A Text to Speech System for the IBM PC

By

Benjamin Wooller,

Department of Computer Science and Electrical Engineering,

University of Queensland

Submitted for the degree of

Bachelor of Engineering (Computer Systems - Honours)

October, 1998

Acknowledgments

I would like to extend my gratitude to my thesis supervisor, Dr. Mark Schulz, for guiding me to the right people and places, and for his motivational comments on my work. Thanks also go to Dr. Julie Vonwiller for her help in the field of speech synthesis, and Alain Ruelle for his help with Mbrola. I must also thank my patient family who made sure their computer was always available when I needed it.

Abstract

Text to speech systems have generated much commercial and academic interest in recent years. While text to speech conversion is a complicated process, it can be broken down into two main tasks - text analysis and speech synthesis. This thesis aims at developing a complete software-based text to speech system, although it concentrates mainly on the text analysis problem.

The system developed for this thesis, TexTalk, is implemented in the Windows environment using Visual C++. The application takes unrestricted English text as input and produces synthesised speech as output. Algorithms combined with a large English dictionary ensure that words are given the correct pronunciation and intonation. Speech quality is good, but does not yet sound like natural human speech.

The system is a good start to producing realistic speech from text, but there are several areas which can be improved. The prosody, or intonation and rhythm, of speech has not been accurately modelled and there is scope for further research there. The pronunciation of ambiguous words (where two or more pronunciations could apply) is another area where further research is required.

Contents

List of Figures

List of Tables

1.0 Introduction

1.1 Text to Speech

1.1.1 An Overview

The aim of a text to speech system is to convert an arbitrary body of text into an intelligible speech waveform. There are many reasons why such a system would be valuable, ranging from reading machines for the visually impaired to intelligent pagers which talk to you. Any application where speech can used to replace, or assist the use of, other senses would benefit from a text to speech system. The same techniques used to produce speech from its basic units could also be used to cut down communication bandwidth dramatically by transmitting only those units instead of the entire speech waveform.

The task of text to speech conversion can be broken down into two main parts: text analysis and speech synthesis. Text analysis transforms the input text into its abstract underlying linguistic representation, and then speech synthesis transforms this representation into the speech waveform output. This process is illustrated below in Figure 1.1. The abstract linguistic representation is made up of phonemes, which are the

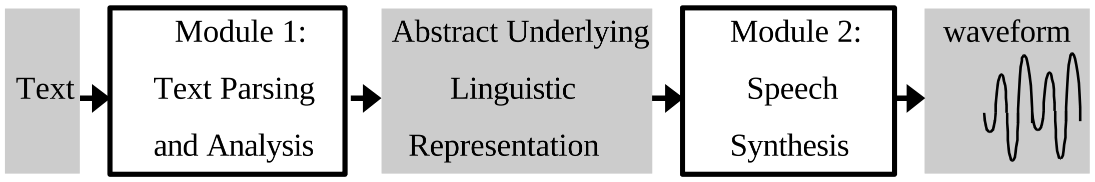


Figure 1.1 - Block diagram of the overall system of a text to speech converter.

most commonly used unit in the transcription of speech. Pitch and duration parameters are also generated by the text analysis routines in order to represent intonation and stress (the prosody of the text).

Text analysis is concerned with generating phonetic and prosodic information from unrestricted English text, which can include any word, name, abbreviation, number or symbol. In order to generate the prosodic information, the structure of each sentence must be understood, and the intonation of each word must be calculated. This is a difficult problem, however, because the prosody of a sentence is often open to interpretation, and there may be more than one way of speaking a sentence.

1.1.2 Text Analysis

Text analysis methods include analysis by algorithm, lexicon look-up, and analysis by neural network. Analysis by algorithm uses a large set of rules based on the structure of words in the English language. There are always exceptions however, and lexicon look-up is often used to complement analysis by algorithm. The lexicon is a large dictionary of words with their phonetic transcription, and stress information. This method gives accurate transcriptions, but only works for words that exist in the dictionary. The final method, analysis by neural network, uses artificial neural networks to analyse the text. They have been shown to work (NETtalk for example [10]), but they do not currently reach the accuracy of algorithmic and lexicon based systems (such as MITalk [5], and DECtalk [11]).

1.1.3 Speech Synthesis

Speech synthesis is concerned with producing the digital samples that make up the speech waveform, and generating soundwaves from these. Synthesis techniques are either concatenative, or parametric in nature. Concatenative techniques use a database of the basic units of speech and concatenate them, sometimes with spectral smoothing, in order to produce speech. Parametric techniques use parameters, which are quantities parameters are applied to a virtual voice box which produces the desired speech output. In linear prediction synthesis, the parameters can be related to the shape of the vocal tract of a human speaker. In formant synthesis, the parameters can be related to the positions of vocal tract resonances.

1.2 An Ideal Solution

An ideal text to speech system would be software based, so that new upgrades or modules can be easily added to the system. It would have an attractive and easy-to-use interface, and the user would be able to direct input from any source, such as:

- email messages
- files
- error messages
- keyboard input

The text analysis module would be able to recognise any word, name, abbreviation or number currently in existence. It would also be able to guess at the pronunciation of misspelt, or made-up words. The system would be accurate, and the output would have the rhythm, stress and intonation of human speech. In cases where the same spelling has two pronunciations (such as live), the system would choose the correct word based on its part of speech.

A utopian system such as this may never be created, but current commercial systems come close. It is certainly way beyond the scope of a thesis, however it is something to aim for.

1.3 Aim of This Thesis

As the title of the thesis suggests, the aim is to develop a text to speech system for the IBM PC. This system aims to meet the following requirements:

- accurate recognition of words
- unrestricted English text as input (any word, abbreviation or symbol)

- clear, intelligible synthesis of speech
- correct prosody of speech (rhythm and stress)
- user-friendliness

The system is built, similarly to a hardware design problem, by selecting the right components and incorporating them into the design. The methods used for text analysis and speech synthesis are chosen by reviewing possible candidates and selecting one. Then, the various parts of the system are interfaced with one another to produce the complete text to speech system. Because speech synthesis and text analysis are expansive fields on their own, this thesis will concentrate on the text analysis section, though synthesis is also examined.

1.4 Thesis Structure

Chapter 1 is intended to introduce the topic of computational linguistics and the related fields of speech synthesis and analysis. Readers new to the field may need to use the glossary in appendix A. Chapter 2 is a review of current solutions to the text to speech problem, while chapter 3 provides more detailed theory on text to speech conversion. Chapter 4 details my approach to solving the problem of text to speech conversion, and describes the design of TexTalk. Chapter 5 gives some detail on the implementation of TexTalk, and how it works. Chapter 6 describes the performance of TexTalk, and lists the results of a small speech quality survey. Lastly, chapter 7 contains the conclusions and discusses possible further developments.

2.0 Review of Systems and Methods

This is a review of important and useful literature on the subject of text to speech synthesis. The items included are either commercial products, or academic research related to the topic which will be used to derive the specifications for the system. Each item is reviewed and compared to the others with respect to various characteristics. There is a table at the end of this chapter which summarises the findings.

2.1 Product Characteristics

The products need to be compared and assessed on certain characteristics (though these characteristics may not apply to all systems). The following characteristics will be considered:

- **Input Source:** May be from various sources, such as user input, file input, or input from other applications. The usefulness and flexibility of the system can be limited by this factor.
- **Input Range:** The input is considered to be a stream of characters representing text. The range of understood words, and the formats supported by text to speech systems can vary. The versatility of a system is limited by this factor.
- **Text Analysis Method:** There are generally three main methods of text analysis used in text to speech. These are algorithmic, look-up, and neural network methods. The algorithmic approach uses a set of clearly defined rules to analyse the text [5]. This method can accept an unlimited vocabulary of words, but may not pronounce strangely spelt words or abbreviations. Look-up methods involve finding words within a dictionary which returns the pronunciation of the word. This will give perfect results for words in the vocabulary, but cannot pronounce any words not included in this vocabulary. The dictionary may also require a large amount of memory depending on the number of words included (approximately 100 000 in the English language).
- **Speech Quality:** This factor defines how comprehensible the speech output is. This is a qualitative attribute which cannot accurately be measured. The best

method of measuring this value would be to survey the comprehension rates of a large population of people. This is beyond the scope of this report however, and only a rough indication of the speech quality of systems will be given.

- **Synthesis Method:** There are a variety of synthesis methods such as phoneme concatenation, diphone concatenation, linear predictive coding (LPC), and formant synthesis. The method chosen for synthesis dictates the type of output needed from the text analysis module. The specific implementation of the synthesis method will have a large impact on the quality of speech.
- **Synthesising options:** A versatile and well-designed system will allow the user to select from a range of options for voice synthesis. These could include pitch, speed, head-size, sex of voice, language and more. Having a greater range of options would allow the user to customise synthesis.
- **Platform:** Only software text to speech systems are reviewed in this report. Even though a system may not work on an IBM PC, the techniques used may be important, so this will not be a very important factor.

2.2 Eloquence

Eloquence is a software text to speech solution from Eloquent Technology Incorporated [9]. It has an unlimited vocabulary because it uses sophisticated algorithms for text analysis. The algorithms also predict stress and pronunciation patterns, and can distinguish between the stress in words such as ‘wicked’ and ‘picked’. Context is taken into account as well as individual words when calculating stress.

The actual method for synthesising speech is the Delta System, a sophisticated software tool for research and development in speech synthesis [2]. The basis for the system is the delta, a data structure that can accommodate both abstract linguistic and quantitative data in a single utterance representation. Formant synthesis [3] is used in the actual production of speech. The Delta programming language or the Delta Tools environment can be used to construct these deltas.

The quality of speech produced by Eloquence is high (see [9] for samples). It still sounds robotic when compared to a human speaker, but the intelligibility is excellent. Eloquence handles inflections and intonation of sentences with a very high rate of accuracy, without sounding monotonous or randomly intonated. Eloquence's algorithms mimic the rises, falls, and pauses in natural speech, which improves the user's comprehension and retention of the spoken text.

There are also a variety of options that can be used in synthesis. The user can control various attributes including head-size, pitch, sex, and breathiness. These options allow the user to customise the synthesised voice to best suit their needs. A variety of languages and dialects are also supported by Eloquence. These are US and UK English, Castilian and Mexican Spanish, French, German, and Italian.

Eloquence runs on the Windows(3.1, 95 and NT) and Unix platforms. Eloquent Technology also distribute software development kits (SDKs) for both of these platforms.

2.3 Monologue 97

Monologue 97 is a text to speech application distributed by First Byte, a market leader in speech technology [8]. It utilises First Byte's ProVoice text-to-speech tool, which is available commercially as the ProVoice Developer's Tool Kit.

Monologue was developed for the Windows 95 platform, and can optionally read text from the clipboard in Windows applications. Other options such as SpeechFonts allow various accents and languages to be chosen. Currently US English, UK English, French, German, Spanish and Italian are supported. Additionally, the output of the SpeechFonts can have the pitch, speech rate, or volume modified during program execution.

The quality of speech is excellent (see [8] for samples). Monologue uses First Byte's patented time domain concatenative synthesiser technology to provide intelligible speech with low CPU usage. This allows Monologue to run simultaneously to other applications such as graphic face animation or word processing. The synthesiser uses concatenative techniques on a table of speech information which is used with the phonetic and prosodic information from the text analysis stage. This table is the result of a series of speech compression processes that analyse raw digitised speech. They determine the optimal combination of compression techniques and sound blending to be used for their storage into tables.

The text analysis is done by dictionary look-up, or by algorithms if the word is not found. Prosodic analysis is also done by a complex set of algorithms [2]. This combination of dictionary look-up and algorithmic methods allows for an unrestricted input range as well as accurate word recognition. The text is converted into phoneme sequences which are used by the synthesiser to form the speech output. These phoneme sequences consist of codes corresponding to the standard set of phonemes as well as certain allophones(phonemic variants). Pitch, duration, and amplitude codes are also calculated from the text input.

2.4 Whistler

Whistler is Microsoft’s trainable Text-to-Speech (TTS) system [1]. It is different to both Eloquence and Monologue, in that it automatically learns the model parameters from a training corpus. Both prosody parameters and concatenative speech units are derived through the use of probabilistic learning methods that have been successfully used for speech recognition. Whistler can produce synthetic speech that sounds very natural and resembles the acoustic and prosodic characteristics of the original speaker. The underlying technologies used in Whistler can significantly facilitate the process of creating generic TTS systems for a new language, a new voice, or a new speech style.

Whistler is different to the other systems reviewed in the way that it generates the speech units and the prosody of the text. The prosody is generated from prosody before-hand to generate the prosody templates and the unit list. The analysis and run-time parts of the Whistler TTS system can be seen in figure 2.1.

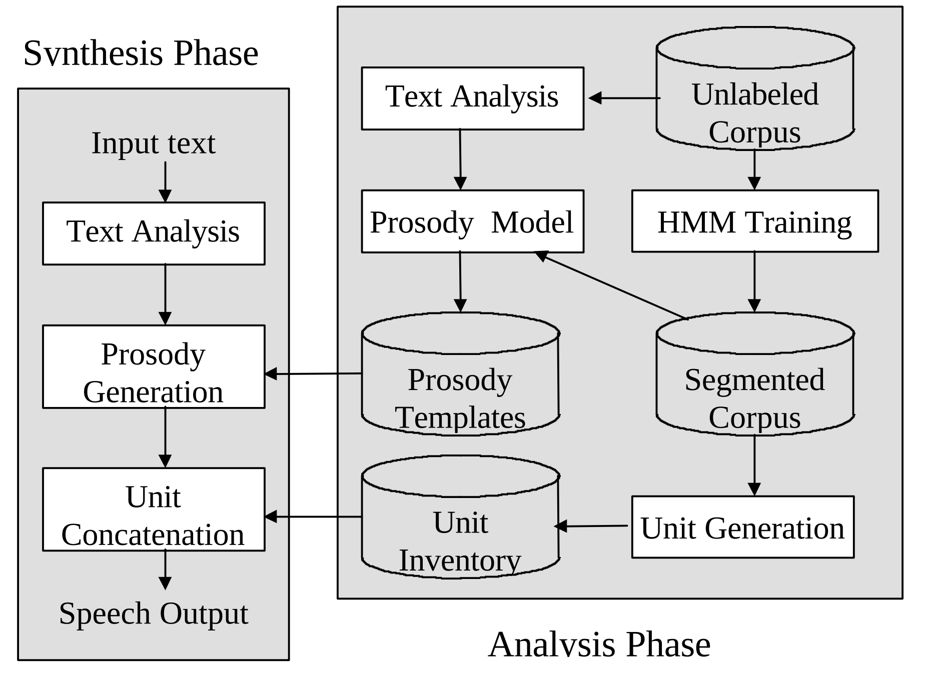


Figure 2.1 - Block diagram of the Whistler TTS system [1].

Whistler's text analysis utilises Microsoft's natural language processing (NLP) engine for basic text understanding. The NLP system performs a flexible parse of domain-independent sentences, producing as output a complete grammatical parse tree, including part of speech tagging for each word. From this parse tree, sentences are broken into phrases and pauses. The input text is then converted to a phonetic string with lexical stress marks and pause marks by the use of:

- the parse tree
- the phrasing information
- the pronunciation dictionary
- and the letter-to-sound (LTS) module.

The phonetic string and the word list, augmented with the phrasing information, are then passed to the prosody generation module. Because of the use of the LTS module as well as pronunciation dictionary, the input text is unrestricted.

The speech output is of whatever form the system was trained to produce. Thus it could be used for most languages, and provide a range of different voices depending on how it was trained in the analysis phase.

Obviously the system runs on the Microsoft Windows platform. Currently there is a software development kit (SDK) available for creating speech applications [1].

2.5 Mbrola v3.00

The Mbrola Project initiated by the TCTS Lab of the Faculté Polytechnique de Mons (Belgium) aims to obtain a set of speech synthesisers for as many languages as possible, and provide them free for non-commercial applications [7]. The ultimate goal is to boost academic research on speech synthesis, focusing on prosody generation, known as one of the biggest challenges taken up by Text-To-Speech synthesisers for the years to come. No text analysis is performed by Mbrola, but it is a highly competent synthesiser.

Mbrola is a speech synthesiser which uses the concatenation of diphones, combined with spectral smoothing, in order to produce speech. As input, it accepts phonemes and prosodic information in a special format. Prosodic information is represented by a piece-wise linear frequency graph, and duration of the phonemes. The speech produced is 16 bit linear PCM [3] at the sampling frequency of the diphone database. As long as the information provided to it is prosodically and phonetically correct, the speech is of excellent quality. This is because of the use of diphones (refer to section 3.2.1), and outstanding diphone smoothing capabilities, due to the particular format of the diphone database.

The diphone database is an important component of Mbrola. These are highly processed speech samples that are used in the synthesis of speech. Different databases can be selected if a synthesiser of another voice or language is desired. The various diphone databases currently available for use with Mbrola are: Brazilian Portuguese, British English, Dutch, French, German, Romanian, and Spanish. Both male and female voices are available for each. Academic and research laboratories are encouraged to share their diphone databases, which can be sent to TCTS for processing into an Mbrola diphone database.

Mbrola is available on a variety of platforms including Unix and Windows 95. The windows version includes dynamic link library (DLL) functions which can be used in the development of speech synthesisers.

2.6 CUV2

CUV2 is the second version of the Computer Useable Version of the Oxford Advanced Learner's Dictionary (CUVOALD). This is a digital dictionary produced by Roger Mitton at the University of London. It contains 70646 entries including explicit forms of all inflections of a word (such as word, words, worded, wordy, wording, etc) as well as common abbreviations and names. Each entry contains these fields:

- the word spelling field (23 characters long)
- the pronunciation field (also 23 characters)
- the syntactic-tag field (also coincidentally 23 bytes long)
- the number of syllables (just one character '1' to '9')
- the verb-pattern field (58 bytes long)

The fields are padded with spaces so that each entry has a constant length of 129 bytes (there is a line feed character appended to the end of each entry). The constant length of the fields simplifies a dictionary look-up. The position in the file of the entry can be calculated by multiplying the entry number by 129. This is only marginally slower than using a more complicated look-up table system (which could be implemented with this dictionary if needed). The dictionary is ordered by the spelling field, which is useful if a binary search is used to find the spelling of a word. The word's other fields such as pronunciation may be used in text to speech conversion.

This dictionary is an easy to use free resource from the on-line Oxford archives [6].

2.7 DECtalk

The DECtalk speech synthesiser is a text to speech system produced by the Digital Equipment Corporation. Unlike the other systems, DECtalk is available in both hardware and software form. The software version, which is reviewed here, has been derived from the original hardware version and has only recently been available. It is interesting to note that an earlier system, MITalk [5], was a precursor to DECtalk.

DECtalk software is available in the form of a software development kit (SDK) at a price of $US 500. It is designed to be used by developers of text to speech applications and comes on either the Windows or DIGITAL Unix platforms. It provides the developer with complete text to speech abilities, and can accept input in the form of an application programming interface (API). Text analysis is performed by a large dictionary which can be customised for the developer's application. Synthesis uses the format method, and the user can select from a choice of 9 different voices for output.

2.8 NETtalk

The NETtalk system was developed by Sejnowski and Rosenberg in 1986 in order to apply the generalising capabilities of artificial neural networks to the problem of text to phoneme conversion [10]. It is concerned only with the text analysis stage of text to speech conversion, and a separate synthesiser is used to create the speech from NETtalk's output. NETtalk manages to obtain a success rate of over 90% for a set of training words, but only a 77% success rate for an unlimited set of words [10]. This a very low rate when compared to other lexicon or algorithmic systems. While the potential for neural networks in text analysis is great, neural net techniques cannot obtain sufficient accuracy to warrant deeper investigation for this thesis project.

2.9 Summary Table

This summarises and compares the important aspects of each item.

System	Input source	Input Range	Text Analysis Method	Synthesis Method	Synthesis Options
Eloquence	Application Programming Interface (API)	unlimited	algorithms	Delta system, (formant)	languages, and speech attributes
Monologue	clip board, file, user input, API	unlimited	dictionary and algorithms	ProVoice, concatenative	languages, SpeechFonts
Whistler	API	unlimited	dictionary and algorithms	variable unit concatenation	depends on speech database
Mbrola	API, file	unlimited	none	diphone concatenation and spectral smoothing	languages, pitch and speed
CUV2	N/A	limited to 70000 words	N/A	N/A	N/A
DECtalk	API	default and custom dictionary	dictionary	formant	9 voices, pitch and speed
NETtalk	training or test applications	unlimited	neural networks	N/A	N/A

Table 2.1 - Comparison of reviewed systems.

3.0 Theory

The aim of this chapter is to familiarise the reader with the vocabulary used by linguists, and the methods used in text to speech conversion. These are the basis on which the design decisions described in Chapter 4 are made.

3.1 The Basics

Text to speech conversion is no trivial task. In order to understand it more easily, it is best to examine the make-up of speech. Phonemes are an important unit of speech. They represent the different sounds that make up the words of a language. A set of 44 phonemes make up the SAMPA standard for the English language [7]. A list of these phonemes, and their pronunciation appears in Appendix II.

Prosody, another important quality of speech, is defined as the patterns of stress and intonation contained in the sounds of speech. It refers to rhythm, and the way our voice rises or falls depending on the structure of the sentence. The difference between natural human speech and a simple robotic monotone can easily be seen. More specifically, these attributes can be defined as the duration and pitch of the phonemes which make up synthesised speech.

3.2 Speech Synthesis

3.2.1 Concatenative Methods

Concatenative techniques invlolve appending various recorded samples, one after another, in order to make up the final speech waveform output. The size of the samples vary from entire phrases, to words, to phonemes or diphones. Sampling entire phrases gives a good quality output, but all possible phrases cannot possibly be recorded. Word sampling is a good way to implement a limited vocabulary system, but the input text will be limited, and there will be no variation of pitch during a sentence. Sampling phonemes is a good method because there are only around 40 of them in the English language (depending on the dialect).

In simple concatenative systems, the phoneme samples are simply played one after another. This simple method has the disadvantage of producing disjointed speech because the joins between phonemes are not smooth. Another unit often used is the diphone. There are many more diphones than phonemes (around 2000) in the English language, but they produce much smoother speech. This is because a diphone is the back half of one phoneme joined to the front half of another phoneme (see Figure 3.1 below). They are produced by processing a large number of phoneme samples to ensure the transitions are smooth. The diphones are then simply concatenated when synthesis is required.

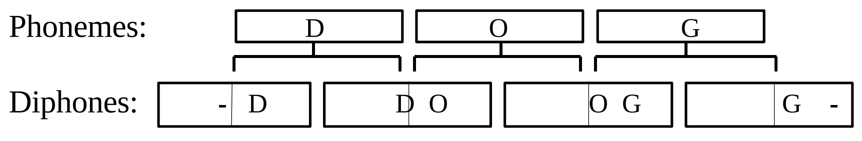


Figure 3.1 - The 3 phonemes and 4 diphones in the word ‘DOG’.

3.2.2 Parametric Methods

Parametric techniques are a different approach entirely. Instead of using a database of samples to produce speech, these techniques use a set of input parameters, together with a system of noise generators and filters. The two main groups of parametric synthesisers are formant and linear predictive.

Formant synthesisers model the human production of speech by using parameters which record information about *formants*, resonances within the vocal tract. The parameters measure things such as the amplitudes and frequencies associated with the different parts of the speech model (voicing, fricative, aspiration, and the various formants). The system is usually made up of a number of formant filters, a sound source, a random noise generator, and a number of adders and multipliers.

Linear predictive synthesisers use parameters which describe vocal tract properties, rather than speech properties. The parameters relate to pitch, energy, and the shape of the vocal tract. These parameters are sampled much less often than speech, and this gives linear predictive methods a much lower data rate than normal speech sampling methods.

3.3 Text Analysis

A text to speech system which uses plain text as its only input faces several very difficult problems. Natural sounding speech has a dynamic pitch and rhythm, and this contains many cues which are used by the listener to aid in understanding. Text cannot, obviously, contain this information. There are other problems such as:

- differentiating between words that are spelt the same, for example read can be pronounced "*reed*" or "*red*", and live can be pronounced "l-eye-v" or "
- determining which words in a sentence should be stressed
- pronouncing irregular words such as colonel
- pronouncing new, or made-up words that do not appear in dictionaries

Each of these problems belong to the area of either prosodic or phonetic analysis.

3.3.1 Phonetic Analysis

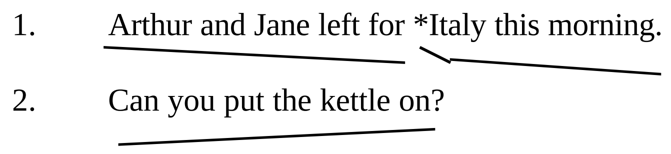
Phonetic analysis is concerned with processing the input text, and producing the correct phonetic representation for the words in that text. There are a few different approaches to phonetic analysis as can be seen in chapter 2, which reviews several speech systems. The most straight-forward method is lexicon look-up, in which each word in the text is searched for in a large database which contains the word's phonetic representation. This method does not work for words not included in the database, such as names of people or places, new technical words, misspellings of normal words, or invented words. In order to overcome these difficulties, algorithmic methods can be used. The rules in these systems are derived from the structure of English words. For example, at the beginning of a word, a 'c' followed by an 'e' has a soft 's' sound, as is "celery" or "cement". The problem with algorithmic systems, is that there are always exceptions, such as the hard ‘c’ in the word “Celt”. To overcome all of these difficulties, some systems have used neural networks for the problem of recognising the pronunciation of words. Unfortunately, they do not currently have the accuracy to compete with algorithmic and lexicon based systems.

An additional problem is determining the pronunciation of words which are spelt the same but sound different. The word “the” is said as either “*thuh*” or “*thee*” depending on whether or not it is stressed, and whether it appears before a vowel. Similarly, the word “read” is pronounced “*reed*” or “*red*” depending on its tense. The only way to determine which is correct is to look at the words surrounding the word in question. The word’s part of speech is important in determining the true pronunciation of ambiguous words.

3.3.2 Prosodic Analysis

Prosodic analysis can use the same techniques used in phonetic analysis, applying them to the task of generating prosodic information from plain text. Each multi-syllable word usually has one or more syllables which are stressed. The nature of this stress can, with a little simplification be represented by a peak in pitch for the stressed syllables. Prosodic analysis, then, must be able to compute the stress points in a word by examining the text input. This can be done word by word, as in phonetic analysis, by a lexicon, algorithmic, or neural network systems. Each having the same advantages and disadvantages as discussed in section 3.3.1.

Another facet of prosody is the gradual rising or falling of pitch over the length of an entire sentence. This is represented by a pitch contour of the fundamental frequency of speech. While each word has its own intonation, the pitch of a natural sounding sentence follows this pitch contour. Figure 3.2 shows two sentences with their pitch contours below them.



1. Arthur and Jane left for *Italy this morning.
2. Can you put the kettle on?

Figure 3.2 - Two sentences with different pitch contours.

Sentence one has a typical descending contour with a stressed word in the middle. This stress is different to syllabic stress and can only be determined if the concept of the sentence is understood. For example, the stress could have been placed on morning if the sentence had a different purpose. The second sentence has a rising pitch contour, typical of a yes/no question. In conclusion, there are two ways in which prosodic information is extracted from text. The first examines each word individually, and the second examines the sentence as a whole, taking into account punctuation and the placement of words within the sentence.

4.0 Design of TexTalk

The TexTalk system derives its specifications from the systems and theory reviewed so far. This chapter gives a logical account of the design processes involved in the implementation of the system, and a brief overview of how it works. As was shown in Figure 1.1, the two main parts of a text to speech system are analysis and synthesis.

4.1 Method of Synthesis

The method of synthesis is the most important specification because it defines and limits the choice of other characteristics in a text to speech system. The quality of synthesis, a very important factor, is excellent in all the systems reviewed. Thus, the selection of a suitable synthesiser depended mainly upon availability and ease of use. The Mbrola speech synthesiser was selected because it is available free for educational purposes, and because it has a simple interface in the form of a Windows compatible Dynamic Link Library (DLL). Mbrola is free for non-commercial use in order to promote research and to enable research projects to produce text to speech systems more easily. Mbrola's functionality is much the same as the other synthesisers. Phonetic and prosodic information are used as input, and the output speech has a variety of options such as different voices, and control of speed and pitch. Interface to soundcard hardware is automatic, and any Windows compatible soundcard will work with Mbrola. In functionality and quality it compares well with the systems used in the commercial products reviewed. For the purposes of this thesis Mbrola is a perfect choice. Because it is in the form of a DLL, future versions of Mbrola can easily be installed and used without altering TexTalk.

4.2 Method of Text Analysis

The two main text analysis methods used by the products reviewed are lexicon lookup and algorithmic. Many of the products combine these methods, giving the accuracy of the lexicon lookup method, and the flexibility of the algorithmic method. TexTalk implements this combination so that the lexicon database is searched first and then, if the word is not found, an algorithmic analysis of the word produces the required phonetic information.

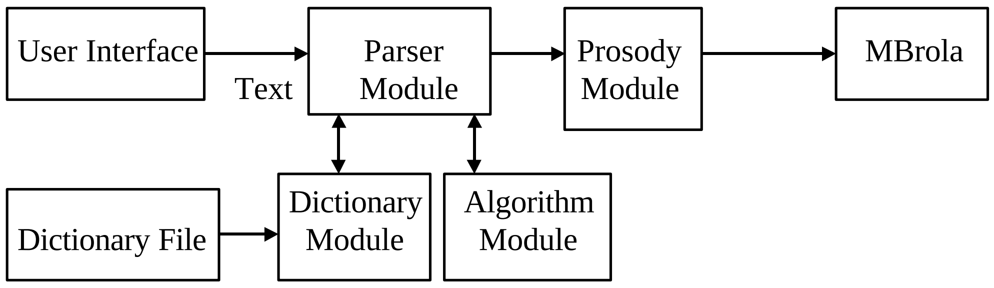


Figure 4.1 - Structure of the TexTalk system.

The lexicon database used in TexTalk is called CUV2 (Computer Useable Version 2). This is a digital version of the Oxford dictionary made specially for linguistics. It contains the spelling, phoneme representation, stress patterns and some other data for each of the 70000 entries. Because each entry has a constant length, the position of each word within the file is can be calculated easily, and a binary search is used in order to take advantage of this. For each word parsed, the dictionary gives the phonetic transcription of the word which includes markers that indicate stressed syllables. If the word is not found, the algorithm module is used to analyse it, as shown in Figure 4.1.

The algorithm module of TexTalk uses a set of rules adapted from a program written by John Wasser [11]. It searches through a list of letter to phoneme rules until it finds one that matches the current letter pattern, and then it moves on to the next section of the word. This gives correct output for most types of words, while exceptions should be picked up by the dictionary.

The other part of text analysis is the prosodic analysis of the text. The basic syllabic stress pattern of a word is contained in the dictionary database. If an unusual name or word is not found in the database then it is best to leave it unstressed. Thus, there is no need for a prosodic analysis of each word. However, there is a requirement for prosodic analysis of the sentence. This is a simple analysis of punctuation which determines whether the sentence has a rising or falling pitch.

Once the analysis of all the text in a sentence is complete, the prosody module converts the information into the format used by Mbrola. This includes calculating phoneme durations, and pitch variations from the data obtained from the analysis.

Module	Component
Parsing	Custom built
Dictionary	Custom built, reading from the CUV2 Oxford Dictionary
Algorithm	Derived from “English to Phoneme Translation” by John Wasser
Prosody	Custom built
Synthesis	Mbrola V3.00, from the TCTS Lab of the Faculté Polytechnique de Mons

Table 4.1 - Summary of components used in TexTalk

4.3 Other Design Considerations

4.3.1 Dictionary Search Method

The search method used to find words in the dictionary database file must run fast enough so that the text to speech system can run in real time. The CUV2 file is 8.7 megabytes in size, and has 70000 entries. A linear search would have to make 35000 disk accesses on average if a linear search method was used. The next best method is a binary search. This method starts in the middle of the file, and skips forwards of backwards depending on whether the search word is ahead or behind of the current position. Because the file is alphabetically ordered and each entry is of constant length, the search routine can skip forwards and backwards fast. The worst case number of jumps is 17, for a word not in the dictionary. This is easily fast enough for real time operation.

4.3.2 Phonetic Set

The set of phonemes that make up all the sounds in the English language vary from dialect to dialect. For example the British Cockney accent does not include the sound made by the letter ‘h’. American English does not include the sound made by the letter ‘o’ in “pot”. The phoneme set used by TexTalk includes all phonemes common to the majority of English dialects, and contains 44 phonemes. There are also several “standards” for representing these phonemes. A list of the phonemes, along with examples of speech and alternate representations appears in Appendix II. This list also contains the average phoneme durations calculated from data provided by Julie Vonwiller of the University of Sydney.

4.3.3 Text Parsing Algorithm

The method used to parse input text must be very versatile. It must handle whitespace, symbols, hyphens, numbers and punctuation. It must also examine each word individually as well as examining the sentence as a whole. To accomplish this the parsing routine parses text until it comes to an end of sentence punctuation. All words prior to this are stored in an array, and then analysed one by one. Symbols are substituted by their full name where applicable, and numbers are converted to words. All words are sent to the dictionary module and then, if not found there, to the algorithm module. Then the pitch contour of the sentence is calculated. All data is then converted to the Mbrola phoneme and pitch format for synthesis.

5.0 Implementation

5.1 Overview

The design of the overall system has been considered, but this section covers the actual implementation of TexTalk, and how the modules fit together.

Because TexTalk is developed under Visual C++ a user-friendly interface deals with loading files, entering text, and editing text. When the text to speech conversion is started, the text is sent to the parsing module as shown in figure 4.1. Once all the analysis and conversion has been performed, an output string is sent to Mbrola by using an Mbrola library function, and the resulting speech waveform is sent to the speakers. Most of the program code must deal with the steps between these two points.

5.2 The Parser

The first section that deals with the raw input text is the parser. The first level of the parser separates words by whitespace (tabs, newlines and spaces). Each word is sent to the second level parser. If the word is pure and contains only numbers, symbols or letters, then it is sent to the third level parser. If it is a hybrid word, the numbers, symbols and letters are separated, and each part is sent back to the level two parser to ensure each part is pure. Once the third level parser has a symbol or number, it is first converted to word form (such as $=dollars, and 12=twelve) before being sent back to the second level. If the third level has a word consisting only of letters, then the word is searched for within the dictionary. If found, the phonetic translation and stress markers are saved, and the next word is parsed. If not, then algorithms are used to convert the letters into their phonetic representation, and this is saved. TexTalk uses an array of word objects to store this data, which is used to calculate prosody. The array is built up as the words are parsed, and deleted after an entire sentence has been parsed. The reason for this follows.

5.3 Word Objects and Prosody

The words are parsed one at a time from the input string, but an entire sentence must be parsed before the prosody can be calculated. TexTalk uses an array of word object pointers to link to word objects containing information about each word parsed so far. As a new word is parsed, a word object is created, and information about the word, such as spelling, phonetic representation, and pitch information is stored in it. The array is then linked to the object, and the next word parsed until an end of sentence punctuation is met. Once the entire sentence is parsed, it is checked for a rising or falling pitch contour. Then the data in each word object is used with the pitch contour to calculate the correct pitch for each phoneme in the sentence. As these calculations are made, an output string is built by adding each phoneme, duration and pitch level in the required Mbrola format. Then all of the word objects are deleted, and the output string is sent to Mbrola through the MBR_Play() function.

6.0 Performance

So what results has the implementation of the TexTalk system yielded? The quality of speech produced is difficult to quantify, although a small survey gives a rough indication.

6.1 Performance vs Specifications

Technically, TexTalk's performance can be described in the following terms:

- text parsing with symbol, punctuation and abbreviation recognition
- word recognition using a dictionary of 70000 words
- dictionary exceptions handled using morph recognition rules
- prosody implemented by word intonation and simple sentence pitch contour
- speech synthesis using diphone concatenation and spectral smoothing

The system meets all requirements of the original specification except for realistic prosody and correct contextual interpretation. The prosodic requirements are not met because TexTalk only implements two styles of sentences - rising and falling. There are many more possible varieties, but the sentence analysis routines would require a great deal of work. When deciding on the pronunciation and stress of a word, its context and part of speech are not taken into account by TexTalk. "Read" will always be pronounced "reed" even in the sentence "I have read it." The algorithms required to achieve this level of comprehension would be considerably complex.

6.2 Speech Quality Survey

Some quantitative results can be extracted from a small survey on TexTalk's quality of speech compared to that of several other speech systems. The survey involves the participants listening to a sentence produced by three different systems, and ranking them on a scale of one to ten. This test was done with ten participants on a set of 5 sentences. The order of the sentences and synthesisers are randomised so that none has an advantage over the others. The synthesisers compared to TexTalk are Monologue 97 [8], and WinSpeech [12] (a simple phoneme concatenation synthesiser). The test sentences include a variety of sentence types, and word usages. They appear with the full results in Appendix VII, while the summary is shown in table 6.1 below.

System	TexTalk	WinSpeech	Monologue 97
Total Score	29.8	18.3	28.5

Table 6.1 - Results of the Quality Survey

These numerical results are far from accurate, and are only a rough guide, but they do give some indication of the systems' relative speech qualities. Other information provided by the survey indicated the following:

- WinSpeech was very difficult to understand, and sounded very robotic.
- Monologue was slightly clearer to understand than TexTalk,
- but TexTalk sounded the most natural.
- Monologue can pick out the different uses of ambiguous words.
- TexTalk was rated better for speaking numbers.

It seems that TexTalk has good potential but to be truly competitive, the sentence analysis must be improved.

7.0 Conclusions

7.1 Summary

During the course of this thesis, several text to speech systems were reviewed. From these, appropriate specifications were set, and then TexTalk system was designed and implemented. As far as the objective of the thesis is concerned, it can be said that a satisfactory text to speech system has been created. TexTalk is a complete, though perhaps imperfect, text to speech system. Its quality of speech is good, but does not quite reach a commercial level. It produces reasonably intelligible speech from unrestricted English text, although it does not yet implement realistic prosody. It was found that a lot more research and development would be required in order to make this a reality.

7.2 Possible Future Work

Although TexTalk is a complete system in itself, there is room for improvement.

Future work on TexTalk is possible in the areas of prosody and text comprehension. In order to produce natural speech, the system must ‘comprehend’ the meaning of the sentence in order to give correct pronunciation, stress and rhythm. Once the system understands how sentences are structured, the pitch contour and rhythm can be calculated more realistically. Knowing how the words are used in the sentence would also assist in determining the correct pronunciation of ambiguous words. Natural language processing is an area of study which would be central to creating a system such as this.

Other parts of the TexTalk system that could also be improved include the dictionary, and the synthesis module. The dictionary can be expanded to include many more names, places, abbreviations and new words. Refer to Appendix V for information on the format of the dictionary.

The synthesis module, consisting of Mbrola 3.00, can be upgraded merely by installing the latest version. This, as well as several different language databases, is available from the Mbrola web site [7]. It is also possible to send diphone databases to the TCTS lab in Belgium to have them converted into Mbrola databases.

Appendix I. Glossary

Corpus: A body or collection of writings, laws, or data for grammar.

Diphone: A sound segment which spans the transition from the center of one phoneme to the center of the next.

Formant Synthesis: Synthesis from a set of parameters which represent the positions of vocal tract resonances. A set of linguistically-oriented rules and models based upon analysing human speech is used.

Linear Predictive Coding (LPC): A form of speech synthesis which produces sound from parameter such as pitch, energy, and the shape of the vocal tract.

Morph: A string segment. Part of a word.

Phoneme: A basic unit of speech. The phonemes of English include consonant sound such as k, b, t, vowel sounds such as ay, oo, i, ee, and the sounds of combined letters like ch, th and sh.

Prosody: Science of versification, laws of metre. The intonation of speech.

Appendix II. Comparison of different English Phoneme Sets

This list of phonemes contains the 44 phonemes used in TexTalk, and matches them with examples of their use and their representation in the various phoneme standards.

TexTalk Index	Mbrola (SAMPA)	CUV2	Algorithm	ANDOSL	Duration (ms)	Example
33	p	p	p	p	97	put
34	b	b	b	b	65	but
35	t	t	t	t	67	ten
36	d	d	d	d	48	den
37	k	k	k	k	87	can
38	m	m	m	m	64	man
39	n	n	n	n	59	not

Index	Mbrola (SAMPA)	CUV2	Algorithm	ANDOSL	Duration (ms)	Example
68	@	@	AX	@	41	about
69	eI	eI	EY	ei	128	bay
70	aI	aI	AY	ai	131	buy
71	OI	oI	OY	oi	112	boy
72	@U	@U	OW	@u	132	no
73	aU	aU	AW	au	134	now
74	I@	I@		I@	119	peer
75	e@	e@		e:	86	pair
76	U@	U@	UW	u@	134	poor

Appendix III. Source Files

Directory Listings

Listing of the TEXTALK directory:

`DEBUG <DIR>`	Direcory containing debug files, and TEXTALK.EXE
`RES <DIR>`	Directory containing resource files used by Visual C++
`ENGLISH.H`	Header file containing letter to phoneme rules as a constant array
`MAINFRM.CPP`	Implementation of the CMainFrame class
`MAINFRM.H`	Header file
`MBRPLAY.H`	Header file for the MBRPLAY.LIB file
`MBRPLAY.LIB`	Library containing synthesis functions
`PHONEME.CPP`	Implementation of letter to phoneme conversion functions
`PHONEME.H`	Header file
`README.TXT`	A file about TexTalk (you are reading it)
`RESOURCE.H`	Header file for resources
`STDAFX.CPP`	File used by Visual C++
`STDAFX.H`	File used by Visual C++
`TEXT.CPP`	Main parsing, text analysis and conversion code
`TEXT.H`	Header file containing defines and prototypes
`TEXT710.DAT`	CUV2 Dictionary file
`TEXTALK.APS`	File used by Visual C++
`TEXTALK.CLW`	File used by Visual C++
`TEXTALK.CPP`	Implementation of the TexTalk application
`TEXTALK.DSP`	Visual C++ project file
`TEXTALK.DSW`	Visual C++ workspace file
`TEXTALK.H`	Header file
`TEXTALK.RC`	Resource file used by Visual C++
`TEXTALKDOC.CPP`	Implementation of the TexTalk document class
`TEXTALKDOC.H`	Header file for the TexTalk document class
`TEXTALKVIEW.CPP`	Implementation of the TexTalk view class
`TEXTALKVIEW.H`	Header file for the TexTalk view class

Listing of DEBUG:

`TEXTALK.EXE`	The program. It must be run from the Visual C++ environment, or you can copy it to the TEXTALK directory and run it from there.

Listing of RES:

`ICON1.ICO`	An icon
`ICON1.ICO`	Another icon
`TEXTALK.ICO`	Yet another icon
`TEXTALK.RC2`	A resource file
`TEXTALK.ICO`	A differnet icon
`TOOLBAR.BMP`	A bitmap of the toolbar

Source Code Listings

The important source code files are:

- TEXT.H
- TEXT.CPP
- ENGLISH.H
- PHONEME.H
- PHONEME.CPP
- MBRPLAY.H
- TEXTALK.H
- TEXTALK.CPP

TEXT.H

```
// TEXT.H
//
// header file for TEXT.CPP. It contains the defines and prototypes

#include <windows.h>
#define CONVERT_LIMIT 2000    // max chars in converted entence
#define TEXT_LIMIT 1000       // maximum chars in text sentence
#define DICTIONARY "text710.dat"

// The number of entries in CUV2 is given as 70646 in the
// documentation. They are numbered 0 to 70645. The entry size
// is 129 bytes (including the line feed character).
#define MAX_ENTRIES 70646
#define ENTRY_SIZE 129
#define TEXTLEN 23      // max length of text and phoneme strings
#define PHON_OFFS 23 // offset for position of phoneme string in entry
#define T_TRUE 0
#define T_FALSE -1
```

```
void add_word(LPCTSTR w_in, LPCTSTR p_in);
int dict(LPCTSTR in, CString& out);
void show_list(void);
void say_number(long int value);
void extract_number(CString& word);
void p_add_word(LPCTSTR w_in, LPCTSTR p_in);


static char *Cardinals[] =
{
      "zero","one","two",     "three","four",   "five",
      "six","seven",
      "eight","nine","ten","eleven","twelve","thirteen","fourteen",
      "fifteen","sixteen","seventeen","eighteen","nineteen"
};

static char *Twenties[] =
{
      "twenty","thirty","forty","fifty","sixty","seventy",
      "eighty","ninety"
};
```

TEXT.CPP

```
// TEXT.CPP
//
// This file contains the code for parsing the input string,
// generating the correct phonems from the text,
// calculating the prosody of the text,
// and converting the phonemes and prosody into an Mbrola string
// which is then sent to the MBROLA module.

#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include "stdafx.h"
#include <afxcoll.h>
#include "textalk.h"
#include "textalkDoc.h"
#include "textalkView.h"
#include <stdio.h>
#include "text.h"
#include "mbrplay.h"

//prototype of the function in phoneme.cpp
void xlate_phrase(LPCTSTR text_word, CString& ret_phon);

// CWordObj is a structure that contains all important
// information about a word
class CWordObj : public CObject
{
private:
public:
    CString w_text;
      CString w_phon;
    char syllables;
      char phonemes;
      char phoneme[MAX_PHONEMES]; //numerical index to each phoneme
    CString field;
      int pitch[MAX_PHONEMES];  //pitch data for each phoneme

    CWordObj() {}
};

// words is an array of CWordObj pointers
// elements are created and linked to this array
// and then deleted after use
CWordObj *words[MAX_WORDS];

// word_ptr is equal to the number of words in the array
int word_ptr = 0;

//msg is a string used to send messages to the user
CString msg;

// MBrola is the string containing all phoneme and
// pitch information and is sent as an argument
// to the MBR_Play function.
CString MBrola;

// variables containing the current pitch, freq and speed ratios
float pitch_ratio = 1.0;
float duration_ratio = 1.0;
long voice_freq = 16000;

// when prosody_flag is zero, prosody is turned off
char prosody_flag = 1;

// a lookup table that takes in a char value from a cuv phoneme
// string and gives an index number for that phoneme.
// The table has an offset of 38 (thus position 0 is ascii 38)
char phon_lookup[85] = {64,0,0,0,0,0,0,0,0,0,
                        66,0,0,61,0,0,0,0,0,0,
                        0,0,0,0,0,0,68,58,0,0,
                        53,0,0,0,0,62,0,0,0,0,
                        51,59,0,0,0,54,52,67,65,0,
                        0,0,55,0,0,0,0,0,0,1,
                        34,0,36,63,42,48,46,57,56,37,
                        40,38,39,2,33,0,41,44,35,60,
                        43,47,0,0,45};

// a phoneme lookup table for the ttp algorithm phonemes
// has an offset of 65
// the index is the ascii (char) value of a character, and the
// result is a char which represents the phoneme.
char p_lookup[59] = {58,0,49,53,61,0,0,46,57,0,
                  0,0,0,51,71,0,0,0,54,52,
                  67,0,47,0,60,55,0,0,0,0,
                  0,0,0,34,0,36,0,42,48,0,
                  0,50,37,40,38,39,0,33,0,41,
                  44,35,0,43,47,0,56,45
};

// a lookup table that gives the average durations for each phoneme
// it is indexed with an offset of 33
int phon_dur[44] = {97,65,67,48,87,64,59,54,59,88,
                  54,104,72,48,61,64,124,84,61,92,
                  21,108,91,35,78,166,127,88,127,59,
                  84,106,90,89,58,41,128,131,112,132,
                  134,119,86,134};

// a lookup table that gives the MBrola phoneme
// It’s indexed with an offset of 33 (as above), and a multiplier of 2
// (each entry is 2 bytes long)
```

```
char mbr_phon[89] = {"p b t d k m n l r f v s z h w g tSdZN T D S Z j
            i:A:O:u:3:I e { V Q U @ eIaIOI@UaUI@e@U@"};

//global file pointer to the dictionary file
FILE *dictfile;

//////////////////////////////////////////////////////////////////////
// speak is called when the play button is pressed
// First the diphone database and the dictionary file are initialised.
// Then, it picks out words (separated by whitespace) and sends them
// to the word parser which adds them to the words array.
// Once a sentence is complete, mbr_convert_list is called to
// analyse the owrds in the array and calculate prosody.
void speak(LPCTSTR textstr_in)
{
      CString window_str = textstr_in;
      int cursor = 0;

      word_ptr = 0;
      // set the database to en1, and show error if en1 is not
      // registered
      if (MBR_SetDatabase("en1") > 0) {
            AfxMessageBox("Database en1 not registered.", MB_OK, 0);
            return;
      }
      // open the dictionary file
      dictfile = fopen(DICTIONARY, "rb");
      if (dictfile == NULL) {
            AfxMessageBox("Error opening dictionary", MB_OK, 0);
            return;
      }
      // parse textstr
      window_str.TrimLeft();
      window_str.TrimRight();
      while (window_str.IsEmpty() == 0) {
            cursor = window_str.FindOneOf(" \n\t");
            if (cursor >= 0) {
                  if (parse_word(window_str.Left(cursor)) ==
                        SENTENCE_END) {
                        // convert the readymade word list (should
                        // empty it) this also does play_list if the
                        // buffer is too big
                        mbr_convert_list();
                  }
                  window_str = window_str.Mid(cursor);
                  window_str.TrimLeft();
            }
            else {
                  if (window_str.IsEmpty() == 0) {
                        parse_word(window_str);
                        window_str.Empty();
                  }
            }
      }
      // convert extra words if there are any.
      if (word_ptr > 0)
            mbr_convert_list();
      // play the MBrola output buffer if there is stuff in it
      if (MBrola.GetLength() > 0)
            play_list();
      fclose(dictfile);
}
```

```
//////////////////////////////////////////////////////////////////////
// Converts the text string given to a phoneme string based on the
// CUV2 dictionary.  binary search is used to locate the text string.
// Because all entries in the ictionary are of constant length, it is
// easy to find the byte offset of an entry given its entry number.
// If the word is not found, a null string is returned.
int CUV2_ttp(char *text, char *phon)
{
      unsigned long int entry; // the number of the current entry

      // upper and lower are the inclusive limits of the area not
      // searched in the dictionary. They refer to entry numbers.
      unsigned long int upper, lower;
      char entry_tex[TEXTLEN + 1];  // contains the current entry's
                                    // text string
      char search_tex[TEXTLEN + 1]; // a copy of the search string
      char compare;
      char not_found = T_FALSE;
      char pos = 0; // position within the phoneme string
      unsigned char count;

      // fill text with spaces so it matches the dictionary
      for (count = 0; count < strlen(text); count++)
            search_tex[count] = text[count];
      for (count = strlen(text); count <= TEXTLEN; count ++)
            search_tex[count] = ' ';
      upper = MAX_ENTRIES - 1;
      lower = 0;
      entry = upper >> 1;     // divide by 2 to give middle entry

      // go to position in dictionary file, and compare the given text
      // to the text for that dictionary entry.
      fseek(dictfile, entry * ENTRY_SIZE, SEEK_SET);
      fread(entry_tex, TEXTLEN, 1, dictfile);
      compare = _strnicmp(search_tex, entry_tex, TEXTLEN);
      while((compare != 0) && (not_found == T_FALSE)) {
```

```
	if (not_found == T_FALSE) {
		fseek(dictfile, (entry * ENTRY_SIZE) + PHON_OFFS,
			SEEK_SET);
		fread(phon, TEXTLEN, 1, dictfile);

		// now we put the null character at the end of the phoneme
		// string i.e. when the first space character is detected.
		while(phon[pos] != ' ')
			pos ++;
		phon[pos] = 0;
		return 1;
	}
	else {
		phon[0] = 0; // null char at start of the phoneme string
		return 0;
	}
}

//////////////////////////////////////////////////////////////////////
// dict is a funtcion that converts between the CStrings used in
// TexTalk, and  the char*'s which are used in the dictionary lookup.
int dict(LPCTSTR in, CString& out) {
	char phon[TEXTLEN + 1];
	char tex[TEXTLEN + 1];
	strncpy(tex, in, TEXTLEN);
	if (tex[0] < 65 || tex[0] > 122 ||
		(tex[0] > 90 && tex[0] < 97)) {
		// first character is bad.
		return 0;
	}
	if (CUV2_ttp(tex, (char *)phon) == 0) {
		return 0;
	}
	else {
		out = phon;
		return 1;
	}
}

//////////////////////////////////////////////////////////////////////
// add_word takes the text and CUV2 phonetic representation, creates
// a new CWordObj object, links the words array to it, and fills in
// all the information about the word including:
// spelling, CUV2 phoneme representation, number of phonemes,
// the index value of each phoneme, and any stress information about
// each phoneme. the bulk of the function deals with converting the
// one or two letter CUV phonemes into the index of phonemes used in
// TexTalk.
void add_word(LPCTSTR w_in, LPCTSTR p_in)
{
	CString phon;
	char p, p2;
	int index = 0;
	int length;
	int phonemes = 0;
	// This version of Mbrola's diphone database cannot say the
	// letters "TCH". special_word indicates if the letters are
	// present in the word.
	int special_word = 0;

	phon = p_in;
	length = phon.GetLength();
	words[word_ptr] = new CWordObj;
```

```
words[word_ptr]->w_text = w_in;
words[word_ptr]->w_phon = p_in;
if (words[word_ptr]->w_text.Find("tch") >= 0 ||
            words[word_ptr]->w_text.Find("TCH") >= 0)
      special_word = 1;
if (p_in[0] == '_') {
      words[word_ptr]->phonemes = phonemes;
      word_ptr ++;
      return;
}
// convert the dict phon string to indexes in the phoneme array
while (index < length) {
      // set the stress level of the next phoneme
      if (phon[index] == 39)
            words[word_ptr]->pitch[phonemes] = PRIMARY_STRESS;
      else if (phon[index] == 44)
            words[word_ptr]->pitch[phonemes] = SECONDARY_STRESS;
      else
            words[word_ptr]->pitch[phonemes] = NORM_PITCH;
      // calculate the next phoneme
      if (phon[index] > 37) {
            p = phon_lookup[phon[index] - 38];
            if (p == 1 || p == 2 || p == 68 || p == 35 ||
                  p == 36 || p == 63 || p == 62 || p == 67) {
                  // the char may be part of a 2 letter phoneme
                  if (length > index + 1) {
                        p2 = phon[index + 1];
                        if (p == 1) {
                              if (p2 == 'I')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 70;
                              else if (p2 == 'U')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 73;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 2) {
                              if (p2 == 'I')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 71;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 68) {
                              if (p2 == 'U')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 72;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 35) {
                              if (p2 == 'S') {
                                    // exception
                                    if (special_word == 1 &&
                                          phonemes > 0 &&
                                          words[word_ptr]->
                                    phoneme[phonemes-1] != 35) {
                                                words[word_ptr]
                                    ->phoneme[phonemes] = 35;
                                                phonemes ++;
                                                words[word_ptr]
                                    ->phoneme[phonemes] = 49;
                                    }
                                    else {
                                          words[word_ptr]->
                                    phoneme[phonemes] = 49;
                                    }
                              }
                              else {
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                              }
                        }
                        else if (p == 36) {
                              if (p2 == 'Z')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 50;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 63) {
                              if (p2 == 'I')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 69;
                              else if (p2 == '@')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 75;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 62) {
                              if (p2 == '@')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 74;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        else if (p == 67) {
                              if (p2 == '@')
                                    words[word_ptr]->
                                    phoneme[phonemes] = 76;
                              else
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                        }
                        if (words[word_ptr]->
                              phoneme[phonemes] != p)
                              index ++; // a 2 letter phoneme
                  }
                  else {
                        words[word_ptr]->phoneme[phonemes] = p;
                  }
            }
            else {
                  words[word_ptr]->phoneme[phonemes] = p;
            }
            if (words[word_ptr]->phoneme[phonemes] > 32) {
                  //phoneme is correct, increment the index
                  phonemes ++;
```

```
                    }
              }
              index ++;
        }
        words[word_ptr]->phonemes = phonemes;
        word_ptr ++;
}

//////////////////////////////////////////////////////////////////////
// parse_word is a recursive function which takes a series of non-
// whitespace characters as input. It breaks the string into parts
// containing numbers, letters, and symbols.
// Symbols that are recognised are parsed as their word equivalents
// Numbers are sent to say_number() for conversion into words.
// Letters are sent to the dictionary, and if not found there,
// to the xlate_phrase() function which uses letter ot sound rules.
int parse_word(LPCTSTR word_in)
{
      CString phoneme;
      CString word = word_in;
      int ptr;

      // get rid of trailing newline characters
      ptr = word.GetLength() - 1;
      if (word[ptr] >= 0 && word[ptr] < 32) {
            word = word.Left(ptr);
      }
      // parse single character strings
      if (word.GetLength() == 1) {
            if (word.FindOneOf(".?!:;,") >= 0) {
                  // add the punctuation mark to the word list,
                  // and add the appropriate delay too.
                  if (word[0] == '.' || word[0] == ':')
                        add_word(word, "_ 600"); //delay for full stop
                  else if (word[0] == '?' || word[0] == '!')
                        add_word(word, "_ 600"); // ! or ? delay
                  else if (word[0] == ',' || word[0] == ';')
                        add_word(word, "_ 300"); // comma delay
                  if (word[0] == '.' || word[0] == '!' ||
                        word[0] == '?')
                        return SENTENCE_END;
                  else
                        return WORD_FOUND;
            }
            // parse known symbols
            if (word[0] == '$')
                  return parse_word("dollars");
            if (word[0] == '+')
                  return parse_word("plus");
            if (word[0] == '/')
                  return parse_word("slash");
            if (word[0] == '=')
                  return parse_word("equals");
            if (word[0] == '-')
                  return parse_word("minus");
            if (word[0] == '&')
                  return parse_word("and");
            if (word[0] == '*')
                  return parse_word("asterisk");
            if (word[0] == '#')
                  return parse_word("hash");
            if (word[0] == '%')
                  return parse_word("percent");
            // not a specific character, then search dict
            //if (word[0] > (if in range then search dict)
            if (dict(word, phoneme) == 0) {
                  if (word.FindOneOf("0123456789") == 0)
                        return parse_word(Cardinals[atoi(word)]);
                  return UNKNOWN_WORD;
            }
            else {
                  // add phoneme to word list
                  add_word(word, phoneme);
                  return WORD_FOUND;
            }
      }
      // multi character word
      if (dict(word, phoneme) != 0) {
            // add phoneme to word list
            add_word(word, phoneme);
            return WORD_FOUND;
      }
      // get rid of hyphens
      ptr = word.Find('-');
      if (ptr != -1) {
            word = word.Left(ptr) + word.Mid(ptr+1);
      }
      // search for a number
      ptr = word.FindOneOf("0123456789");
      if (ptr == 0) {
            extract_number(word);
            if (word.GetLength() > 0)
                  return parse_word(word);
            else
                  return UNKNOWN_WORD;
      }
      // search for a number or symbol
      ptr = word.FindOneOf(".,;:?!%&#/-+*=$0123456789\"\'");
      if (ptr == 0) {
            // process the first char as a word, then process the rest
            if (word[0] == '$') {
                  // process a dollar value.
            }
            if (word[0] == '\"' || word[0] == '\'') {
                  // ignore the char, and process the rest.
                  return parse_word(word.Mid(1));
            }
            // process the first character, then the rest
            if (parse_word(word.Left(1)) == SENTENCE_END) {
                  parse_word(word.Mid(1));
                  return SENTENCE_END;
            }
            return parse_word(word.Mid(1));
      }
      else if (ptr > 0) {
            // process the part up to the char, then process the rest
            parse_word(word.SpanExcluding
                  (".,;:?!%&#/-+*=$0123456789\"\'"));
            return parse_word(word.Mid(ptr));
      }
      else {
            // there is no wierd character in the word
            // construct the pronunciation manually
            if (word[0] < 65 || word[0] > 122 ||
                        (word[0] > 90 && word[0] < 97)) {
                  // there first character is bad
                  //parse the rest of the word
                  if (word.GetLength() > 0)
                        return parse_word(word.Mid(1));
            }
            // send the word to the translation module
            xlate_phrase(word, phoneme);
            p_add_word(word, phoneme);
      }
      return UNKNOWN_WORD;
}


//////////////////////////////////////////////////////////////////////
// say number takes an integer and converts it to words.
// whenever a new word part of the number is determined, the word
// is sent to parse_word() so it can be added to the word list.
void say_number(long int value)
{
      if (value < 0) {
            parse_word("minus");
            value = (-value);
            if (value < 0)    // Overflow!  -32768
                  {
                  parse_word("infinity");
                  return;
                  }
      }
      if (value >= 1000000000L) {         // Billions
            say_number(value/1000000000L);
            parse_word("billion");
            value = value % 1000000000;
            if (value == 0)
                  return;           // Even billion
            if (value < 100)  // as in THREE BILLION AND FIVE
                  parse_word("and");
      }
      if (value >= 1000000L) {            // Millions
            say_number(value/1000000L);
            parse_word("million");
            value = value % 1000000L;
            if (value == 0)
                  return;           // Even million
            if (value < 100)  // as in THREE MILLION AND FIVE
                  parse_word("and");
      }
      if (value >= 1000L) {         // Thousands
            say_number(value/1000L);
            parse_word("thousand");
            value = value % 1000L;
            if (value == 0)
                  return;                 // Even thousand
            if (value < 100)  // as in THREE THOUSAND AND FIVE
                  parse_word("and");
      }
      if (value >= 100L) {          // Hundreds
            parse_word(Cardinals[value/100]);
            parse_word("hundred");
            value = value % 100;
            if (value == 0)
                  return;           // Even hundred
            parse_word("and");
      }
      if (value >= 20) {
            parse_word(Twenties[(value-20)/ 10]);
            value = value % 10;
            if (value == 0)
                  return;           // Even ten
      }
      parse_word(Cardinals[value]);
      return;
}

//////////////////////////////////////////////////////////////////////
// extract_number takes the first numerical string out of the word
// string and sent the number string to say_number()
void extract_number(CString& word) {
      // TO DO: process a number including decimal point
      CString number;
      int ptr = 0;
      while (word.FindOneOf("0123456789") >= 0) {
            number += word.Left(1);
            word = word.Mid(1);
      }
      say_number(atol(number));
}

//////////////////////////////////////////////////////////////////////
// p_add_word is the equivalent of add_word, but works on the phoneme
// string returned by xlate_phrase().
void p_add_word(LPCTSTR w_in, LPCTSTR p_in)
{
      CString phon;
      char p, p2;
      int index = 0;
      int length;
      int phonemes = 0;
      int special_word = 0; // indicates an exception to the rule

      phon = p_in;
      length = phon.GetLength();
      words[word_ptr] = new CWordObj;
      words[word_ptr]->w_text = w_in;
      words[word_ptr]->w_phon = p_in;
      // now convert the dict phoneme string to numbers in the phoneme
      // array
      while (index < length && phonemes < (MAX_PHONEMES - 1)) {
            if (phon[index] > 64) {
                  p = p_lookup[phon[index] - 65];
                  if (phon[index] < 97) {
                        // the character is a capital
                        // and is the first of a 2 char phoneme
                        if (length > index + 1) {
                              p2 = phon[index + 1];
                              if (p == 57) {          // if I*
                                    if (p2 == 'Y')
                                          words[word_ptr]->
                                          phoneme[phonemes] = p;
                                    else if (p2 == 'H')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 62;
                              }
                              if (p == 58) {          // If A*
                                    if (p2 == 'A')
                                          words[word_ptr]->
                                          phoneme[phonemes] = p;
                                    else if (p2 == 'O')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 59;
                                    else if (p2 == 'H')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 65;
                                    else if (p2 == 'X')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 68;
                                    else if (p2 == 'Y')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 70;
                                    else if (p2 == 'W')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 73;
                                    else if (p2 == 'E')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 64;
                              }
                              if (p == 61) {          // if E*
                                    if (p2 == 'R')
                                          words[word_ptr]->
                                          phoneme[phonemes] = p;
                                    else if (p2 == 'H')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 63;
                                    else if (p2 == 'Y')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 69;
                              }
                              if (p == 67) {          // if U*
                                    if (p2 == 'H')
                                          words[word_ptr]->
                                          phoneme[phonemes] = p;
                                    else if (p2 == 'W')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 76;
                              }
                              if (p == 71) {          // if O*
                                    if (p2 == 'Y')
                                          words[word_ptr]->
                                          phoneme[phonemes] = p;
                                    else if (p2 == 'W')
                                          words[word_ptr]->
                                          phoneme[phonemes] = 72;
                              }
                              // if p was 46,47,49,51 to 55 or 60,
                              // then it doesn't matter what the
                              // second letter is.
                              else if (p == 46 || p == 47 || p == 49
                                    ||(p >= 51 && p <= 55) || p == 60)
                                    words[word_ptr]->
                                    phoneme[phonemes] = p;
                              index ++; // it was a 2 letter phoneme
                        }
                  }
                  else {
                        words[word_ptr]->phoneme[phonemes] = p;
                  }
                  if (words[word_ptr]->phoneme[phonemes] > 32) {
                        //phoneme is correct
                        words[word_ptr]->pitch[phonemes] = NORM_PITCH;
                        phonemes ++;
                  }
```

```
            }
            index ++;
        }
        words[word_ptr]->phonemes = phonemes;
        word_ptr ++;
}


//////////////////////////////////////////////////////////////////////
// mbr_convert_list works on the list of words and word attributes
// stored in the words array. word_ptr gives the current number of
// swords in the list. the function first calculates the overall pitch
// contour of the sentence, depending of the ending punctuation, and
// the start word.
// Then, each word is converted into an Mbrola string including pitch
// information for each phoneme. this is then added to the string:
// MBrola. If the MBrola string is too long, it is sent to MBR_Play()
// to synthesise.
void mbr_convert_list(void) {
        char duration[10];
        char s_pitch[10];
        int pitch;
        int dummy_ptr;
        int count, i;
        int total_phon = 0; // number of phonemes in sentence
        int num_phon = 0; // the current phoneme number
        char falling = 2;
        // (2 = no pitch variation. 0 = rising. 1 = falling.)
        CString word_text;

        if (MBrola.GetLength() == 0)
            MBrola = "_ 75 \n"; // starting phoneme
        word_text = words[0]->w_text;
        if (words[word_ptr-1]->w_text[0] == '!') {
            falling = 0;
        }
        if (words[word_ptr-1]->w_text[0] == '.') {
            falling = 1;
        }
        if (words[word_ptr-1]->w_text[0] == '?') {
            if (word_text.CompareNoCase("why") &&
                        word_text.CompareNoCase("where") &&
                        word_text.CompareNoCase("what") &&
                        word_text.CompareNoCase("how") &&
                        word_text.CompareNoCase("who") &&
                        word_text.CompareNoCase("when")) {
                falling = 0;
            }
            else {
                falling = 1;
            }
        }
        // if prosody turned off, then set contour to flat.
        if (prosody_flag == 0)
            falling = 2;
        // count the number of phonemes in the sentence
        for(dummy_ptr = 0;dummy_ptr < word_ptr;dummy_ptr ++) {
            total_phon += words[dummy_ptr]->phonemes;
        }

        for(dummy_ptr = 0;dummy_ptr < word_ptr;dummy_ptr ++) {
            if (words[dummy_ptr]->w_phon[0] == '_') {
                // straight map to mbrola
```

```
                        MBrola += words[dummy_ptr]->w_phon;
                        MBrola += " \n";
                  }
                  else {
                        for (count = 0; count < words[dummy_ptr]->phonemes;
                              count ++) {
                              num_phon ++;
                              i = words[dummy_ptr]->phoneme[count];
                              MBrola += mbr_phon[(i - 33) * 2];
                              MBrola += mbr_phon[(i - 33) * 2 + 1];
                              MBrola += " ";
                              MBrola += _itoa(phon_dur[i-33],duration, 10);
                              //pitch data is at the 50% position
                              MBrola += " 50 ";
                              if (falling == 2)
                                    pitch = N_PITCH;
                              else if (falling == 1) {
                                    pitch = F_PITCH_START -
                                          ((F_PITCH_START - F_PITCH_END)
                                          * num_phon / total_phon) +
                                          words[dummy_ptr]->pitch[count];
                              }
                              else {
                                    pitch = R_PITCH_START +
                                          ((R_PITCH_END - R_PITCH_START)
                                          * num_phon / total_phon) +
                                          words[dummy_ptr]->pitch[count];
                              }
                              MBrola += _itoa(pitch, s_pitch, 10);
                              MBrola += " \n";
                        }
                  }
            }

            // clear up the allocated blocks
            for (word_ptr --; word_ptr >= 0; word_ptr--)
                  delete words[word_ptr];
            word_ptr = 0;
            if (MBrola.GetLength() > 500)
                  play_list(); // play the buffer
}

//////////////////////////////////////////////////////////////////////
// play_list waits until MBrola has stopped synthesising, and then
// sends the MBrola string to the synthesiser before emtying it
// and returning.
void play_list(void)
{
      MBrola += "_ 100 \n# \n"; // end phoneme flushes buffer
      // wait until previous sentence is finished
      MBR_WaitForEnd();
      if (MBR_Play(MBrola, MBROUT_SOUNDBOARD, NULL, NULL) > 0)
            AfxMessageBox("Error speaking", MB_OK, 0);
      MBrola.Empty(); //empty the buffer
}
```

ENGLISH.H

```
// ENGLISH.H
//
// This file contains letter to sound rules for the English language.
//
```

```
//    English to Phoneme rules:
//
//    Derived from:
//
//         AUTOMATIC TRANSLATION OF ENGLISH TEXT TO PHONETICS
//              BY MEANS OF LETTER-TO-SOUND RULES
//
//                 NRL Report 7948
//
//                 January 21st, 1976
//        Naval Research Laboratory, Washington, D.C.
//
//
//    Published by the National Technical Information Service as
//    document "AD/A021 929".
//
//
//
//    The Phoneme codes:
//
//          IY    bEEt        IH    bIt
//          EY    gAte        EH    gEt
//          AE    fAt         AA    fAther
//          AO    lAWn        OW    lOne
//          UH    fUll        UW    fOOl
//          ER    mURdER            AX    About
//          AH    bUt         AY    hIde
//          AW    hOW         OY    tOY
//          YU    YOU
//
//          p     Pack        b     Back
//          t     Time        d     Dime
//          k     Coat        g     Goat
//          f     Fault       v     Vault
//          TH    eTHer       DH    eiTHer
//          s     Sue         z     Zoo
//          SH    leaSH       ZH    leiSure
//          HH    How         m     suM
//          n     suN         NG    suNG
//          l     Laugh       w     Wear
//          y     Young       r     Rate
//          CH    CHar        j     Jar
//          WH    WHere
//
//
//    Rules are made up of four parts:
//
//          The left context.
//          The text to match.
//          The right context.
//          The phonemes to substitute for the matched text.
//
//    Procedure:
//
//          Seperate each block of letters (apostrophes included)
//          and add a space on each side.  For each unmatched
//          letter in the word, look through the rules where the
//          text to match starts with the letter in the word.  If
//          the text to match is found and the right and left
//          context patterns also match, output the phonemes for
//          that rule and skip to the next unmatched letter.
//
//
```

```
//    Special Context Symbols:
//
//          #     One or more vowels
//          :     Zero or more consonants
//          ^     One consonant.
//          .     One of B, D, V, G, J, L, M, N, R, W or Z (voiced
//                consonants)
//          %     One of ER, E, ES, ED, ING, ELY (a suffix)
//                (Found in right context only)
//          +     One of E, I or Y (a "front" vowel)
//



// Context definitions
static char Anything[] = "";    // No context requirement
static char Nothing[] = " ";    // Context is beginning or end of word

// Phoneme definitions
static char Pause[] = " ";      // Short silence
static char Silent[] = "";      // No phonemes

#define LEFT_PART 0
#define MATCH_PART      1
#define RIGHT_PART      2
#define OUT_PART  3

typedef char *PRule[4]; // Rule is an array of 4 character pointers

// 0 = Punctuation

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule punct_rules[] =
      {
      {Anything,  " ",            "'",            Silent  },
      {Anything,  " ",            Anything,       Pause   },
      {Anything,  "-",            Anything,       Silent  },
      {".",       "'S",           Anything,       "z"     },
      {"#:.E",    "'S",           Anything,       "z"     },
      {"#",       "'S",           Anything,       "z"     },
      {Anything,  "'",            Anything,       Silent  },
      {Anything,  ",",            Anything,       Pause   },
      {Anything,  ".",            Anything,       Pause   },
      {Anything,  "?",            Anything,       Pause   },
      {Anything,  "!",            Anything,       Pause   },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule A_rules[] =
      {
      {Anything,  "A",            Nothing,        "AX"    },
      {Nothing,   "ARE",          Nothing,        "AAr"   },
      {Nothing,   "AR",           "O",            "AXr"   },
      {Anything,  "AR",           "#",            "EHr"   },
      {"^",       "AS",           "#",            "EYs"   },
      {Anything,  "A",            "WA",           "AX"    },
      {Anything,  "AW",           Anything,       "AO"    },
      {" :",      "ANY",          Anything,       "EHnIY" },
      {Anything,  "A",            "^+#",          "EY"    },
      {"#:",      "ALLY",         Anything,       "AXlIY" },
      {Nothing,   "AL",           "#",            "AXl"   },
      {Anything,  "AGAIN",        Anything,       "AXgEHn"},
      {"#:",      "AG",           "E",            "IHj"   },
      {Anything,  "A",            "^+:#",         "AE"    },
      {" :",      "A",            "^+ ",          "EY"    },
      {Anything,  "A",            "^%",           "EY"    },
      {Nothing,   "ARR",          Anything,       "AXr"   },
      {Anything,  "ARR",          Anything,       "AEr"   },
      {" :",      "AR",           Nothing,        "AAr"   },
      {Anything,  "AR",           Nothing,        "ER"    },
      {Anything,  "AR",           Anything,       "AAr"   },
      {Anything,  "AIR",          Anything,       "EHr"   },
      {Anything,  "AI",           Anything,       "EY"    },
      {Anything,  "AY",           Anything,       "EY"    },
      {Anything,  "AU",           Anything,       "AO"    },
      {"#:",      "AL",           Nothing,        "AXl"   },
      {"#:",      "ALS",          Nothing,        "AXlz"  },
      {Anything,  "ALK",          Anything,       "AOk"   },
      {Anything,  "AL",           "^",            "AOl"   },
      {" :",      "ABLE",         Anything,       "EYbAXl"},
      {Anything,  "ABLE",         Anything,       "AXbAXl"},
      {Anything,  "ANG",          "+",            "EYnj"  },
      {Anything,  "A",            Anything,       "AE"    },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule B_rules[] =
      {
      {Nothing,   "BE",           "^#",           "bIH"   },
      {Anything,  "BEING",        Anything,       "bIYIHNG"},
      {Nothing,   "BOTH",         Nothing,        "bOWTH" },
      {Nothing,   "BUS",          "#",            "bIHz"  },
      {Anything,  "BUIL",         Anything,       "bIHl"  },
      {Anything,  "B",            Anything,       "b"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule C_rules[] =
      {
      {Nothing,   "CH",           "^",            "k"     },
      {"^E",      "CH",           Anything,       "k"     },
      {Anything,  "CH",           Anything,       "CH"    },
      {" S",      "CI",           "#",            "sAY"   },
      {Anything,  "CI",           "A",            "SH"    },
      {Anything,  "CI",           "O",            "SH"    },
      {Anything,  "CI",           "EN",           "SH"    },
      {Anything,  "C",            "+",            "s"     },
      {Anything,  "CK",           Anything,       "k"     },
      {Anything,  "COM",          "%",            "kAHm"  },
      {Anything,  "C",            Anything,       "k"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule D_rules[] =
      {
      {"#:",      "DED",          Nothing,        "dIHd"  },
      {".E",      "D",            Nothing,        "d"     },
      {"#^:E",    "D",            Nothing,        "t"     },
      {Nothing,   "DE",           "^#",           "dIH"   },
      {Nothing,   "DO",           Nothing,        "dUW"   },
      {Nothing,   "DOES",         Anything,       "dAHz"  },
      {Nothing,   "DOING",        Anything,       "dUWIHNG"},
      {Nothing,   "DOW",          Anything,       "dAW"   },
      {Anything,  "DU",           "A",            "jUW"   },
      {Anything,  "D",            Anything,       "d"     },
      };
```

```
//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule E_rules[] =
      {
      {"#:",      "E",            Nothing,        Silent  },
      {"'^:",     "E",            Nothing,        Silent  },
      {" :",      "E",            Nothing,        "IY"    },
      {"#",       "ED",           Nothing,        "d"     },
      {"#:",      "E",            "D ",           Silent  },
      {Anything,  "EV",           "ER",           "EHv"   },
      {Anything,  "E",            "^%",           "IY"    },
      {Anything,  "ERI",          "#",            "IYrIY" },
      {Anything,  "ERI",          Anything,       "EHrIH" },
      {"#:",      "ER",           "#",            "ER"    },
      {Anything,  "ER",           "#",            "EHr"   },
      {Anything,  "ER",           Anything,       "ER"    },
      {Nothing,   "EVEN",         Anything,       "IYvEHn"},
      {"#:",      "E",            "W",            Silent  },
      {"T",       "EW",           Anything,       "UW"    },
      {"S",       "EW",           Anything,       "UW"    },
      {"R",       "EW",           Anything,       "UW"    },
      {"D",       "EW",           Anything,       "UW"    },
      {"L",       "EW",           Anything,       "UW"    },
      {"Z",       "EW",           Anything,       "UW"    },
      {"N",       "EW",           Anything,       "UW"    },
      {"J",       "EW",           Anything,       "UW"    },
      {"TH",      "EW",           Anything,       "UW"    },
      {"CH",      "EW",           Anything,       "UW"    },
      {"SH",      "EW",           Anything,       "UW"    },
      {Anything,  "EW",           Anything,       "YUw"   },
      {Anything,  "E",            "O",            "IY"    },
      {"#:S",     "ES",           Nothing,        "IHz"   },
      {"#:C",     "ES",           Nothing,        "IHz"   },
      {"#:G",     "ES",           Nothing,        "IHz"   },
      {"#:Z",     "ES",           Nothing,        "IHz"   },
      {"#:X",     "ES",           Nothing,        "IHz"   },
      {"#:J",     "ES",           Nothing,        "IHz"   },
      {"#:CH",    "ES",           Nothing,        "IHz"   },
      {"#:SH",    "ES",           Nothing,        "IHz"   },
      {"#:",      "E",            "S ",           Silent  },
      {"#:",      "ELY",          Nothing,        "lIY"   },
      {"#:",      "EMENT",        Anything,       "mEHnt" },
      {Anything,  "EFUL",         Anything,       "fUHl"  },
      {Anything,  "EE",           Anything,       "IY"    },
      {Anything,  "EARN",         Anything,       "ERn"   },
      {Nothing,   "EAR",          "^",            "ER"    },
      {Anything,  "EAD",          Anything,       "EHd"   },
      {"#:",      "EA",           Nothing,        "IYAX"  },
      {Anything,  "EA",           "SU",           "EH"    },
      {Anything,  "EA",           Anything,       "IY"    },
      {Anything,  "EIGH",         Anything,       "EY"    },
      {Anything,  "EI",           Anything,       "IY"    },
      {Nothing,   "EYE",          Anything,       "AY"    },
      {Anything,  "EY",           Anything,       "IY"    },
      {Anything,  "EU",           Anything,       "YUw"   },
      {Anything,  "E",            Anything,       "EH"    },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule F_rules[] =
      {
      {Anything,  "FUL",          Anything,       "fUHl"  },
      {Anything,  "F",            Anything,       "f"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule G_rules[] =
      {
      {Anything,  "GIV",          Anything,       "gIHv"  },
      {Nothing,   "G",            "I^",           "g"     },
      {Anything,  "GE",           "T",            "gEH"   },
      {"SU",      "GGES",         Anything,       "gjEHs" },
      {Anything,  "GG",           Anything,       "g"     },
      {" B#",     "G",            Anything,       "g"     },
      {Anything,  "G",            "+",            "j"     },
      {Anything,  "GREAT",        Anything,       "grEYt" },
      {"#",       "GH",           Anything,       Silent  },
      {Anything,  "G",            Anything,       "g"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule H_rules[] =
      {
      {Nothing,   "HAV",          Anything,       "hAEv"  },
      {Nothing,   "HERE",         Anything,       "hIYr"  },
      {Nothing,   "HOUR",         Anything,       "AWER"  },
      {Anything,  "HOW",          Anything,       "hAW"   },
      {Anything,  "H",            "#",            "h"     },
      {Anything,  "H",            Anything,       Silent  },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule I_rules[] =
      {
      {Nothing,   "IN",           Anything,       "IHn"   },
      {Nothing,   "I",            Nothing,        "AY"    },
      {Anything,  "IN",           "D",            "AYn"   },
      {Anything,  "IER",          Anything,       "IYER"  },
      {"#:R",     "IED",          Anything,       "IYd"   },
      {Anything,  "IED",          Nothing,        "AYd"   },
      {Anything,  "IEN",          Anything,       "IYEHn" },
      {Anything,  "IE",           "T",            "AYEH"  },
      {" :",      "I",            "%",            "AY"    },
      {Anything,  "I",            "%",            "IY"    },
      {Anything,  "IE",           Anything,       "IY"    },
      {Anything,  "I",            "^+:#",         "IH"    },
      {Anything,  "IR",           "#",            "AYr"   },
      {Anything,  "IZ",           "%",            "AYz"   },
      {Anything,  "IS",           "%",            "AYz"   },
      {Anything,  "I",            "D%",           "AY"    },
      {"+^",      "I",            "^+",           "IH"    },
      {Anything,  "I",            "T%",           "AY"    },
      {"#^:",     "I",            "^+",           "IH"    },
      {Anything,  "I",            "^+",           "AY"    },
      {Anything,  "IR",           Anything,       "ER"    },
      {Anything,  "IGH",          Anything,       "AY"    },
      {Anything,  "ILD",          Anything,       "AYld"  },
      {Anything,  "IGN",          Nothing,        "AYn"   },
      {Anything,  "IGN",          "^",            "AYn"   },
      {Anything,  "IGN",          "%",            "AYn"   },
      {Anything,  "IQUE",         Anything,       "IYk"   },
      {Anything,  "I",            Anything,       "IH"    },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule J_rules[] =
      {
      {Anything,  "J",            Anything,       "j"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule K_rules[] =
      {
      {Nothing,   "K",            "N",            Silent  },
      {Anything,  "K",            Anything,       "k"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule L_rules[] =
      {
      {Anything,  "LO",           "C#",           "lOW"   },
      {"L",       "L",            Anything,       Silent  },
      {"#^:",     "L",            "%",            "AXl"   },
      {Anything,  "LEAD",         Anything,       "lIYd"  },
      {Anything,  "L",            Anything,       "l"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule M_rules[] =
      {
      {Anything,  "MOV",          Anything,       "mUWv"  },
      {Anything,  "M",            Anything,       "m"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule N_rules[] =
      {
      {"E",       "NG",           "+",            "nj"    },
      {Anything,  "NG",           "R",            "NGg"   },
      {Anything,  "NG",           "#",            "NGg"   },
      {Anything,  "NGL",          "%",            "NGgAXl"},
      {Anything,  "NG",           Anything,       "NG"    },
      {Anything,  "NK",           Anything,       "NGk"   },
      {Nothing,   "NOW",          Nothing,        "nAW"   },
      {Anything,  "N",            Anything,       "n"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule O_rules[] =
      {
      {Anything,  "OF",           Nothing,        "AXv"   },
      {Anything,  "OROUGH",       Anything,       "EROW"  },
      {"#:",      "OR",           Nothing,        "ER"    },
      {"#:",      "ORS",          Nothing,        "ERz"   },
      {Anything,  "OR",           Anything,       "AOr"   },
      {Nothing,   "ONE",          Anything,       "wAHn"  },
      {Anything,  "OW",           Anything,       "OW"    },
      {Nothing,   "OVER",         Anything,       "OWvER" },
      {Anything,  "OV",           Anything,       "AHv"   },
      {Anything,  "O",            "^%",           "OW"    },
      {Anything,  "O",            "^EN",          "OW"    },
      {Anything,  "O",            "^I#",          "OW"    },
      {Anything,  "OL",           "D",            "OWl"   },
      {Anything,  "OUGHT",        Anything,       "AOt"   },
      {Anything,  "OUGH",         Anything,       "AHf"   },
      {Nothing,   "OU",           Anything,       "AW"    },
      {"H",       "OU",           "S#",           "AW"    },
      {Anything,  "OUS",          Anything,       "AXs"   },
      {Anything,  "OUR",          Anything,       "AOr"   },
      {Anything,  "OULD",         Anything,       "UHd"   },
      {"^",       "OU",           "^L",           "AH"    },
      {Anything,  "OUP",          Anything,       "UWp"   },
      {Anything,  "OU",           Anything,       "AW"    },
      {Anything,  "OY",           Anything,       "OY"    },
      {Anything,  "OING",         Anything,       "OWIHNG"},
      {Anything,  "OI",           Anything,       "OY"    },
      {Anything,  "OOR",          Anything,       "AOr"   },
      {Anything,  "OOK",          Anything,       "UHk"   },
      {Anything,  "OOD",          Anything,       "UHd"   },
      {Anything,  "OO",           Anything,       "UW"    },
      {Anything,  "O",            "E",            "OW"    },
      {Anything,  "O",            Nothing,        "OW"    },
      {Anything,  "OA",           Anything,       "OW"    },
      {Nothing,   "ONLY",         Anything,       "OWnlIY"},
      {Nothing,   "ONCE",         Anything,       "wAHns" },
      {Anything,  "ON'T",         Anything,       "OWnt"  },
      {"C",       "O",            "N",            "AA"    },
      {Anything,  "O",            "NG",           "AO"    },
      {"^:",      "O",            "N",            "AH"    },
      {"I",       "ON",           Anything,       "AXn"   },
      {"#:",      "ON",           Nothing,        "AXn"   },
      {"#^",      "ON",           Anything,       "AXn"   },
      {Anything,  "O",            "ST ",          "OW"    },
      {Anything,  "OF",           "^",            "AOf"   },
      {Anything,  "OTHER",        Anything,       "AHDHER"},
      {Anything,  "OSS",          Nothing,        "AOs"   },
      {"#^:",     "OM",           Anything,       "AHm"   },
      {Anything,  "O",            Anything,       "AA"    },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule P_rules[] =
      {
      {Anything,  "PH",           Anything,       "f"     },
      {Anything,  "PEOP",         Anything,       "pIYp"  },
      {Anything,  "POW",          Anything,       "pAW"   },
      {Anything,  "PUT",          Nothing,        "pUHt"  },
      {Anything,  "P",            Anything,       "p"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule Q_rules[] =
      {
      {Anything,  "QUAR",         Anything,       "kwAOr" },
      {Anything,  "QU",           Anything,       "kw"    },
      {Anything,  "Q",            Anything,       "k"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule R_rules[] =
      {
      {Nothing,   "RE",           "^#",           "rIY"   },
      {Anything,  "R",            Anything,       "r"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule S_rules[] =
      {
      {Anything,  "SH",           Anything,       "SH"    },
      {"#",       "SION",         Anything,       "ZHAXn" },
      {Anything,  "SOME",         Anything,       "sAHm"  },
      {"#",       "SUR",          "#",            "ZHER"  },
      {Anything,  "SUR",          "#",            "SHER"  },
      {"#",       "SU",           "#",            "ZHUW"  },
      {"#",       "SSU",          "#",            "SHUW"  },
      {"#",       "SED",          Nothing,        "zd"    },
      {"#",       "S",            "#",            "z"     },
      {Anything,  "SAID",         Anything,       "sEHd"  },
      {"^",       "SION",         Anything,       "SHAXn" },
      {Anything,  "S",            "S",            Silent  },
      {".",       "S",            Nothing,        "z"     },
      {"#:.E",    "S",            Nothing,        "z"     },
      {"#^:##",   "S",            Nothing,        "z"     },
      {"#^:#",    "S",            Nothing,        "s"     },
      {"U",       "S",            Nothing,        "s"     },
      {" :#",     "S",            Nothing,        "z"     },
      {Nothing,   "SCH",          Anything,       "sk"    },
      {Anything,  "S",            "C+",           Silent  },
      {"#",       "SM",           Anything,       "zm"    },
      {"#",       "SN",           "'",            "zAXn"  },
      {Anything,  "S",            Anything,       "s"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule T_rules[] =
      {
      {Nothing,   "THE",          Nothing,        "DHAX"  },
      {Anything,  "TO",           Nothing,        "tUW"   },
      {Anything,  "THAT",         Nothing,        "DHAEt" },
      {Nothing,   "THIS",         Nothing,        "DHIHs" },
      {Nothing,   "THEY",         Anything,       "DHEY"  },
      {Nothing,   "THERE",        Anything,       "DHEHr" },
      {Anything,  "THER",         Anything,       "DHER"  },
      {Anything,  "THEIR",        Anything,       "DHEHr" },
      {Nothing,   "THAN",         Nothing,        "DHAEn" },
      {Nothing,   "THEM",         Nothing,        "DHEHm" },
      {Anything,  "THESE",        Nothing,        "DHIYz" },
      {Nothing,   "THEN",         Anything,       "DHEHn" },
      {Anything,  "THROUGH",      Anything,       "THrUW" },
      {Anything,  "THOSE",        Anything,       "DHOWz" },
      {Anything,  "THOUGH",       Nothing,        "DHOW"  },
      {Nothing,   "THUS",         Anything,       "DHAHs" },
      {Anything,  "TH",           Anything,       "TH"    },
      {"#:",      "TED",          Nothing,        "tIHd"  },
      {"S",       "TI",           "#N",           "CH"    },
      {Anything,  "TI",           "O",            "SH"    },
      {Anything,  "TI",           "A",            "SH"    },
      {Anything,  "TIEN",         Anything,       "SHAXn" },
      {Anything,  "TUR",          "#",            "CHER"  },
      {Anything,  "TU",           "A",            "CHUW"  },
      {Nothing,   "TWO",          Anything,       "tUW"   },
      {Anything,  "T",            Anything,       "t"     },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule U_rules[] =
      {
      {Nothing,   "UN",           "I",            "yUWn"  },
      {Nothing,   "UN",           Anything,       "AHn"   },
      {Nothing,   "UPON",         Anything,       "AXpAOn"},
      {"T",       "UR",           "#",            "UHr"   },
      {"S",       "UR",           "#",            "UHr"   },
      {"R",       "UR",           "#",            "UHr"   },
      {"D",       "UR",           "#",            "UHr"   },
      {"L",       "UR",           "#",            "UHr"   },
      {"Z",       "UR",           "#",            "UHr"   },
      {"N",       "UR",           "#",            "UHr"   },
      {"J",       "UR",           "#",            "UHr"   },
      {"TH",      "UR",           "#",            "UHr"   },
      {"CH",      "UR",           "#",            "UHr"   },
      {"SH",      "UR",           "#",            "UHr"   },
      {Anything,  "UR",           "#",            "yUHr"  },
      {Anything,  "UR",           Anything,       "ER"    },
      {Anything,  "U",            "^ ",           "AH"    },
      {Anything,  "U^^",          Anything,       "AH"    },
      {Anything,  "UY",           Anything,       "AY"    },
      {" G",      "U",            "#",            Silent  },
      {"G",       "U",            "%",            Silent  },
      {"G",       "U",            "#",            "w"     },
      {"#N",      "U",            Anything,       "YUw"   },
      {"T",       "U",            Anything,       "UW"    },
      {"S",       "U",            Anything,       "UW"    },
      {"R",       "U",            Anything,       "UW"    },
      {"D",       "U",            Anything,       "UW"    },
      {"L",       "U",            Anything,       "UW"    },
      {"Z",       "U",            Anything,       "UW"    },
      {"N",       "U",            Anything,       "UW"    },
      {"J",       "U",            Anything,       "UW"    },
      {"TH",      "U",            Anything,       "UW"    },
      {"CH",      "U",            Anything,       "UW"    },
      {"SH",      "U",            Anything,       "UW"    },
      {Anything,  "U",            Anything,       "YUw"   },
      };

//      LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule V_rules[] =
      {
      {Anything,  "VIEW",         Anything,       "vYUw"  },
      {Anything,  "V",            Anything,       "v"     },
      };

//      LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule W_rules[] =
      {
      {Nothing,   "WERE",         Anything,       "wER"   },
      {Anything,  "WA",           "S",            "wAA"   },
      {Anything,  "WA",           "T",            "wAA"   },
      {Anything,  "WERE",         Anything,       "WHEHr" },
      {Anything,  "WHAT",         Anything,       "WHAAt" },
      {Anything,  "WHOL",         Anything,       "hOWl"  },
      {Anything,  "WHO",          Anything,       "hUW"   },
      {Anything,  "WH",           Anything,       "WH"    },
      {Anything,  "WAR",          Anything,       "wAOr"  },
      {Anything,  "WOR",          "^",            "wER"   },
      {Anything,  "WR",           Anything,       "r"     },
      {Anything,  "W",            Anything,       "w"     },
      };

//      LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule X_rules[] =
      {
      {Anything,  "X",            Anything,       "ks"    },
      };

//      LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule Y_rules[] =
      {
      {Anything,  "YOUNG",        Anything,       "yAHNG" },
      {Nothing,   "YOU",          Anything,       "yUW"   },
      {Nothing,   "YES",          Anything,       "yEHs"  },
      {Nothing,   "Y",            Anything,       "y"     },
      {"#^:",     "Y",            Nothing,        "IY"    },
      {"#^:",     "Y",            "I",            "IY"    },
      {" :",      "Y",            Nothing,        "AY"    },
      {" :",      "Y",            "#",            "AY"    },
      {" :",      "Y",            "^+:#",         "IH"    },
      {" :",      "Y",            "^#",           "AY"    },
      {Anything,  "Y",            Anything,       "IH"    },
      };

//    LEFT_PART   MATCH_PART  RIGHT_PART  OUT_PART
static PRule Z_rules[] =
      {
      {Anything,  "Z",            Anything,       "z"     },
      };

PRule *Rules[] =
      {
      punct_rules,
      A_rules, B_rules, C_rules, D_rules, E_rules, F_rules, G_rules,
      H_rules, I_rules, J_rules, K_rules, L_rules, M_rules, N_rules,
      O_rules, P_rules, Q_rules, R_rules, S_rules, T_rules, U_rules,
      V_rules, W_rules, X_rules, Y_rules, Z_rules
      };
```

PHONEME.H

```
// PHONEME.H
//
// This is a header file which contains prototypes for functions
// used in PHONEME.CPP. It also includes the letter to sound rules
// in ENGLISH.H

#include "english.h"

#define MAX_LENGTH 128 //max length of words to be converted
#define P_FALSE (0)
#define P_TRUE (!0)
#define EOW '\0' //end of word

int makeupper(int character);
int isvowel(char chr);
int isconsonant(char chr);
void xlate_word(char *word);
void xlate_phrase(LPCTSTR text_word, CString& ret_phon);
int find_rule(char *word, int index, PRule *rules);
char leftmatch(char *pattern, char *context);
char rightmatch(char *pattern, char *context);
void have_letter(void);
void new_char(void);
void outstring(char *op);
```

PHONEME.CPP

```
// PHONEME.CPP
//
// this file contains the functions used to convert a word in
// text form into a string of phonemes
//
```

```
//    English to Phoneme translation.
//
//    Rules are made up of four parts:
//
//          The left context.
//          The text to match.
//          The right context.
//          The phonemes to substitute for the matched text.
//
//    Procedure:
//
//          Seperate each block of letters (apostrophes included)
//          and add a space on each side.  For each unmatched
//          letter in the word, look through the rules where the
//          text to match starts with the letter in the word.  If
//          the text to match is found and the right and left
//          context patterns also match, output the phonemes for
//          that rule and skip to the next unmatched letter.
//
//
//    Special Context Symbols:
//
//          #     One or more vowels
//          :     Zero or more consonants
//          ^     One consonant.
//          .     One of B, D, V, G, J, L, M, N, R, W or Z (voiced
//                consonants)
//          %     One of ER, E, ES, ED, ING, ELY (a suffix)
//                (Right context only)
//          +     One of E, I or Y (a "front" vowel)

#include <
```

```
{
      return (chr == 'A' || chr == 'E' || chr == 'I' ||
            chr == 'O' || chr == 'U');
}

//////////////////////////////////////////////////////////////////////
// returns a boolean value, FALSE for a vowel, and TRUE for a
// consonant.
int isconsonant(char chr)
{
      return (isupper(chr) && !isvowel(chr));
}

//////////////////////////////////////////////////////////////////////
// xlate_word is called from the hane_letter function
// It turns turns the parsed word into phonemes by finding rules
// that match the letter patterns of the word.
void xlate_word(char *word)
{
      int index;  // Current position in word
      int type;   // First letter of match part

      index = 1;  // Skip the initial blank
      do {
            if (isupper(word[index]))
                  type = word[index] - 'A' + 1;
            else
                  type = 0;
            index = find_rule(word, index, Rules[type]);
      } while (word[index] != '\0');
}

//////////////////////////////////////////////////////////////////////
// find_rule tries to find a rule that matches the current set of
// letters that are being looked at in the word.
int find_rule(char *word, int index, PRule *rules)
{
      PRule *rule;
      char *left, *match, *right, *output;
      int remainder;

      // Search for the rule
      for (;;) {
            rule = rules++;
            match = (*rule)[1];

            if (match == 0)   { // bad symbol!
                  return index+1;   // Skip it!
            }
            for (remainder = index; *match != '\0'; match++,
                  remainder++) {
                  if (*match != word[remainder])
                        break;
            }
            if (*match != '\0')     // found missmatch
                  continue;
            left = (*rule)[0];
            right = (*rule)[2];
            if (!leftmatch(left, &word[index-1]))
                  continue;
            if (!rightmatch(right, &word[remainder]))
                  continue;
            output = (*rule)[3];
            outstring(output);
            return remainder;
      }
}


//////////////////////////////////////////////////////////////////////
// leftmatch finds patterns which match the left end of the word.
char leftmatch(char *pattern, char *context)
{
      char *pat;
      char *text;
      int count;

      if (*pattern == '\0') { // null string matches any context
            return P_TRUE;
      }

      // point to last character in pattern string
      count = strlen(pattern);
      pat = pattern + (count - 1);

      text = context;

      for (; count > 0; pat--, count--) {
            // First check for simple text or space
            if (isalpha(*pat) || *pat == '\'' || *pat == ' ')
                  if (*pat != *text)
                        return P_FALSE;
                  else {
                        text--;
                        continue;
                  }
            switch (*pat) {
            case '#':   // One or more vowels
                  if (!isvowel(*text))
                        return P_FALSE;

                  text--;

                  while (isvowel(*text))
                        text--;
                  break;

            case ':':   // Zero or more consonants
                  while (isconsonant(*text))
                        text--;
                  break;

            case '^':   // One consonant
                  if (!isconsonant(*text))
                        return P_FALSE;
                  text--;
                  break;

            case '.':   // B, D, V, G, J, L, M, N, R, W, Z
                  if (*text != 'B' && *text != 'D' && *text != 'V'
                     && *text != 'G' && *text != 'J' && *text != 'L'
                     && *text != 'M' && *text != 'N' && *text != 'R'
                     && *text != 'W' && *text != 'Z')
                        return P_FALSE;
                  text--;
                  break;
```

```
        case '+':   // E, I or Y (front vowel)
              if (*text != 'E' && *text != 'I' && *text != 'Y')
                    return P_FALSE;
              text--;
              break;

        case '%':
              return P_FALSE;
        }
  }
  return P_TRUE;
}

//////////////////////////////////////////////////////////////////////
// rightmatch finds patterns which match the right end of the word.
char rightmatch(char *pattern, char *context)
{
      char *pat;
      char *text;

      if (*pattern == '\0')   // null string matches any context
            return P_TRUE;
      pat = pattern;
      text = context;
      for (pat = pattern; *pat != '\0'; pat++) {
            // First check for simple text or space
            if (isalpha(*pat) || *pat == '\'' || *pat == ' ')
                  if (*pat != *text)
                        return P_FALSE;
                  else {
                        text++;
                        continue;
                  }
            switch (*pat) {
            case '#':   // One or more vowels
                  if (!isvowel(*text))
                        return P_FALSE;
                  text++;
                  while (isvowel(*text))
                        text++;
                  break;
            case ':':   // Zero or more consonants
                  while (isconsonant(*text))
                        text++;
                  break;
            case '^':   // One consonant
                  if (!isconsonant(*text))
                        return P_FALSE;
                  text++;
                  break;
            case '.':   // B, D, V, G, J, L, M, N, R, W, Z
                  if (*text != 'B' && *text != 'D' && *text != 'V'
                     && *text != 'G' && *text != 'J' && *text != 'L'
                     && *text != 'M' && *text != 'N' && *text != 'R'
                     && *text != 'W' && *text != 'Z')
                        return P_FALSE;
                  text++;
                  break;
            case '+':   // E, I or Y (front vowel)
                  if (*text != 'E' && *text != 'I' && *text != 'Y')
                        return P_FALSE;
                  text++;
                  break;
            case '%':   // ER, E, ES, ED, ING, ELY (a suffix)
                  if (*text == 'E') {
                        text++;
                        if (*text == 'L') {
                              text++;
                              if (*text == 'Y') {
                                    text++;
                                    break;
                              }
                              else {
                                    text--; // Don't gobble L
                                    break;
                              }
                        }
                        else if (*text == 'R' || *text == 'S'
                           || *text == 'D')
                              text++;
                        break;
                  }
                  else if (*text == 'I') {
                        text++;
                        if (*text == 'N') {
                              text++;
                              if (*text == 'G') {
                                    text++;
                                    break;
                              }
                        }
                        return P_FALSE;
                  }
                  else
                        return P_FALSE;
            default:
                  return P_FALSE;
            }
      }
      return P_TRUE;
}

//////////////////////////////////////////////////////////////////////
// have_letter processes the word, assuming the word contains letters
// Then it sends it to xlate_word.
void have_letter(void)
{
      char buff[MAX_LENGTH];
      int count;

      count = 0;
      buff[count++] = ' ';    // Required initial blank
      buff[count++] = makeupper(Char);
      for (new_char() ; isalpha(Char) || Char == '\'' ; new_char()) {
            buff[count++] = makeupper(Char);
            if (count > MAX_LENGTH-2) {
                  buff[count++] = ' ';
                  buff[count++] = '\0';
                  xlate_word(buff);
                  count = 1;
            }
      }
      buff[count++] = ' ';    // Required terminating blank
      buff[count++] = '\0';
      xlate_word(buff);
      if (Char == '-' && isalpha(Char1))
            new_char(); // Skip hyphens
}

//////////////////////////////////////////////////////////////////////
// new_char gets the next char of the word and pumps the last chars
// down through char1, char2, and char3
void new_char(void)
{
      // If the cache is full of newline, time to prime the look-ahead
      // again.  If an EOW is found, fill the remainder of the queue
      // with EOW's.
      if (Char == '\n'  && Char1 == '\n' && Char2 == '\n' &&
            Char3 == '\n') {
            // prime the pump again
            if (the_word.GetLength() == 0) {
                  Char = EOW;
                  Char1 = EOW;
                  Char2 = EOW;
                  Char3 = EOW;
                  return;
            }
            Char = the_word[0];
            the_word = the_word.Mid(1);
            if (the_word.GetLength() == 0) {
                  Char1 = EOW;
                  Char2 = EOW;
                  Char3 = EOW;
                  return;
            }
            if (Char == '\n')
                  return;
            Char1 = the_word[0];
            the_word = the_word.Mid(1);
            if (the_word.GetLength() == 0) {
                  Char2 = EOW;
                  Char3 = EOW;
                  return;
            }
            if (Char1 == '\n')
                  return;
            Char2 = the_word[0];
            the_word = the_word.Mid(1);
            if (the_word.GetLength() == 0) {
                  Char3 = EOW;
                  return;
            }
            if (Char2 == '\n')
                  return;
            Char3 = the_word[0];
            the_word = the_word.Mid(1);
      }
      else {
            // Buffer not full of newline, shuffle the characters and
            // either get a new one or propagate a newline or EOW.
            Char = Char1;
            Char1 = Char2;
            Char2 = Char3;
            if (the_word.GetLength() > 0) {
                  Char3 = the_word[0];
                  the_word = the_word.Mid(1);
            }
            else
```

```
                Char3 = EOW;
        }
        return;
}

//////////////////////////////////////////////////////////////////////
// xlate_phrase is the function called from TEXT.CPP to convert the
// word into phoneme form.
void xlate_phrase(LPCTSTR text_word, CString& ret_phon)
{
        // Prime the queue
        Char = '\n';
        Char1 = '\n';
        Char2 = '\n';
        Char3 = '\n';
        the_phon.Empty();
        the_word = text_word;
        new_char(); // Fill Char, Char1, Char2 and Char3
        have_letter();
        ret_phon = the_phon;
}

//////////////////////////////////////////////////////////////////////
// this function is called when another phoneme is added to the
// phoneme representation of the word.
void outstring(char *op)
{
        the_phon += op;
}
```

MBRPLAY.H

```
// MBRPLAY.H
//
// This file contains the prototypes and defines for MBRPLAY.LIB

/*
 * FPMs-TCTS SOFTWARE LIBRARY
 *
 * File :    folderdialog.cpp
 * Purpose : MbrPlay functions & constants defines
 * Author  : Alain Ruelle
 * Email   : ruelle@tcts.fpms.ac.be
 *
 * Copyright (c) 1997-1998 Faculte Polytechnique de Mons (TCTS lab)
 * All rights reserved.
 *
 * Statically imported functions (through a .lib file)
 *
 */

#ifndef __MBRPLAY_H__
#define __MBRPLAY_H__

// MBR_Play & MBR_PlayToFile Flags
#define MBR_MSGINIT                     1
#define MBR_MSGREAD                     2
#define MBR_MSGWAIT                     4
#define MBR_MSGWRITE                    8
#define MBR_MSGEND                      16
#define MBR_MSGALL                      31
#define MBR_BYFILE                      32
```

```
#define MBR_WAIT                          64
#define MBR_CALLBACK                      128
#define MBR_QUEUED                        256
#define MBR_ASPHS                         512

// MBR_SetOutputMode & MBR_GetOutputMode flags
#define MBROUT_SOUNDBOARD                 0
#define MBROUT_RAW                        1024
#define MBROUT_WAVE                       2048
#define MBROUT_AU                         4096
#define MBROUT_AIFF                       8192

#define MBROUT_ALAW                       16384
#define MBROUT_MULAW                      32768

// Mbrola Errors
#define MBRERR_NOREGISTRY           -13         // Registry keys error
#define MBRERR_NOMBROLADLL          -12         // Mbrola DLL not
found
#define MBRERR_DBINIT               -11         // No database loaded
#define MBRERR_WARNING              -10
// Information errors
#define MBRERR_CANTOPENWAVEOUT      -9
// Can't open the wavout device
#define MBRERR_CANTOPENFILEOUT      -8
#define MBRERR_CANTOPENFILE         -7
#define MBRERR_ERRORSPEAKING        -6    // Error while Speaking
#define MBRERR_DBNOTDATABASE        -5    // Not a valid database
#define MBRERR_DBREGNOTFOUND        -4
// Database ID not found in registry
#define MBRERR_ISPLAYING            -3
// Error function used but synthe still playing
#define MBRERR_CANCELLEDBYUSER      -2
#define MBRERR_NORESOURCE           -1
// Not enough resources to play
#define MBRERR_NOERROR             0

// Mbrola Windows Messages (used for notification)
#define WM_MBR_INIT                       (WM_USER+0x1BFF)
#define WM_MBR_READ                       (WM_USER+0x1C00)
#define WM_MBR_WAIT                       (WM_USER+0x1C01)
#define WM_MBR_WRITE                      (WM_USER+0x1C02)
#define WM_MBR_END                        (WM_USER+0x1C03)

// Callback Function type
typedef int (*LPPLAYCALLBACKPROC)(UINT msg,
      WPARAM wParam, LPARAM lParam);
typedef BOOL (*LPENUMDATABASECALLBACK)
      (LPCTSTR lpszDatabase, DWORD dwUserData);

extern "C"
{
LONG __declspec(dllimport) WINAPI MBR_Play(LPCTSTR lpszText,DWORD
      dwFlags,LPCTSTR lpszOutFile,DWORD dwCallback);
LONG __declspec(dllimport)WINAPI MBR_Stop();
LONG __declspec(dllimport) WINAPI MBR_WaitForEnd();
LONG __declspec(dllimport) WINAPI MBR_SetPitchRatio(float fPitch);
LONG __declspec(dllimport) WINAPI
      MBR_SetDurationRatio(float fDuration);
LONG __declspec(dllimport) WINAPI MBR_SetVoiceFreq(LONG lFreq);
float __declspec(dllimport) WINAPI MBR_GetPitchRatio();
float __declspec(dllimport) WINAPI MBR_GetDurationRatio();
LONG __declspec(dllimport) WINAPI MBR_GetVoiceFreq();
LONG __declspec(dllimport) WINAPI MBR_SetDatabase(LPCTSTR lpszID);
LONG __declspec(dllimport) WINAPI
      MBR_GetDatabase(LPTSTR lpID, DWORD dwSize);
BOOL __declspec(dllimport) WINAPI MBR_IsPlaying();
LONG __declspec(dllimport) WINAPI
      MBR_LastError(LPTSTR lpszError,DWORD dwSize);

// Syntheszier General informations
void __declspec(dllimport)
      WINAPI MBR_GetVersion(LPTSTR lpVersion, DWORD dwSize);

// Current Database Info
LONG __declspec(dllimport) WINAPI MBR_GetDefaultFreq();
LONG __declspec(dllimport) WINAPI
      MBR_GetDatabaseInfo(DWORD idx, LPTSTR lpMsg, DWORD dwSize);
LONG __declspec(dllimport) WINAPI
      MBR_GetDatabaseAllInfo(LPTSTR lpMsg, DWORD dwSize);

// Registry Related Functions
LONG __declspec(dllimport) WINAPI
      MBR_RegEnumDatabase(LPTSTR lpszData,DWORD dwSize);
LONG __declspec(dllimport) WINAPI MBR_RegEnumDatabaseCallback
      (LPENUMDATABASECALLBACK lpedCallback,DWORD dwUserData);
LONG __declspec(dllimport) WINAPI MBR_RegGetDatabaseLabel
      (LPCTSTR lpszID, LPTSTR lpLabel, DWORD dwSize);
LONG __declspec(dllimport) WINAPI MBR_RegGetDatabasePath
      (LPCTSTR lpszID, LPTSTR lpPath, DWORD dwSize);
LONG __declspec(dllimport) WINAPI MBR_RegGetDatabaseCount();
LONG __declspec(dllimport) WINAPI MBR_RegGetDefaultDatabase
      (LPTSTR lpID, DWORD dwSize);
LONG __declspec(dllimport) WINAPI
      MBR_RegSetDefaultDatabase(LPCTSTR lpszID);
BOOL __declspec(dllimport) WINAPI MBR_RegisterDatabase
      (LPCTSTR dbId,LPCTSTR dbPath,LPCTSTR dbLabel,
      BOOL isDef,LPTSTR lpBuffer,DWORD dwSize);
BOOL __declspec(dllimport) WINAPI
      MBR_UnregisterDatabase(LPCTSTR dbId);
BOOL __declspec(dllimport) WINAPI MBR_UnregisterAll();
BOOL __declspec(dllimport) WINAPI MBR_DatabaseExist(LPCTSTR lpszID);

// Registry Releated Functions, accessed by index
BOOL __declspec(dllimport) WINAPI MBR_RegIdxGetDatabaseId(LONG
      nIdx, LPTSTR lpszId, DWORD dwSize);
BOOL __declspec(dllimport) WINAPI MBR_RegIdxGetDatabasePath
      (LONG nIdx, LPTSTR lpszPath, DWORD dwSize);
BOOL __declspec(dllimport) WINAPI MBR_RegIdxGetDatabaseLabel
      (LONG nIdx, LPTSTR lpszLabel, DWORD dwSize);
LONG __declspec(dllimport)
      WINAPI MBR_RegIdxGetDatabaseIndex(LPCTSTR lpszID);
LONG __declspec(dllimport) WINAPI MBR_RegIdxGetDefaultDatabase();
}

#endif
```

TEXTALK.H

```
// textalk.h : main header file for the TEXTALK application
//

#if !defined(AFX_TEXTALK_H__CE3C5F85_62ED_11D2_A882_0020AF68E0A5
      __INCLUDED_)
#define AFX_TEXTALK_H__CE3C5F85_62ED_11D2_A882_0020AF68E0A5__INCLUDED_
```

```
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
	#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"		// main symbols

/////////////////////////////////////////////////////////////////////////////
// CTextalkApp:
// See textalk.cpp for the implementation of this class
//

class CTextalkApp : public CWinApp
{
public:
	CTextalkApp();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CTextalkApp)
	public:
	virtual BOOL InitInstance();
	//}}AFX_VIRTUAL

// Implementation

	//{{AFX_MSG(CTextalkApp)
	afx_msg void OnAppAbout();
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_in
```

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


//////////////////////////////////////////////////////////////////////
// CTextalkApp

BEGIN_MESSAGE_MAP(CTextalkApp, CWinApp)
	//{{AFX_MSG_MAP(CTextalkApp)
	ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
	//}}AFX_MSG_MAP
	// Standard file based document commands
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
	// Standard print setup command
	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////
// CTextalkApp construction

CTextalkApp::CTextalkApp()
{
	// Add construction code here,
	// Place all significant initialization in InitInstance
}

//////////////////////////////////////////////////////////////////////
// The one and only CTextalkApp object

CTextalkApp theApp;

//////////////////////////////////////////////////////////////////////
// CTextalkApp initialization

BOOL CTextalkApp::InitInstance()
{
	AfxEnableControlContainer();
	// Standard initialization
	// If you are not using these features and wish to reduce
	// the size
	//  of your final executable, you should remove from the
	// following
	//  the specific initialization routines you do not need.

#ifdef _AFXDLL
	Enable3dControls(); // Call this when using MFC in a shared DLL
#else
	Enable3dControlsStatic();
	// Call this when linking to MFC statically
#endif

	// Change the registry key under which our settings are stored.
	// You should modify this string to be something appropriate
	// such as the name of your company or organization.
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));

	LoadStdProfileSettings();
	//Load standard INI file options (including MRU)
	// Register the application's document templates.  Document
	// templates serve as the connection between documents, frame
	// windows and views.

	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
		IDR_MAINFRAME,
		RUNTIME_CLASS(CTextalkDoc),
		RUNTIME_CLASS(CMainFrame),       // main SDI frame window
		RUNTIME_CLASS(CTextalkView));
	AddDocTemplate(pDocTemplate);

	// Parse command line for standard shell commands, DDE,
	// file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;

	// The one and only window has been initialized, so show
	// and update it.
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

//////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	BOOL  m_prosody;
	float m_pitch;
	float m_duration;
	long  m_freq;
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);
	// DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
	DECLARE_EVENTSINK_MAP()
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
      extern char prosody_flag;
      extern float pitch_ratio;
      extern float duration_ratio;
      extern long voice_freq;
      //{{AFX_DATA_INIT(CAboutDlg)
      m_prosody = prosody_flag;
      m_pitch = pitch_ratio;
      m_duration = duration_ratio;
      m_freq = voice_freq;
      //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CAboutDlg)
      DDX_Check(pDX, IDC_CHECK1, m_prosody);
      DDX_Text(pDX, IDC_EDIT1, m_pitch);
      DDV_MinMaxFloat(pDX, m_pitch, 0.3f, 3.f);
      DDX_Text(pDX, IDC_EDIT2, m_duration);
      DDV_MinMaxFloat(pDX, m_duration, 0.1f, 10.f);
      DDX_Text(pDX, IDC_EDIT3, m_freq);
      DDV_MinMaxLong(pDX, m_freq, 4000, 64000);
      //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
      //{{AFX_MSG_MAP(CAboutDlg)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
// This runs the dialog, then gets the new values of the variables
// and sets the appropriate functions.
void CTextalkApp::OnAppAbout()
{
      CAboutDlg aboutDlg;
      extern char prosody_flag;
      extern float pitch_ratio;
      extern float duration_ratio;
      extern long voice_freq;

      aboutDlg.DoModal();

      prosody_flag = aboutDlg.m_prosody;
      pitch_ratio = aboutDlg.m_pitch;
      duration_ratio = aboutDlg.m_duration;
      voice_freq = aboutDlg.m_freq;
      MBR_SetDurationRatio(aboutDlg.m_duration);
      MBR_SetPitchRatio(aboutDlg.m_pitch);
      MBR_SetVoiceFreq(aboutDlg.m_freq) == 0;
}

/////////////////////////////////////////////////////////////////////
// CTextalkApp commands

BEGIN_EVENTSINK_MAP(CAboutDlg, CDialog)
    //{{AFX_EVENTSINK_MAP(CAboutDlg)
      //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

Appendix IV. TexTalk User Guide

Installing TexTalk and Mbrola

Before TexTalk can be run, Mbrola must be installed on the system, and the english diphone database must be registered. Mbrola 3.00 is provided with this thesis, although the latest version can be downloaded from [1]. Simply run the installation executable, and follow the instructions. Once Mbrola is installed, the EN1.ZIP file must be unzipped into the Mbrola directory, and the database registered through the Windows control panel Mbrola program.

TexTalk can be installed simply by unzipping the TEXTALK.ZIP file into the MYPROJECTS directory of the Visual C++ development studio, or to some other directory.

Starting TexTalk

TexTalk can be compiled and run in the Visual C++ environment. Alternatively, the file TEXTALK.EXE can be copied from the /TEXTALK/DEBUG directory into /TEXTALK, and run from there.

Using TexTalk

The TexTalk application has a menu bar, a control bar, and a large edit window. Text can either be typed into the window, loaded from a file, or pasted from the windows clipboard. The application operates in many ways like a simple text editor.

Once the text is in the window, the text to speech conversion is started by pressing the blue play button on the control bar. It can be stopped by pressing the red stop button, but will only stop when the play buffer is empty.

The options can be changed by selecting options - change from the menu bar. The options are:

- Prosody on/off checkbox
- Pitch ratio, ranging from 0.3 to 3.0
- Speed ratio, ranging from 0.1 to 10
- Voice frequency, ranging from 4000 to 64000 Hz

Appendix V. CUV2 Documentation

A DESCRIPTION OF A COMPUTER-USABLE DICTIONARY FILE BASED ON THE OXFORD ADVANCED LEARNER'S DICTIONARY OF CURRENT ENGLISH

Roger Mitton,
Department of Computer Science,
Birkbeck College,
University of London,
Malet Street,
London WC1E 7HX
June 1992 (supersedes the versions of March and Nov 1986)

In 1985-86 I produced a dictionary file called CUVOALD (Computer Usable Version of the Oxford Advanced Learner's Dictionary). This was a partial dictionary of English in computer-usable form - "partial" because each entry contained only some of the information from the original dictionary, and "computer-usable" (rather than merely "computer-readable") because it was in a form that made it easy for programs to access it. A second file, called CUV2, was produced at the same time. This was derived from CUVOALD and was the same except that it also contained all inflected forms explicitly, eg it contained "added", "adding" and "adds" as well as "add". I have now added some information to each entry and some more entries to CUV2, to produce a new version of CUV2. This document describes this new file.
These files were derived originally from the Oxford Advanced Learner's Dictionary of Current English [1], third edition, published by the Oxford University Press, 1974, the machine-readable version of which is available to researchers from the Oxford Text Archive. The task of deriving them from the machine-readable OALDCE was carried out as part of a research project, funded by the Leverhulme Trust, into spelling correction. The more recent additions have been carried out as part of my research as a lecturer in Computer Science at Birkbeck College.

THE FILE FORMAT

CUV2 contains 70646 entries. Each entry occupies one line. Samples are given at the end of this document. The longest spelling is 23 characters; the longest pronunciation is also 23; the longest syntactic-tag field is also (coincidentally) 23; the number of syllables is just one character ('1' to '9'), and the longest verb-pattern field is 58. The fields are padded with spaces to the lengths of the longest, ie 23, 23, 23, 1 and 58, making the record length 128. The spelling begins at position 1, the pronunciation at position 24, the syntactic-tag field at position 47, the number of syllables is character 70, and the verb-pattern field begins at position 71. The file is sorted in ASCII sequence; this means, of course, that the entries are not in the same order as in the OALDCE.

WHAT THE DICTIONARY CONTAINS

Each entry consists of a spelling, a pronunciation, one or more syntactic tags (parts-of-speech) with rarity flags, a syllable count, and a set of verb patterns for verbs.
The first file derived from the OALDCE (CUVOALD) contained all the headwords and subentries from the original dictionary - subentries are words like "abandonment" which comes under the headword "abandon" - except for a handful that contained funny characters (such as "Lsd" where the "L" was a pound sign). Subentries were not included if they consisted of two or three separate words that occurred individually elsewhere in the dictionary, such as "division bell" which comes under the headword "division", except when the combination formed a syntactic unit not immediately predictable from its constituents, eg "above board", which is listed as an adverb. To this list of about 35,000 entries, I added about 2,500 proper names - common forenames, British towns with a population of over 5,000, countries, nationalities, states, counties and major cities of the world. I would like to have added many more proper names, but I didn't have the time.
The second version of the file (CUV2) contained all these entries plus inflected forms making a total of about 68,000 entries. Since 1986 I have made a number of corrections, added the rarity flags and the syllable counts and inserted about 2,000 new entries. The new entries, nearly all of which were derived forms of words already in the dictionary, were selected from a list of several thousand words that occurred in the LOB Corpus[3] but were not in CUV2. I also made changes to existing entries where these were implied by the new entries; for example, when adding a plural form of a word whose existing tag was "uncountable", it was necessary to change the tag of the singular form. I also added about 300 reasonably common abbreviations (see note below).
A number of words (ie spellings) have more than one entry in the OALDCE, eg "water 1" (noun) and "water 2" (verb). In CUV2, each word has only one entry unless it has two different pronunciations, eg "abuse" (noun and verb). I have departed from this rule in the case of compound adjectives, such as "hard-working", which have a slightly different stress pattern depending on whether they are used attributively ("she's a hard-working girl") or predicatively ("she's very hard-working"). These are entered only once; they generally have the attributive stress pattern except when the predicative one seemed the more natural. (See also the note below on abbreviations.) I have also given only one entry to those words that have strong and weak forms of pronunciation, such as "am" (which can be pronounced &m, @m or m). Generally it is the strong form that is entered.
As regards the coverage of the dictionary, readers might be interested in a paper by Geoffrey Sampson [4] in which he analyses a set of words from a sample of the LOB corpus[3] that were not in CUV2. The recent additions should have gone some way to plugging the gaps that his study identified.

THE SPELLINGS
The spelling contains the characters "A" to "Z", "a" to "z", hyphen, apostrophe, space, umlaut or diaeresis (HEX 22), cedilla (3C), circumflex (5E), acute (5F), grave (60) and tilde (7E). These diacritic characters precede the letter that they mark, eg "se~nor". (There are also the characters "5" and "6" in "MI5" and "MI6".)

THE PRONUNCIATIONS

The pronunciation uses a set of characters very like the one adopted by the Alvey Speech Club for representing IPA in ASCII [2]. The system is as follows:

```
 i   as in  bead       N  as in  sing
 I           bid       T          thin
 e           bed       D          then
 & (ampsnd)  bad       S          shed
 A          bard       Z         beige
 0 (zero)    cod      tS          etch
 O (cap O)  cord      dZ          edge
 U          good
 u          food       p t k b d g
 V           bud       m n f v s z
 3 (three)  bird       r l w h j
 @  "a" in about
eI   as in   day      R-linking (the sounding
@U            go      of a /r/ at the end of a
aI           eye      word when it is
aU           cow      followed by a vowel)
oI           boy      is marked R
I@          beer      eg fAR for "far"
e@          bare      (compare "far away"
U@          tour      with "far beyond").
```

Primary stress: apostrophe eg @'baUt ("about")
Secondary stress : comma eg ,&ntI'septIk
Plus-sign as in "courtship" and "bookclub" 'kOt+Sip 'bUk+klVb
When the spelling contains a space and/or a hyphen, the pronunciation has one also, eg above board @,bVv 'bOd air-raid 'e@-reId

THE SYNTACTIC TAGS

Every entry in the dictionary has at least one syntactic tag (part-of-speech code). If an entry has more than one (eg "report" noun and verb), they are in ASCII order and separated by commas. A code consists of three characters, the first two being the syntactic tag and the third a frequency class. The first is one of the capital letters "G" to "Z" (inclusive), which have the following meanings:

G	Anomalous verb
H	Transitive verb
I	Intransitive verb
J	Both transitive and intransitive verb
K	Countable noun
L	Uncountable noun
M	Both countable and uncountable noun
N	Proper noun
O	Adjective
P	Adverb
Q	Pronoun
R	Definite article
S	Indefinite article
T	Preposition
U	Prefix
V	Conjunction
W	Interjection
X	Particle
Y	Abbreviation
Z	Not classified

Into the M class go nouns used frequently in both ways, such as "coffee" ("a pot of coffee", "two coffees please"), and also nouns that are predominantly one or the other; they may be mainly uncountable with an occasional countable use, such as "waste" and "understanding" ("the barren wastes", "reach an understanding"), or mainly countable with an occasional uncountable use, like "ceremony" and "line" ("too much ceremony", "stand in line").
The second character in the tag code is either in the group "0" (zero) to "9", "@", or "A" to "E", in which case it indicates how to form inflexions, or it is one of the characters "a" to "z", "+" or "-", in which case it gives some extra information about the word. (Abbreviations have the following extra code symbols, not used by other entries: ">", ")", "]", "}", ":", "=" and "~".)
The inflexion codes "0" to "5" are for verbs and have the following meanings:

```
0  stem+s, stem+ing, stem+ed (like "work")
1  stem+es, stem+ing, stem+ed (like "wish")
2  replace final "e" by es, ing or ed (like "love")
3  replace final "y" by ies, ying or ied (like "apply")
4  stem+s; double final letter +ing or +ed (like "abet")
5  all inflexions are given in full since at least one of them
is irregular
The inflexion codes "6" to "@" are for nouns:
6  add s to form the plural (like "cat")
7  add es (like "fox")
8  replace final y by ies (like "pony")
9  plural is the same as the singular (like "sheep")
(if there is another plural form, this is entered separately,
eg "herring" - "shoals of herring / we'll have the herrings for
tea")
@  no plural
The remaining inflexion codes "A" to "E" are for adjectives:
A  No -r or -st form
B  Comp is +r, Sup is +st (like "subtle")
C  +er, +est (like "light")
D  Change final y to ier, iest (like "heavy")
E  Comp or Sup irregular - given in full
The letters "a" to "z" give extra  information  about  the
word. The letters "a" to "h" follow verbs, with the following
meaning:
a  3rd person sing present tense
b  present participle (-ing form)
c  past tense
d  past participle
e  some other part of the verb
f to h follow anomalous verbs only:
f  contraction of pronoun with verb
g  contraction of verb with "not"
h  other contraction
The letters "i" to "o" follow nouns:
i  singular form (pl is irregular or non-existent)
j  plural form
k  plural in form but behaves like a singular, eg "economics"
(may be used as a plural also, eg "acoustics is a modern
science/ the acoustics of this hall are dreadful")
l to o follow proper nouns only:
l  forenames of people
m  countries, states, counties
```

```
n  towns and cities
o  other
The letters "p" to "t" follow adjectives:
p  only used predicatively
q  only used attributively
r  comparative
s  superlative
t  can be attached to a preceding word by a hyphen
The remaining small letters (and "+") are as follows:
u  adverb (not interrog or relative)
v  interrogative adverb
w  relative adverb
+  adverbial particle
x  pronoun (not interrog or relative)
y  interrogative pronoun
z  relative pronoun
If the first character of the tag code is "R" to "X" or "Z",
the second   character  is  always  "-",  ie  there  is  never
any  extra information about words in these classes.
The  following  characters  are  used   only   after   "Y"
(the abbreviation code):
>  singular noun  (see notes below)
)  plural noun
]  both sing and plur
}  uncountable noun
:  title
=  proper noun
~  other
```

Examples of tags are: K7, countable noun that forms its plural by adding es; H3, transitive verb that forms its inflexions like "apply"; Ic, past tense of an intransitive verb; Qz, relative pronoun; T-, preposition. The syntactic tags are presented in tabular form later in this document.

There is, intentionally, some redundancy in this coding system. With the exception of "-", any given character in the second position only occurs with a particular wordclass; a "6", for example, can only qualify a noun, an "r" can only qualify an adjective, and so on. This makes the programming a bit easier. There is, obviously, no mnemonic significance to the codes; it is not intended that people should have to read these codes directly.

THE RARITY FLAGS

The third character of the syntactic tag is either "*", "%" or "$". This is a marker of word-frequency. "*" means that the word occurs in the most frequent 500 words of the LOB Corpus[3], the Brown Corpus[5], the Thorndike-Lorge word count[6] and the American Heritage Word Frequency Book[7], ie it occurs in the most frequent 500 of all four lists.

The "$" code means that the word is, in my opinion, rare, with my opinions being combined to some extent with those of two friends of mine. I realise that this definition of rarity seems highly unscientific, but there is no appreciably better way of doing it. I could perhaps have taken the opinions of many more people, but this would have been a long job and I doubt if the resulting list would have been much different. The problem is that today's computer-readable corpora, while certainly large enough to provide data about common words, are nowhere near large enough to provide data about rare words. A word that fails to appear in a corpus of several million words is not necessarily rare; conversely, a word that appears several

times in one sample might still be rare in general use. My spelling corrector needed to know something about the frequency of words in its dictionary and, in the absence of hard data, it was better for it to have my estimates than none at all.
The third code "%" is by far the commonest in the dictionary and denotes words that are neither "*" nor "$".
The rarity codes are attached to tags rather than to words because a word can be common in one use but rare in another. "Go", for example, is very common as a verb, but less common as a noun. The OALDCE lists "aneroid" as adjective and noun. While I am reasonably familiar with this word in the phrase "aneroid barometer", I can't remember ever coming across it as a noun.

THE VERB PATTERNS

The final string of letters and numbers, separated by commas, is for verbs only, and shows the "verb patterns" - the sentence structures - in which the verbs can occur. If an entry has more than one verb pattern, they are entered in number order and then in letter order within numbers. This (fairly complicated) system is taken straight from the OALDCE, and is explained in the book's introduction.

THE SYLLABLE COUNTS

The number of syllables was computed for each word by separate algorithms applied to the spelling and the pronunciation. If they produced the same number, as they did in the great majority of cases, this was entered in the dictionary. The remaining three thousand or so I did by hand.
For the great majority of words, the number of syllables is obvious. There are a few, however, for which this is not the case. The problems generally concern the "@" phoneme.
The sounds "I@" ("pier"), "U@" ("tour") and "aI@" ("hire") seem sometimes to be one syllable and sometimes two. I find that my own feelings - and those of others I have spoken to - are influenced by the spelling of the word. Whereas I am happy to count "higher" as having two syllables, I am not so sure about "hire". Similarly with "sear" (one) and "seer" (two). The sounds that follow the "@" also seem to have an effect. While I might be persuaded that "fire" has two syllables, I would be not happy about "fire-alarm" having four. Similarly, if "acquire" has three, does "acquiring" have four?
The problem is that the "@" is such a small part of the sound that it hardly qualifies as a syllable. If, on the one hand, it signifies the presence of a morpheme, its status seems raised and I am happy to accept it as a syllable. If, on the other hand, it has no special status and, furthermore, the adjacent sounds cause it almost to disappear, then I can't bring myself to call it a syllable at all. If it is in-between, then I am simply not sure.
Being forced to make a decision, I have generally counted "fire/hire/wire/pier/tour" and the like as one syllable, but, on another day, I might easily have counted them as two.
There is another continuum of "@" sounds in the middle of words like "labelling". Some seem fairly clear, such as "enamelling"; others not so, like "gambling" and "peddling" (and are "gambolling" and "pedalling" any different?). I suspect my decisions on these have been somewhat arbitrary, depending on whether a pronunciation with more "@" or less "@" seemed more natural at the time.
One more group of problematic words are those ending "ion" pronounced sometimes "I@n" and sometimes "j@n". I can imagine a vicar intoning the word

"communion" in church so as to give it a full four syllables, but then ordering a case of communion wine over the phone and giving it only three. "Champion" in "Champion the Wonder Horse" had three but in "We are the champions" it has two. Some of these have only one regular pronunciation - "companion", for example, clearly has three syllables - but, for the others, I suspect my decisions depended on which pronunciation came to mind when I was considering them.

THE ABBREVIATION ENTRIES

Largely because of the paper by Geoffrey Sampson referred to above, I have included many more abbreviations in the 1992 version of the dictionary, but I have done so with some reluctance since they do not fit easily into the existing scheme.

There were about 50 abbreviations (examples include "eg", "ie", "OAP" and "TNT") in the previous version, because they were listed in the main body of the OALDCE. They were not given any distinctive tags in the 1986 version of CUV2. This was a nuisance since, for example, any algorithm attempting to match spelling and pronunciation would be puzzled by an entry such as "etc" pronounced It'set@r@. I have now added about 300 abbreviations that seemed to me to be reasonably common, and given all abbreviations their own tag.

Some abbreviations, such as "amp" and "rev", seem to behave pretty much like ordinary words and I have not marked them as abbreviations. The rest now have their own tag - "Y". (The Y tag in the previous version was used for adverbial particles; these are now tagged P+.)

Some abbreviations clearly have their own pronunciation, eg UNESCO, and others clearly don't, eg cwt (hundredweight). I have given them their own pronunciation when it seemed to me that the abbreviation was sometimes pronounced on its own. For example, I can imagine someone saying that some event takes place in dZ&n @n feb (Jan and Feb), but I can't imagine them saying it takes place in mAr @n &pr
(Mar and Apr), so "Jan" gets dZn whereas "Apr" gets 'eIprIl. But this is often pretty arbitrary.

It is not uncommon for two words to share the same abbreviation, eg "Dr" for "Doctor" and "Drive" or "St" for "Saint" and "Street". It would have been a possibility to put in two (or sometimes more) entries for such items, along the lines of "convert" (noun and verb), but I did not feel that 'd0kt@R (or draIv) was the pronunciation of "Dr" in the way that 'k0nv3t (or k@n'v3t) was the pronunciation of "convert", so I was unwilling to give such abbreviations two or more entries, but at the same time I wanted to put something in the pronunciation field, so I just put one of the pronunciations in.

There is also an unsatisfactorily arbitrary quality to some of the tags. Abbreviations that can go after an article or possessive ("my PhD", "an FRS", "the MCC") were tagged singular noun ("Y>"), and a few can be plural ("GCSEs") ("Y)"). Some, mostly units of measurement ("cc", "rpm"), can be both ("Y]"). Uncountable noun abbreviations ("LSD", "TB") get "Y}". Titles ("Mr", "Col") get "Y:" while proper names ("Mon", "Aug") or abbreviations likely to form part of a proper name ("Ave", "Rd") get "Y=". Others ("asap", "viz") get "Y~". Oddly, some organization names seem to be proper names ("RADA", "UNESCO") while others don't ("the BBC", "the UN").

In short, then, I am uneasy about many of the decisions I have had to make in order to get these abbreviation entries into the same form as the rest of the dictionary, but the important thing is that they are now in the dictionary, so a piece of software using the dictionary will recognize them, and they are distinctively tagged for anyone who wants to take them out.

ACKNOWLEDGEMENTS
Most of the work of extracting the required information from the machine-readable OALDCE, putting it into a standard form in CUVOALD and generating CUV2 was carried out by me between January 1985 and March 1986. I added the rarity flags in 1988 and the syllable counts in 1990 and inserted the new entries in March-June 1992. Susan Drew keyed in the pronunciations of some of the entries, and the file was proofread by Philip Baker, Sylvia Davidson, Ann Jones, Ed Hastings, Kate Murray and Diana Whitaker. Deepa Dougal, as part of her Msc project, carried out the task of looking up all the words from the LOB Corpus in CUV2.

COPYRIGHT
I am making the file available to others, via the Oxford Text Archive, so that researchers who need a reasonably large computer-usable dictionary do not need to spend months, as I did, putting one together. Anyone contemplating commercial use of the file should contact the Oxford University Press.

REFERENCES
[1] Hornby A.S., Oxford Advanced Learner's Dictionary of Current English, Third Edition, Oxford University Press, 1974
[2] Wells J.W., "A standardised machine-readable phonetic notation", IEE conference "Speech input/output: techniques and applications" London, Easter 1986
[3] Hofland K, and S. Johansson, Word Frequencies in British and American English, Norwegian Computing Centre for the Humanities/ Longman, 1982
[4] Sampson G., "How fully does a machine-usable dictionary cover English text?" Literary and Linguistic Computing, Vol 4, No 1, 1989, pp 29-35
[5] Kucera H. and W.N. Francis, Computational Analysis of Present-day American English, Brown University Press, 1967
[6] Thorndike E.L. and I. Lorge, The Teacher's Word Book of 30,000 Words, Teachers College, Columbia University, 1944
[7] Carroll J.B., P. Davies and B. Richman, Word Frequency Book, American Heritage, 1971

Appendix VI. Mbrola Documentation

This is the documentation for the Mbrola input format, and the MBRPLAY DLL though which the programmer can use the Mbrola speech synthesiser.

MBROLA INPUT FORMAT

Phoneme commands

The input file bonjour.pho in the above example simply contains :

```
; bonjour
_ 51 25 114
b 62
o~ 127 48 170.42
Z 110 53.5 116
u 211
R 150 50 91
_ 91
```

This shows the format of the input data required by MBROLA. Each line contains a phoneme name, a duration (in ms), and a series (possibly none) of pitch targets composed of two float numbers each : the position of the pitch target within the phoneme (in % of its total duration), and the pitch value (in Hz) at this position.

In order to increase readability, it is also possible to enclose pitch target in parentheses. Hence, the first line of bonjour.pho could be written :

```
_ 51 (25,114)
```

it tells the synthesizer to produce a silence of 51 ms, and to put a pitch target of 114 Hz at 25% of 51 ms. Pitch targets define a piecewise linear pitch curve. Notice that the pitch targets they define is continuous, since the program automatically drops pitch information when synthesizing unvoiced phones.

The data on each line are separated by blank characters or tabs. Comments can optionally be introduced in command files, starting with a semi-colon ';'. This default can be overrun with the -c option of the command line.

Another special escape sequence ';;' allow the user to introduce commands in the middle of .pho files as described below. This escape sequence is also affected by the -c option.

Changing the Freq Ratio or Time Ratio

A command escape sequence containing a line like "T=xx" modifies the time ratio to xx, the same result is obtained on the fundamental frequency by replacing T with F, like in:

```
;; T = 1.2
```

;;F=0.8

Renaming phonemes in a set
Command escape sequences may also define renaming tables of for the phoneme set. A line like:

```
;; RENAME A my_a
```

tells the synthesizer that the phoneme previously called A is now called my_a. This facility is provided to make your life easier when your Natural Language Processing unit does not complies to our SAMPA alphabet. The only limitation is that the phoneme name can't contain blank characters.

We suggest that you don't mix renaming commands and true .pho files, for example grouping all your rename command in a '.set' file, and then calling:
mbrola fr1/fr1 fr1.set command1.pho command2.pho output.wav
WARNING: circular renaming can lead to name collision, like in
;; RENAME y u
;; RENAME u ou

THIS GENERATES AN ERROR BECAUSE OF NAME COLLISIONS (old y and u will
be named as ou)

which should be written:
;; RENAME u ou
;; RENAME y u

When circuits in renaming can't be avoided, like in:
;; RENAME # _
;; RENAME _ #

you should write:
;; RENAME # temp
;; RENAME _ #
;; RENAME temp _

Once the renaming has occurred there is absolutely NO PERFORMANCE DROPS related to this renaming, so use it rather than a pre-processor.

Before renaming anything as # check the paragraph below!

5.4 Flush the output stream
Note, finally, that the synthesizer outputs chunks of synthetic speech determined as sections of the piecewise linear pitch curve. Phones inside a section of this curve are synthesized in one go. The last one of each chunk, however, cannot be properly synthesized while the next phone is not known (since the program uses diphones as base speech units). When using mbrola with pipes, this may be a problem.

Imagine, for instance, that mbrola is used to create a pipe-based speaking clock on an HP:

```
speaking_clock | mbrola - -.au | splayer
```

which tells the time, say, every 30 seconds. The last phone of each time announcement will only be synthesized when the next announcement starts. To bypass this problem, mbrola accepts a special command phone, which flushes the synthesis buffer : "#"

This default character can be replaced by another symbol thanks to the command:

```
;; FLUSH new_flush_symbol
```

Limitations of the program
1. There may be up to 20 pitch targets in each phone, although not more than three or four are sufficient to copy natural prosody. We have set up a higher limit so as to enable the use of MBROLA to produce synthetic singing voices, in which case long vowels with vibrato may require a large number of pitch targets.

2. Phones can be synthesized with a maximum duration which depends on the fundamental frequency with which they are produced. The higher the frequency, the lower the duration. For a frequency of 133 Hz, the maximum duration is 7.5 sec. For a frequency of 66.5 Hz, it is 15 sec. For a frequency of 266 Hz, it is 3.75 sec.

3. Although pitch targets are facultative, the synthesizer will refuse to produce sequences of more than 250 phones with no pitch information.

MBRPLAY DLL FUNCTIONS

MBR_Play

Play pho or phs files to an output device (sound board or file).
Play can play memory or disk files.

LONG MBR_Play(LPCTSTR lpszText,DWORD dwFlags,LPCTSTR lpszOutFile,DWORD dwCallback);

Parameters :

LPCTSTR lpszText : - string containing the pho or phs memory file to play or a filename if the MBR_BYFILE is specified in dwFlags
DWORD dwFlags : flags setting properties of the play. The values are :

MBR_BYFILE lpszText is a .pho file name and not a phoneme string
MBR_WAIT play in synchronous mode
MBR_QUEUED the output device is not close when the play ends and the system waits for another play to continue writting to the device

MBR_ASPHS the file specified in lpszText is a phs file
MBR_CALLBACK the dwCallback is the address of a callback function instead of a window handle
MBR_MSGINIT send the init notification message
MBR_MSGREAD send the read notification message
MBR_MSGWAIT send the wait notification message
MBR_MSGWRITE send the write notification message
MBR_MSGEND send the end notification message
MBR_MSGALL send all the notification messages
MBROUT_SOUNDBOARD send the result of synthesizer to the soundboard
MBROUT_RAW send the result of synthesizer to an output file, raw datas format
MBROUT_WAVE send the result of synthesizer to an output file, windows wave format
MBROUT_AU send the result of synthesizer to an output file, sun audio format
MBROUT_AIFF send the result of synthesizer to an output file, macintosh audio format
MBROUT_ALAW the output is filtered with an A-law 8000Hz coding (only for raw & wave files)
MBROUT_MULAW the output is filtered with an mu-law 8000Hz coding (only for raw & wave files) The parameter can be NULL;
LPCTSTR lpszOutFile : Specify the name of an output file that receive the output datas (when MBROUT_RAW,WAVE,AU or AIFF is chosen) This parameter can be NULL;
DWORD dwCallback : Specify a Window Handle that will receive notification messages on the current state of the player.
if the MBR_CALLBACK flags is set, the dwCallback is the address of a callback function receiving the messages.
This parameter can be NULL.

Remarks :

see the MBR_MSG* messages for more explanation
see the the PlayCallbackProc for more explanation

Return Value :

0 if no error occured
an error code (see below)

MBR_Stop

Stop the current play task.

LONG MBR_Stop();

Return Value :

0 if no error occured,
or an error code (see below)

MBR_WaitForEnd

Wait until the end of the current play task.

LONG MBR_WaitForEnd();

Return Value :
0 if no error occured,
or an error code (see below)

MBR_SetPitchRatio

Set the pitch ratio.

LONG MBR_SetPitchRatio(float fPitch)

Parameters :

fPitch	: the new pitch ratio

Return Value :
0 if no error occured,
or an error code (see below)

MBR_SetDurationRatio

Set the duration ratio.

LONG MBR_SetDurationRatio(float fDuration);

Parameters :

fDuration	: the new duration ratio

Return Value :
0 if no error occured,
or an error code (see below)

MBR_SetVoiceFreq

Set the voice frequency.

LONG MBR_SetVoiceFreq(LONG lFreq);

Parameters :

lFreq	: the new frequency

Return Value :
0 if no error occured,
or an error code (see below)

MBR_GetPitchRatio

Get the pitch ratio.

float MBR_GetPitchRatio();

Return Value :
the current pitch ratio.

MBR_GetDurationRatio

Get the duration ratio.

float MBR_GetDurationRatio();

Return Value :
the current duration ratio.

MBR_GetVoiceFreq

Get the voice frequency.

float MBR_GetVoiceFreq();

Return Value :
the current voice frequency.

MBR_SetDatabase

Load a diphone database.

LONG MBR_SetDatabase(LPCTSTR lpszID);

Parameters :

lpszID	: the ID of the database (a registered ID), or an absolute path. If the ID is not found the register, MBR_SetDatabase try to load lpszID as an absolute path.

Return Value :
0 if no error occured,
or an error code (see below)

MBR_GetDatabase

Get the ID of the current database loaded.

LONG MBR_GetDatabase(LPTSTR lpID, DWORD dwSize);

Parameters :

lpID	: pointer to a buffer that will receive the database Id.
dwSize	: size of the buffer.

Return Value :
0 if no error occured,
or an error code (see below)

MBR_IsPlaying

Does the dll play ?

BOOL MBR_IsPlaying();

Return Values :

TRUE if the dll is playing, FALSE otherwise.

MBR_LastError

Get the text and the code of the last error occured.

LONG MBR_LastError(LPTSTR lpszError,DWORD dwSize);

Parameters :

lpszError	: a pointer to a buffer receiving the text of the last error occured.
dwSize	: size of the buffer.

Return Value :

The code of the last error occured.

MBR_GetVersion

Get the version of the Mbrola Synthesizer used.

void MBR_GetVersion(LPTSTR lpVersion, DWORD dwSize);

Parameters :

lpVersion	: a pointer to the buffer receiving the version of the synthesizer
dwSize	: size of the buffer.

MBR_GetDefaultFreq

Get the default frequency of the current loaded database.

LONG MBR_GetDefaultFreq();

Return Value :

The default frequency.

MBR_GetDatabaseInfo

Get A paragraph of information of the comments contained in the current loaded database.

LONG MBR_GetDatabaseInfo(DWORD idx, LPTSTR lpMsg, DWORD dwSize);

Parameters :

idx	: index of the paragraph
lpMsg	: buffer that will receive the paragraph
dwSize	: size of the buffer

Remarks :
if lpMsg is set to NULL, the MBR_GetDatabaseInfo will return the length of the paragraph.

Return Value :
the length of the paragraph if lpMsg is NULL,
or the number of characters copied if lpMSg is not NULL.
if idx is not a valid paragraph number, the MBR_GetDatabaseInfo will return 0

MBR_GetDatabaseAllInfo
Get all the paragraph of comments contained in the current loaded database.

LONG MBR_GetDatabaseAllInfo(LPTSTR lpMsg, DWORD dwSize);

Parameters :

lpMsg	: pointer to a buffer that will receive the comments.
dwSize	: size of the buffer

Remarks :
if lpMsg is set to NULL, the MBR_GetDatabaseAllInfo will return the length
of the comments text.

Return Value :
the length of the comments text if lpMsg is NULL,
or the number of characters copied if lpMSg is not NULL.
0 if there is no comments for the current database.

MBR_RegEnumDatabase
Enum the database ID of all the databases registered in Windows's Registry.

LONG MBR_RegEnumDatabase(LPTSTR lpszData,DWORD dwSize);

Parameters :

lpszData	: pointer to a buffer that will receive all the database ID. The database ID's are separated by a null character. A second null character is placed after the last database.
dwSize	: size of the buffer.

Return Value :
0 if no error occured,
or an error code (see below)

MBR_RegEnumDatabaseCallback
Enum the database ID of all the the databases registered in Window's Registry, using a
callback function.

LONG MBR_RegEnumDatabaseCallback(LPENUMDATABASECALLBACK lpedCallback,DWORD dwUserData);

Parameters :

lpedCallback : the address of the callback function (see below the LPENUMDATABASECALLBACK)

dwUserData : a pointer to users datas passed to the callback function at each call.

Return Value :

0 if no error occured,
or an error code (see below)

MBR_RegGetDatabaseLabel

Get the label of a specified database (passing a database ID)

LONG MBR_RegGetDatabaseLabel(LPCTSTR lpszID, LPTSTR lpLabel, DWORD dwSize);

Parameters :

lpszID	: a null terminated string containing the database ID
lpLabel	: a pointer to the buffer receiving the label
dwSize	: the size of the buffer.

Return Value :

0 if no error occured,
or an error code (see below)

MBR_RegGetDatabasePath

Get the path of a specified database (passing a database ID)

LONG MBR_RegGetDatabasePath(LPCTSTR lpszID, LPTSTR lpPath, DWORD dwSize);

Parameters :

lpszID	: a null terminated string containing the database ID
lpPath	: a pointer to the buffer receiving the path
dwSize	: the size of the buffer.

Return Value :

0 if no error occured,
or an error code (see below)

MBR_RegGetDatabaseCount

Get the number of registered databases.

LONG MBR_RegGetDatabaseCount();

Return Value :
    the number of registered databases.

MBR_RegGetDefaultDatabase

Get the ID of the default database registered in the Windows Registry

LONG MBR_RegGetDefaultDatabase(LPTSTR lpID, DWORD dwSize);

Parameters :

lpId	: a pointer to the buffer receiving the id
dwSize	: the size of the buffer.

Return Value :
    0 if no error occured,
    or an error code (see below)

MBR_RegSetDefaultDatabase

Set the new default registered database (passing the ID)

LONG MBR_RegSetDefaultDatabase(LPCTSTR lpszID);

Parameters :

lpszID	: a null terminated string containing the ID of the new default database.
dwSize	: the size of the buffer.

Return Value :
    0 if no error occured,
    or an error code (see below)

MBR_RegisterDatabase

Register a new database in the Windows registry

BOOL MBR_RegisterDatabase(LPCTSTR dbId,LPCTSTR dbPath,LPCTSTR dbLabel,BOOL isDef,LPTSTR lpBuffer,DWORD dwSize);

Parameters :

dbId	: a null terminated string containing the ID of the new database.
dbPath	: a null terminated string containing the path of the new database.
dbLabel	: a null terminated string containing the label of the new database.
isDef	: if TRUE, the new database becomes the default database.
lpBuffer	: a pointer to a buffer receiving the real ID of the new database. If dbId is an existing ID, a new name is created and is sent back in lpBuffer.
dwSize	: the size of the buffer.

Return Value :

TRUE if no error occured,
FALSE otherwise

MBR_UnregisterDatabase

Unregister the database from the Windows Registry (passing an ID)

BOOL MBR_UnregisterDatabase(LPCTSTR dbId);

Parameters :

dbID : a null terminated string containing the ID of the database to be removed.

Return Value :

TRUE if no error occured,
FALSE otherwise.

MBR_UnregisterAll

Unregister all the databases

BOOL MBR_UnregisterAll();

Return Value :

TRUE if no error occured,
FALSE otherwise.

MBR_DatabaseExist

Test if a database is registered.

BOOL MBR_DatabaseExist(LPCTSTR lpszID);

Parameters :

dbID : a null terminated string containing the ID of the database.

Return Value :

TRUE if the database is registered,
FALSE otherwise.

MBR_RegIdxGetDatabaseId

Get the database ID of a registered database, passing an index.

BOOL MBR_RegIdxGetDatabaseId(LONG nIdx, LPTSTR lpszId, DWORD dwSize);

Parameters :

nIdx	: index of the database to retrieve the ID
lpszId	: a pointer to a buffer that will receive the ID
dwSize	: size of the buffer.

Return Value :
TRUE if no error occured,
FALSE otherwise.

MBR_RegIdxGetDatabasePath

Get the path of a registered database, passing an index.

BOOL MBR_RegIdxGetDatabasePath(LONG nIdx, LPTSTR lpszPath, DWORD dwSize);

Parameters :

nIdx	: index of the database to retrieve the ID
lpszPath	: a pointer to a buffer that will receive the path
dwSize	: size of the buffer.

Return Value :
TRUE if no error occured,
FALSE otherwise.

MBR_RegIdxGetDatabaseLabel

Get the label of a registered database, passing an index.

BOOL MBR_RegIdxGetDatabaseLabel(LONG nIdx, LPTSTR lpszLabel, DWORD dwSize);

Parameters :

nIdx	: index of the database to retrieve the ID
lpszLabel	: a pointer to a buffer that will receive the label
dwSize	: size of the buffer.

Return Value :
TRUE if no error occured,
FALSE otherwise.

MBR_RegIdxGetDatabaseIndex

Get the index of a registered database (passing its ID).

LONG MBR_RegIdxGetDatabaseIndex(LPCTSTR lpszID);

Parameters :

lpszID	: a null terminated string containing the ID of the database

Return Value :
the index of the database.
-1 if the database is not registered.

MBR_RegIdxGetDefaultDatabase

Get the index of the default registered database.

```
LONG MBR_RegIdxGetDefaultDatabase();
```

Return Value :

the index of the database.

MbrPlay Error Codes

```
MBRERR_NOREGISTRY       -13    // Registry keys error
MBRERR_NOMBROLADLL      -12    // Mbrola DLL not found
MBRERR_DBINIT           -11    // No database loaded
MBRERR_WARNING          -10    // A Warning occured
MBRERR_CANTOPENWAVEOUT  -9     // Can't open the wavout device
MBRERR_CANTOPENFILEOUT  -8     // Can't open the file for output
MBRERR_CANTOPENFILE     -7     // Can't open a file
MBRERR_ERRORSPEAKING    -6     // Error while Speaking
MBRERR_DBNOTDATABASE    -5     // Not a valid database
MBRERR_DBREGNOTFOUND    -4     // Database ID not found in registry
MBRERR_ISPLAYING        -3     // Error function used but synthe still playing
MBRERR_CANCELLEDBYUSER  -2     // Process cancelled by the programmer
MBRERR_NORESOURCE       -1     // Not enough resources to play
MBRERR_NOERROR           0     // No error
```

MbrPlay Windows Custom messages, used for notification

```
WM_MBR_INIT       (WM_USER+0x1BFF)
WM_MBR_READ       (WM_USER+0x1C00)
WM_MBR_WAIT       (WM_USER+0x1C01)
WM_MBR_WRITE      (WM_USER+0x1C02)
WM_MBR_END        (WM_USER+0x1C03)
```

Callback Functions :

```
typedef int (*LPPLAYCALLBACKPROC)(UINT msg, WPARAM wParam,
LPARAM lParam);
typedef BOOL (*LPENUMDATABASECALLBACK)(LPCTSTR lpszDatabase,
DWORD dwUserData);
```

Appendix VII. Survey Results

There are 5 test sentences involved in the survey. The sentences are:

1. Does it read these sentences correctly?
2. I will read this document, but first, have you read my latest book?
3. According to scientists, it is possible that dinosaurs lived in California 256157012 years ago.
4. Let's go to the library complex and discuss complex issues.
5. I thought he owned both the red car and the black van.

The three text to speech systems used in the survey are TexTalk, WinSpeech and Monologue 97. Ten people were surveyed, and the average of their scores are shown in the table below. A score of one is for the worst quality of speech, and a score of ten is for the best.

Sentence	TexTalk	WinSpeech	Monologue 97
1	6.5	3.1	6.5
2	5.5	4.3	5.9
3	7.4	3.6	4.6
4	5.0	3.8	6.2
5	5.4	3.5	5.3
Total	29.8	18.3	28.5

Because of the small number of people surveyed these results are far from accurate, but they do give some indication of relative speech qualities. When asked to comment on these results, the consensus was that:

- WinSpeech was very difficult to understand, and sounded very robotic;
- While Monologue was slightly clearer to understand than TexTalk,
- TexTalk sounded the most natural.
- Monologue can pick out the different uses of ambiguous words.

References

[1] Microsoft Research’s Speech Technology Group web page: http://www.research.microsoft.com/research/srg/, [10/5/98]

[2] Schindler,

Bibliography

These are some sources of additional information on the topic.

Abercrombie, D., *Studies in Phonetics and Linguistics*, London: Oxford University Press, 1965.

Bristow, G., *Electronic Speech Synthesis: Techniques, Technology, and Applications*, London: Granada Publishing Ltd, 1984.

Brown, G., Currie, K.L., and Kenworthy, J., *Questions of Intonation*, London: Croom Helm, 1980.

Dutoit, T., "*An Introduction to Text-To-Speech Synthesis*", Dordrecht: Kluwer Academic Publishers, 1997.

Dutoit, T., Pagel, V., Pierret, N., Bataille, F., Van Der Vrecken, O., "*The MBROLA Project: Towards a Set of High-Quality Speech Synthesizers Free of Use for Non-Commercial Purposes*", Proc. ICSLP'96, Philadelphia, vol. 3, pp. 1393-1396.

Flanagan, J.L. and Rabiner, L.R., *Speech Synthesis*, Stroudsburg: Dowden, Hutchinson & Ross, Inc., 1973.

Holmes, J.N., *Speech Synthesis and Recognition*, London: Chapman and Hall, 1988.

Teja, E.R., and Gonnella, G., *Voice Technology,* Reston: Reston Publishing Company, 1983.

Witten, I.H., *Principles of Computer Speech*, London: Academic Press, 1982.