new speaking rate in words per minute.

- **Speaking Voice.** The speaking voice is the type of voice used for synthesis. Often both male and female voices are provided by a text to speech synthesizer. Occasionally, a child’s or robot’s voice is available as well. The default speaking voice for the Speech Manager is a male’s voice. To change the speaking voice, use the “<voice=v>” instruction where *v* is either the strings “male”, “female”, “child”, or “robot”. If the designated voice is not supported by the text to speech synthesizer, the particular device driver attempts to make the best match at its own discretion. If it cannot produce such a match, the voice is left unchanged.

- **Speaking Pitch.** The speaking pitch is the baseline pitch for the synthesis voice. Some text to speech synthesizers do not allow the application to control both the speaking voice and the pitch. In this case, any pitch command is ignored. To change the speaking pitch, use the “<pitch=p>” command where *p* is the new pitch in hertz (Hz).

- **Pause.** The only mechanism to control the timing between the synthesis of phrases is inserting a pause instruction. The pause instruction tells the text to speech synthesizer
to keep silent for a certain number of seconds. The pause instruction look like:
“<pause=p>”, where p is the number of seconds to pause.

- **Emphasis.** At times, applications want to emphasize a certain phrase or sentence. Warning messages or alerts are examples of phrases which require emphasis. The emphasis instruction must be paired with an ending emphasis instruction. Those words contained within the beginning and ending tag are emphasized. If the synthesizer does not support emphasis, the instruction is ignored and the text is spoken normally. For example, *In this sentence, <emphasis>this phrase is emphasized</emphasis>*.

- **Phonetic Spelling.** It is common for applications to spell certain words phonetically. Proper names are common examples. The phonetic instruction has a beginning and ending tag and instructs the Speech Manager to interpret the enclosed text as phonetic characters. A text to speech synthesizer which does not support some way of providing phonetic input is rare. However, in this rare case, the enclosed text is ignored. The beginning and ending tags are “<phonetic>” and “</phonetic>”. The actual phonetic character set understood by the Speech Manager is left unanswered until the next subsection. As mentioned in the introduction, each text to speech synthesizer often has its own unique phonetic character set. Finding common ground among synthesizers is not an easy task, especially if the Speech Manager supports multiple languages in addition to English. All of these issues are discussed below.

- **Spell Mode.** Often abbreviations or acronyms are properly pronounced by speaking each letter individually. The spell mode instruction allows applications to easy highlight text which is to be spelled. The spell mode has a begin tag, “<spell>” and an end tag “</spell>”. Even if a text to speech synthesizer does not support spell mode, the device driver can simply separate each letter with a whitespace, which causes the word to be spoken letter-by-letter. The enclosed text is represented using the typical ASCII English alphabet, where all punctuation is pronounced explicitly as well.

- **Synthesizer Dependent Instructions.** At times, particular applications want to make specific use of some instruction supported by a text to speech synthesizer which is not supported by most. Synthesizer dependent instructions allow applications to provide a synthesizer-dependent string which is not interpreted by the Speech Manager. The Speech Manager simply passes the string along with the other text to the synthesizer in
use. The synthesizer tag looks like: "<synth=s>"", where \( s \) is the synthesizer-dependent instruction.

### 3.4.10 Prosody and Phonetic Character Sets

Perhaps the biggest challenging facing the design of a uniform API and platform to multiple third-party text to speech synthesizers is devising a synthesizer-independent prosodic model and a synthesizer-independent phonetic character set. "Prosody" means the rhythm by which text is spoken. Each text to speech synthesizer bases its prosodic controls on a particular theory of intonation. Some have only basic controls such as word emphasis and lexical stress. Others are based upon complicated theories, such as the Pierrehumbert theory of generative intonation. \[\text{Hirschberg 1986}] [\text{Pierrehumbert 1980}]\] Unfortunately, prosodic models do not map well to one another making a synthesizer-independent prosodic model extremely difficult to define. A robust, synthesizer-independent set of prosodic controls is in fact, a thesis probably unto itself.

Designing a phonetic character set independent of any particular text to speech synthesizer is a solved problem to some extent. The International Phonetic Alphabet (IPA) \[\text{IPA 1996}]\], provides a language-independent phonetic character set used by linguists. While its relative merits and problems are not considered here, a subset of IPA exists for just the English language. This subset is often called Arpabet \[\text{Shoup 1980}]\], and is suitable for the purposes of this thesis. Arpabet is an ideal candidate for the English language since virtually all text to speech synthesizers support some ASCII representation of Arpabet. The only remaining chore is to decide upon the particular ASCII representation to use. The thesis simply selected the one used by DECTalk \[\text{DECTALK 1987}]\] since each phonetic character closely represents its sound. The Speech Manager does provide some flexibility to systems programmers to control which phonetic character set is used by the Speech Manager.
3.4.11 Specifying User Dictionaries

The Speech Manager does not robustly support user dictionaries. As discussed in Chapter 1: Introduction, user dictionaries allow applications to list a number of "exception" words and their phonetic spellings. These dictionaries are most often used for proper names or domain-specific jargon. Each text to speech synthesizer has its own file format for user dictionaries. User dictionaries were dealt with in the previous chapter of this thesis regarding markup languages for text to speech synthesis.

However, the Speech Manager API should not completely prohibit applications from providing a user dictionary to a text to speech synthesizer. The function speech_dictionary() is provided in the API for this purpose. The Speech Manager, however, does not provide format translation between different dictionary formats.

3.4.12 Directing Audio to a Particular Output

The Speech Manager allows applications to direct the synthesized speech-audio output to one of a number of destinations. These destinations are: the application itself, an audio file, an audio device, or a file descriptor. To direct audio to an output, the application simply calls a function with the required information. In order to direct the output to a file descriptor, the application calls the speech_output_fd() and passes to the Speech Manager the desired file descriptor. The speech_output_file() directs the audio to a Sun audio file. Applications supply the name of the file which is overwritten if it exists. The speech_output_device() function directs audio to a particular device, while speech_output_application() directs the audio to the application. In the case of directing output to the application, applications must supply the name of the function to be called with the synthesized audio data. The example below shows how the
speech_output_application() API function is used.

```c
void speech_cb(char *data, int len)
{
    fprintf(stderr, "Got buffer of size %d\n", len);
}

main(int argc, char *argv[])
{
    speech_output_application(speech_cb);
    speech_main_loop();
}
```

Figure 22. Directing audio output to the application

Applications route audio to only one destination. The most recent call to one of these functions determines the destination of the audio. If one of these functions is never called, the Speech Manager routes the audio data to the default audio device on the workstation. If the Speech Manager is unable to direct the audio to the designated output, the synthesized audio data is discarded.

3.4.13 Opening and Closing the Output Destination

Before synthesized audio data is sent to the designated output, applications must first open the audio destination. When applications are finished synthesizing audio to a destination, it must close the output. To open the output, applications call the speech_open_output() function; to close the output, applications call the speech_close_output() function. Both of these return SPEECH_ERROR if not successful. By requiring this explicit control over opening and closing the output destination, applications may choose when to use the output audio resource. This is particularly useful if an audio device permits only one client at a time. When the Speech Manager is idle, applications may choose to close the audio device, allowing other applications to play
audio data in the meanwhile.

3.4.14 Controlling the Quality of Synthesized Audio Data

Many text to speech synthesizers allow applications to determine the encoding and sampling rate of the audio to which speech is synthesized. For high-quality devices, applications require audio at “CD quality” which has a sampling rate of 44.1 Khz and each sample is represented by a 16-bit linear value. Over-the-telephone applications require that the audio is at “telephone quality” which has a sampling rate of 8 Khz and each sample is represented by an 8-bit mu-law value. Applications use the speech_output_format() function to set these parameters. This functions accepts three arguments: the sampling rate, the precision (8-bit vs. 16-bit), and the encoding (mu-law vs. linear). In some instances, text to speech synthesizers cannot produce synthesized speech at the desired audio quality. In that case, speech_output_format() returns SPEECH_ERROR.

3.4.15 A Synthesizer-Dependent Extension Mechanism

The Speech Manager cannot account for all functionality in each specific text to speech synthesizer. This is true if some general functionality is added to all text to speech synthesizers or if one particular synthesizer has functionality which others do not. For example, DECtalk [DECTALK 1987] interfaces directly to an analog telephone line. The application must have access to DECtalk’s analog call control capabilities. These types of capabilities are available via the Speech Manager extension mechanism. The Speech Manager API call speech_extension_message() takes a single argument, a character string. This character string is some synthesizer-specific message which is interpreted by the device driver in a synthesizer-dependent manner.
3.4.16 Summary of the Speech Manager API

The following chart summarizes the Speech Manager API. All of these functions return *SPEECH_SUCCESS* upon success and *SPEECH_ERROR* upon error, unless otherwise noted.

**Table 1: Summary of the Speech Library API**

<table>
<thead>
<tr>
<th>Function Prototypes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int speech_initialize(void);</code></td>
<td>Initializes the Speech Manager. Loads the configuration files for the device drivers and audio modules.</td>
</tr>
<tr>
<td><code>int speech_engine(char *engine);</code></td>
<td>Selects the device driver specified by <em>engine</em>. That name must appear in the <em>drivers.conf</em> configuration file.</td>
</tr>
<tr>
<td><code>int speech_drivers(char ***drivers);</code></td>
<td>Returns the names of the available device drivers in <em>drivers</em>. Also returns the number of available drivers. The Speech Library allocates the required memory.</td>
</tr>
<tr>
<td><code>int speech_modules(char ***modules);</code></td>
<td>Returns the names of the available audio modules in <em>modules</em>. Also returns the number of available audio modules. The Speech Library allocated the required memory.</td>
</tr>
<tr>
<td><code>int speech_audio_module(char *module);</code></td>
<td>Selects the audio module specified by <em>module</em>. That name must appear in the <em>audio.conf</em> configuration file.</td>
</tr>
<tr>
<td><code>int speech_close(void);</code></td>
<td>Closes the current device driver and audio module if any are active. Another device driver and audio module must be selected before the Speech Manager may be used again.</td>
</tr>
<tr>
<td><code>int speech_output_file(char *path);</code></td>
<td>Instructs the device driver to direct the audio output to a Sun audio file named <em>path</em> which it creates.</td>
</tr>
<tr>
<td><code>int speech_output_device(char *device);</code></td>
<td>Instructs the device driver to direct the audio output to a Sun audio device named <em>device</em>.</td>
</tr>
</tbody>
</table>
Table 1: Summary of the Speech Library API

<table>
<thead>
<tr>
<th>Function Prototypes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int speech_output_fd(int fd);</td>
<td>Instructs the device driver to direct the audio output to a file descriptor, fd. This file descriptor must be already opened for writing.</td>
</tr>
<tr>
<td>int speech_output_application (void (*audiocb)(char *buf, int length));</td>
<td>Instructs the device driver to direct the audio output back to the application. The application receives the audio data by providing the name of a callback function, audiocb.</td>
</tr>
<tr>
<td>int speech_output_available(void);</td>
<td>Returns SPEECH_TRUE if the device driver is able to control the destination of the audio, SPEECH_FALSE otherwise.</td>
</tr>
<tr>
<td>int speech_audio_format(int rate, int width, int encoding);</td>
<td>Instructs the device driver to obtain synthesized audio whose sampling rate is rate, bit width is width, and has encoding encoding. The instruction only takes effect if both the underlying speech synthesizer and audio module support the specified format.</td>
</tr>
<tr>
<td>int speech_open_output(void);</td>
<td>Opens the chosen output for writing. This function must be called immediately before text is sent to be synthesized.</td>
</tr>
<tr>
<td>int speech_close_output(void);</td>
<td>Closes the chosen output once speech has been completely synthesized and played.</td>
</tr>
<tr>
<td>int speech_speak(char *text);</td>
<td>Provides text to the Speech Manager which is to be synthesized and played.</td>
</tr>
<tr>
<td>int speech_done(void);</td>
<td>Returns SPEECH_TRUE if the Speech Manager is currently inactive, SPEECH_FALSE otherwise.</td>
</tr>
<tr>
<td>int speech_pause(void);</td>
<td>Temporarily pauses the output of synthesized speech.</td>
</tr>
<tr>
<td>int speech_resume(void);</td>
<td>Resume the output of previously paused synthesized speech.</td>
</tr>
<tr>
<td>int speech_flush(void);</td>
<td>Halts and flushes the output of synthesized speech. Any text waiting to be synthesized in the Speech Manager is disposed of.</td>
</tr>
</tbody>
</table>
Table 1: Summary of the Speech Library API

<table>
<thead>
<tr>
<th>Function Prototypes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>int speech_dictionary(char *dictionary);</td>
<td>Instructs the device driver to provide the dictionary named <code>dictionary</code> to the underlying speech synthesizer.</td>
</tr>
<tr>
<td>int speech_extension(char *message);</td>
<td>Provides the extension message <code>message</code> to the device driver.</td>
</tr>
<tr>
<td>void speech_main_loop(void);</td>
<td>Sits in the dispatch loop forever, this function never returns.</td>
</tr>
<tr>
<td>void speech_dispatch(void);</td>
<td>Dispatches the current events waiting to be sent to the application and allows the device driver to perform any processing it requires.</td>
</tr>
<tr>
<td>int speech_begin_cb(void (*startcb)());</td>
<td>Instructs the Speech Manager to call <code>startcb</code> whenever a start information event occurs.</td>
</tr>
<tr>
<td>int speech_end_cb(void (*endcb)());</td>
<td>Instructs the Speech Manager to call <code>endcb</code> whenever an end information event occurs.</td>
</tr>
<tr>
<td>int speech_marker_cb(void (*markercb)());</td>
<td>Instructs the Speech Manager to call <code>markercb</code> whenever an index marker event occurs.</td>
</tr>
<tr>
<td>int speech_extension_cb(void (*cb)());</td>
<td>Instructs the Speech Manager to call <code>cb</code> whenever an extension information event occurs.</td>
</tr>
</tbody>
</table>

3.5. The Synthesizer-Dependent Device Drivers

Synthesizer-dependent device drivers serve as an intermediary between the Speech Library API and the third-party text to speech synthesizers. Each speech synthesizer has its own unique device driver which translates the Speech Library API functions to the specific API function made available by the speech synthesizer. At times, this translation entails only changing the function name. Most often, however, device drivers must perform additional steps for this translation. Device drivers, themselves, export a C language interface used only by the Speech Library. In the interest of brevity, the specifications for this internal interface is absent in this discussion. Instead, this section outlines the respon-
sibilities of device drivers in detail, noting when the translation between the Speech Library API and the third-party text to speech synthesizer API's may be complex.

3.5.1 Implementing an Asynchronous Interface

The Speech Library presents an asynchronous interface applications, while different speech synthesizers presents either a synchronous or asynchronous interface. Device drivers are responsible for implementing an asynchronous interface on top of all synchronous interfaces. If the underlying speech synthesizer has an asynchronous interface, the device driver may be implemented as a shared object which is loaded by the Speech Library. If the underlying speech synthesizer has a synchronous interface, the device driver is often implemented as a stand-alone executable to achieve the desired asynchronous behavior. The Speech Library communicates with this stand-alone executable via an asynchronous, sockets-based RPC mechanism [Arons 1992]. Applications may continue with further processing while the stand-alone executable device driver communicates synchronously with the speech synthesizer.

Allowing device drivers to be implemented as shared objects is merely a performance enhancement. Often, text to speech synthesizers which have an asynchronous interface come with their own stand-alone executable server, such as Truetalk [TRUETALK 1995]. By allowing device drivers to exist as a shared objects, the synthesized audio often passes through one less process space. The type of device driver for a particular synthesizer, whether an executable or shared object, is provided in the drivers.conf configuration file, as introduction in Section 3.4.2.

3.5.2 Implementing the Text Input Model

The Speech Manager device drivers implement the text input model as described previously. Recall that the input model defines how much data is accepted by the Speech Manager, and on what boundaries the text must lie. The Speech Manager, somewhat arbi-
trary defines the upper limit on the amount of text handles at once at 50 Kilobytes. Allocating this amount of space on modern workstations is not a problem. Device drivers are intimately knowledgeable of how much data its underlying text to speech synthesizer can handle at once. Device drivers are responsible for passing to the speech synthesizer only the amount of text it can handle at once. Excess text within the 50 Kilobyte limit is queued by the device driver. Device drivers signal an error to the Speech Library (which subsequently signals an error to the application) if the internal text buffer overflows. Any new text which causes this overflow is discarded.

The device driver must also ensure that text sent to the speech synthesizer lies along the proper boundaries. The Speech Manager specifies that text passed by the application to the Speech Library must lie along sentence boundaries. This specification eases the device driver programmer’s chore since most speech synthesizers use this as an acceptable boundary as well. Nevertheless, the device driver must analyze the text and only pass text onto the speech synthesizer along the proper boundaries.

In addition to implementing the text input model, the device driver is responsible for noting errors returned from the underlying speech synthesizer. For example, if the device driver passes text to the synthesizer which returns an error, the device driver must ascertain the cause of the error and queue the text until the proper conditions exist to re-send the text for synthesis.

3.5.3 Obtaining the Synthesized Audio from the Speech Synthesizers

Perhaps the most powerful mechanism of the Speech Manager is its ability to direct the synthesized audio to a wide variety of output destinations using a number of audio platforms or handling mechanisms. Device drivers load a particular audio module as specified by the application; each audio module has the same interface. Audio modules are discussed further in the following section. In order to provide audio modules with the audio data, the device drivers themselves must obtain a copy of the audio data. Some text to
speech synthesizer do not provide this ability, such as hardware DECTalk [DECTALK 1987]. In this case, the device driver must relay this information to the Speech Library. However, most text to speech synthesizers provide the audio data through their APIs. The device drivers are responsible for performing all necessary function calls to obtain this audio data and subsequently route it to the audio module.

### 3.5.4 Returning Status and Information to the Speech Library Asynchronously

Applications require status and information notifications from the Speech Manager. Device drivers must keep on top of the activity of the underlying speech synthesizer and signal the Speech Library of any new information it obtains. There are five types of status or information events the device driver handles: when speech begins being synthesized, when speech finishes being synthesized, the occurrence of index markers, audio data to be sent to the application, and any synthesizer specific extension events.

Often, applications need to know when text is first synthesized and played and when all text has finished being synthesized and played. This information is often used to update graphical displays or play auditory cues following a synthesized message. For precise accuracy concerning these events, the device driver must also consult the particular audio module in use to find out exactly when audio has begun or finished played. If the underlying speech synthesizer supports index markers, device drivers must note when these index markers are processed.

One possible destination for audio which applications may request are themselves. When device drivers receive the audio data from the speech synthesizer they must then asynchronously provide this audio data to the Speech Library, which in turn, passes it onto the application. Finally, device drivers must catch synthesizer-specific event information, and return this information in the form of an extension information event. For example, if applications request that synthesized text be returned as a phonetic translation, this phonetic translation is returned to the Speech Library as an extension event.
3.5.5 In-Stream Synthesizer Instructions

Device drivers must parse the input text for all in-stream synthesizer instructions as specified previously in this chapter. The Speech Manager provides a utility to the device driver developer which parses the input text for these synthesizer independent instructions and dispatches to the device driver for handling. The device driver developer must implement dispatch handlers and convert in-stream instructions to synthesizer-dependent strings. Device drivers may need to convert the units of a particular in-stream instruction or, in the case of a phonetically spelled word (i.e. the <phonetic> tag), it must convert the individual phonetic characters to the phonetic characters supported by the underlying third-party text to speech synthesizer.

The difficulty in specifying a universal phonetic character set is discussed towards the beginning of this chapter. While this thesis does not present a solution to this problem, it does present a compromise. The mechanisms for translating phonetic character sets is provided to the device driver developer in the most general way possible. The device driver developer only need to complete a translation chart between the uniform phonetic character set chosen for the Speech Manager and the synthesizer-specific character set. Code provided with the Speech Manager performs a lookup in this chart automatically when translating phonetic characters. This facility allows an industrious engineer to change the uniform phonetic character set used by the Speech Manager entirely. It also takes the first steps towards a mechanism to support other language’s phonetic character sets. True, it does not define a multi-language phonetic character set, however, the Speech Manager can support any ASCII-based character set which requires only a table lookup for translation.

3.5.6 Synthesizer-dependent Extension Messages

Device drivers translate all synthesizer-dependent extension messages sent to it via the Speech Library into function calls of the underlying speech synthesizer. Device drivers must publicly define the formats of acceptable extension messages received from the
Speech Library and application. Device drivers signal an error to the Speech Library if they receive an unsupported or unrecognized extension message.

3.6. The Audio Modules

Audio modules are responsible for the playing of synthesized audio data. Because no universal platform exists for playing audio on a workstation, different workstations have their own means to play audio data. Even on the same brand of workstation, different sites may choose to use a different mechanism. One way to play audio data is to directly write to the device interface provided by the operating system. Another way is to use a high-level interface provided by an audio server such as the Media Lab’s Audio Server [Slott 1996] or Digital’s AudioFile [Levergood 1993]. Each different means to play the audio data requires its own audio module. Audio modules, therefore, allow applications to chose how to play audio data. Perhaps when a standard audio interface platform gains universal acceptance, the need for audio modules will vanish. The device drivers in the Speech Manager communicate with the audio modules. Each audio module is a shared object which communicates with the device drivers through its own internal C language interface. This interface is not presented here. Instead, the fundamental issues concerning the implementation of audio modules are discussed below.

Audio modules must be capable of playing to a specified audio device, an audio file, or a file descriptor. Audio modules must be able to notify the device drivers when audio first begins playing and when it eventually completes playing. The device drivers pass this information back to the Speech Library. Applications need the most accurate indication of these types of events. This is because the time difference between when audio is passed to the audio module and when it actually plays may be several seconds. Reporting the playing of index markers also falls under this category. Often, noting when audio actually begins playing and finishes playing is impossible for an audio module. Similarly, audio modules cannot accurately pause and resume audio. Therefore, the audio modules must do
the most accurate job they can.

3.7. Summary of the Speech Manager

The Speech Manager has three components:

- *The Speech Library* presents the uniform speech synthesis API to the application and communicates with the device driver in use.
- *The Device Drivers* translate requests from applications into synthesizer-specific API function calls. They also obtain the synthesized audio data from the speech synthesizer and route it either to the application or an Audio module. Device drivers are also responsible for providing robust information and status events and translating in-stream synthesizer instructions into a synthesizer-dependent form.
- *The Audio Modules* provide a means by which the Speech Manager is not bound to play audio data using a single mechanism. Audio modules allow applications to use the audio server or platform of their choice to play synthesized audio data.

The Speech Manager provides the system by which applications actually interface to the text to speech technology and make use of this technology as a means of feedback to the user.
Chapter 4  Discussion and Future Directions

This chapter provides some concluding remarks for this thesis. Section 4.1 provides a status update for the thesis and outlines the implementation status of various modules and components mentioned throughout the thesis. Section 4.2 presents some future directions for this work.

4.1. Implementation Status Update

Both the Speech Manager and the TTSML Preprocessing system are fully implemented as described in this thesis. The following subsections provide details on each.

4.1.1 The TTSML Preprocessing System

The TTSML Preprocessing system is fully implemented and has more than twenty markup and process experts. These experts are: *abbreviation*, *acronym*, *address*, *compound*, *computer*, *contraction*, *date*, *lexicon*, *math*, *organizations*, *people*, *plural*, *political*, *possessive*, *propername*, *punctuation*, *state*, *telephone*, *time*, and *part of speech*. New experts and tags are added as more extensive experiments with text samples are per-
formed. Very few problems have been encountered with respect to the ordering of experts. Almost always, an entire domain of text items (contained within a single expert) intuitively comes before other domains of text items during the Markup Stage. When a certain text samples were incorrectly marked up in the initial stages of implementation, the problem was easily fixed by editing the configuration file and changing the location of a single expert.

One slight problem of the TTSML system is its performance: its is tremendously slow during the Markup Stage on older workstations. The TTSML Preprocessing system runs in real-time on an 85 Mhz SparcStation 5. However, on an older SparcStation 1+, a paragraph of a dozen lines took up to thirty seconds to preprocess. The code which attempts to match a tag’s definition is not optimized, making many more recursive function calls than is necessary. The searching of lexicons during the Markup Stage is most responsible for the performance hit on the older workstations. Currently, the input text is searched for each entry in every lexicon. If such a word exists, the input text is marked with the appropriate tag. The particular implementation used to accomplish this is responsible for the poor performance. The search time grows polynomial with the length of the input text grows and the length of each lexicon. A proposed solution to this problem involves breaking the input text into words at first, and then hashing these words into the lexicons. The hashing will allow the performance to gain independence of the size of the lexicons.

4.1.2 The Speech Manager

The Speech Manager’s current implementation integrates two text to speech synthesizers: DECtalk, Truetalk. Rsynth may easily be integrated into the thesis if an interest exists. A fourth synthesizer, British Telecom’s Laureate [Ashworth 1995] was originally planned to be part of the Speech Manager, yet the required software has not yet been obtained. The synthesizers integrated into the Speech Manager so far provide a breadth of different underlying interfaces, and exercise the Speech Managers capabilities of integrating dispar-
ate interfaces into one API. A standard, default audio module is fully implemented in addition to an audio module which uses the Media Lab's Audio Server. The development of an audio module for Digital's AudioFile is left for the future, depending upon interest. Also, the Speech Manager may integrate Sun's SunSpeak API when it becomes available.

4.2. Future Directions

This section describes some future directions for work which follow from this thesis. Some of this work would have been examined by this thesis time permitting.

4.2.1 Prosodic Controls and Phonetic Character Sets

As was mentioned several times throughout this thesis, there is a great need for a language-independent phonetic character set. Some claim that IPA achieves this goal, while others claim it falls short. Another problem left unsolved by this thesis was a single set of robust prosodic controls for text to speech synthesis. The first means of attack for this problem is an extensive study of all available text to speech synthesizers. Unfortunately, third-party vendors were not willing to provide a copy of their documentation without first paying the price of the software development kits. A synthesizer-independent set of prosodic controls will permit applications to add intonation and emotion to their synthesized speech.

4.2.2 Support for multiple languages in TTSML

Foreign languages may have their own constructs, completely different from English. Foreign languages not only may require a different character set, they may also require a completely different set of experts. Support in TTSML exists now for applications to switch between entirely disjoint sets of experts. While this facility has been used only for debugging purposes to date, it may also be used to switch between sets of experts for different languages.
4.2.3 Definition of new tags using TTSML itself

Although TTSML borrowed many concepts from SGML, there are still many which it can benefit from. Because SGML is a *language*, it contains constructs which extend the language, using the language itself. With respect to text to speech synthesis, TTSML can allow applications to define new tags via some TTSML "definition" tag. In this way, a document itself can extend the processing capabilities of the TTSML system. For example, suppose a document from the United Kingdom contained a TTSML tag which defines a new tag to describe British telephone numbers. After processing this document, the TTSML Preprocessing system has extended its capabilities to identify different telephone numbers. The TTSML Preprocessing system may extend its *Process Stage* as well, as discussed below. Metaphorically, the TTSML system gains intelligence through the more documents it sees.

4.2.4 A Simple Language to Simplify the Process Experts

The development of process experts is sometimes convoluted. Although a utility is provided to navigate the parse tree, the code is still messy. Ideally, the processing of marked-up text can happen using some scripting language. In FREND [Pavlovcik 1991], the Pars language provides a set of primitives to process text in this manner. This scripting language does not have the capabilities of a general compute language such as C, yet it provides an easy means to process each component subtag of a tag and perform more efficient and clean string manipulations. This scripting language for processing is especially useful because these scripting language commands may be imbedded within TTSML-tagged text. This way, applications may alter the way a tag is presented. Applications may imbed the new way to process a tag within the text itself.
References


