THE DESIGN AND EVALUATION OF A SPEECH RECOGNITION SYSTEM FOR ENGINEERING WORKSTATIONS

by

Robert A. Kavaler

Memorandum No. UCB/ERL M86/39

5 May 1986

THE DESIGN AND EVALUATION OF A SPEECH RECOGNITION SYSTEM FOR ENGINEERING WORKSTATIONS

by

Robert A. Kavaler

Memorandum No. UCB/ERL M86/39

5 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Acknowledgements

I would like to thank everyone who helped with the project from its beginning many years ago: Hy Murveit who developed the original recognition algorithm and hardware, Bill Baringer who built the prototype SPUDS board, Tobias Noll who helped design and lay out the chip, and Meni Lowy who designed the first generation dynamic-time-warp chip. I would also like to thank my test subjects Peter Ruetz and Steve Lewis. I would also like to thank GTE for their financial support during the final year of my graduate work.

Finally, I would like to thank my advisor, Bob Brodersen, for his support and guidance over the past four years.

The Design and Evaluation of a Speech Recognition System for Engineering Workstations

Ph.D. Robert A. Kavaler E.E.C.S.

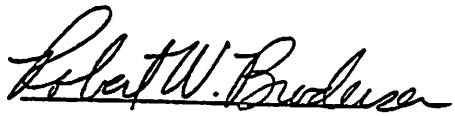
Abstract

This thesis describes a complete speech recognition system for engineering workstations. Four areas are discussed in detail: the basic recognition algorithm, the hardware design, the user interface, and the application interface. Each system component is evaluated separately, and the system as a whole is evaluated for one application well suited to speech input.

The speech recognizer is integrated into an existing engineering workstation as another input device, along with the existing keyboard and mouse. Five copies of the recognizer board were built. Although the basic recognition algorithm is a speaker dependent dynamic-time-warp algorithm, the resulting system is easy to train and has high performance. Up to one thousand words can be recognized in real time. An adaptive training algorithm was developed using user feedback to increase recognition accuracy from 97.5% to 99.5%.

The thesis arrives at four main conclusions. First, using special purpose integrated circuits tailored to a specific algorithm is a practical and efficient way to produce large vocabulary speech recognition systems. Second, recognition feedback to the user will increase user acceptance of the system and increase recognition accuracy. Third, the interface provided by most computer systems and applications is not immediately usable by speech input. Speech is inherently word oriented while most applications depend on keyboards which are character oriented. Some mechanism other than the simple one word corresponds to one character string must convert speech commands into computer commands. Finally, the speech recognition system can best be evaluated by examining the user interface as a whole, not evaluating the speech recognizer alone. The recognizer was evaluated with a typical application and it was found that while the speed of the interaction did not increase significantly, users experienced less

strain and could work longer with the recognizer than without it.



Committee Chairman

Table of Contents

Introduction

In recent years much has been said about the coming boom of speech-oriented interfaces to computers. In fact, most papers summarizing speech recognition start with a statement such as "Speech has long been thought of as the ultimate man-machine interface..." The purpose of this project is to develop a system based on current speech recognition techniques to verify that premise. Many people think it would be preferable to "talk" to their computer instead of typing, however, but to date no such systems really exist. Current attempts at using speech recognition have been in either highly automated environments such as assembly lines, mass data entry, or in "hands-busy" environments where the use of a keyboard severely restricts the speed and accuracy of data entry. People are not really "talking" to their computer, they are entering text and data using their voice.

The Mara system is an attempt to integrate a template-based speech recognizer into a single-user engineering/programming workstation. In this application there are three components of importance: the computer and its programs, the speech recognizer and, most importantly, the computer's user. The Mara system addresses the needs of ALL THREE components in an attempt to build a very high accuracy, easy to use, computer system, based on speech input.

INTRO.1. The Application

With the current increase in computer power available with 16 and 32 bit microprocessors, engineering/programming workstations are quickly becoming the norm. These workstations generally consist of a 1 MIPS (million instructions per second) CPU, 2-4 Megabytes of memory, a high resolution (1000 by 1000 pixels) display, a keyboard, and a mouse. The workstations do not however run the same software. Some workstations run LISP,[1] some UNIX,[2] and others proprietary operating systems. Due to the general computing environment at Berkeley, we chose UNIX workstations made by SUN Microsystems (model 120). These workstations are used for in many different applications:

- Writing programs (editing/compiling/debugging).
- Design, layout, simulation of integrated circuits.

- Writing papers.
- Daily routines such as electronic mail, messages, record keeping, etc.

The UNIX environment is based on keyboard input devices and CRT output devices. Thus, a retrofit speech recognizer will have to create simulated keyboard input; as a speech synthesizer would have to read the CRT output. Since the main input device is a keyboard, the standard user interface exploits this by:

- minimizing the number of keystrokes for commands (common program names are usually a few characters long: "od", "wc", "vi", "as", "cc", etc.),
- using many special symbols that have special meanings, and
- allowing aliasing of complex commands to a few keystrokes.

The UNIX user interface is "character-oriented", not "word-oriented". For speech recognizers this has tremendous consequences since speech is inherently word-oriented. The obvious keyboard retrofit interface is to simulate a programmable string which is typed when a word is spoken. This technique is popular, but has many flaws:

- Different programs require different strings (e.g. the word "quit" must send the string ":q" in one program, "q" in another, and "quit" in another).
- The syntax implied by a keyboard, that of concatenating characters, seems awkward when speech is used.
- Words are implicitly separated, but character strings require explicit separation, and the separation characters are not always the same.
- Most keyboard commands are terminated with a "return" character, while speech commands may or may not be terminated with long silences. Requiring users to say "now" or some similar word at the end of a command is cumbersome.

The design of the speech recognition system must bridge the gap between word and character oriented interfaces.

The general purpose UNIX workstation application was chosen for a few reasons:

- The application should contain most of the aspects of normal computer usage.
- To date, attempts at designing such interfaces have not been proven to be useful.
- The flexibility needed for this application could be applied to less complex interfaces such as data entry and hands-busy environments.

INTRO.2. The Speech Recognizer

Adding a speech recognizer to an existing system has many effects on the computer and user: new hardware is needed, the user must wear a microphone, some of the screen may be used by the recognizer, and the user will have to learn how to use the new system. Also, most recognizers must be trained by the user.

The speech recognizer used in this project is a speaker-dependent template-based system. The system is typical of the technology currently available in commercial recognizers. Template-based systems require that each word to be recognized (called the vocabulary) must be spoken at least once by the user. A template is then generated for each vocabulary word in a process called training. Training is a very important part of the overall system because:

- High recognition accuracy depends on good training.
- Existing input devices to computers (keyboard/mouse) do not require training but speech recognizers do require training.
- During training, the user is not performing "useful work".

In fact, training is considered the biggest drawback to speaker-dependent template-based systems and the reason why many people think these recognizers are not desirable. Keyboard input devices themselves require no training, instead users must be trained to use them. Speech recognizers have the opposite problem: since everyone knows how to speak they want the recognizer to train to them.

The recognizer needs speech input from a microphone placed near the speaker. Many different types and styles of microphones can be used. Close-talking head-mounted microphones provide for some noise rejection at the cost of "wearing" the microphone. Other microphones such as lapel microphones are also worn, but are not as noticeable. Unfortunately, lapel microphones must be mounted on suitable

clothing. Stationary microphones are difficult to use because users tend to move while they work, and thus the speech signal at the microphone fades in and out. Also, stationary microphones tend to pick up background noise because they are not close to the speaker. The selection of the proper microphone is dependent on the application, the user, and the cost of the microphone. Our system was designed to use a close-talking head-mounted microphone (Shure 10A), but other microphones have been used successfully.

INTRO.3. Input Devices

In order to design the most efficient interface possible, each input device should do only what it does best.

- Mouse - use for positioning and selection of existing items. Also, a mouse button can supply a menu of commands, but this is only needed as a "help" feature.
- Keyboard - use for mass text entry and adding new words to the recognizer. The speech recognizer is not too useful as a "dictation machine" due to its limited vocabulary.
- Speech Recognizer - use for everything else. This includes all commands, file names, and limited vocabulary text entry (such as programs with a few hundres variable names).

INTRO.4. System Design Goals

The application forced many of the design goals for the system. First the system must have "real-time" response. Delays between the spoken end of a word and that word's corresponding action (called the latency) must be very small (about 0.5 sec is considered by many to be the upper limit). Second, the system must have high accuracy for large vocabularies. The projected vocabulary sizes were 300 words, but because multiple applications could be running at one time this number could easily reach 500 to 1000 words. Accuracy should be high enough that the user not be significantly slowed down due to recognition errors.

After a few iterations of designing suitable user interfaces we found that the major bottleneck to using the recognizer was the difficulty in adding new words to the vocabulary. Thus, an important design

goal of the system was to be able to add new applications, extend current applications, add new words, and change (adapt) words to new pronunciations without running a special program and with a minimum of user intervention.

In order to implement these goals we decided early in the project that special hardware needed to be designed because commercially available speech recognizers could not be used.

- Commercial speech recognition products are proprietary. Therefore an important part of the system, the recognizer, would have to be treated as a "black box". This would be too limiting for our research.
- No existing systems can handle the required vocabulary size.
- Most systems either connect directly to an IBM PC through its internal bus or talk on low-speed (9600 baud or less) terminal lines. Thus, we would have to redesign the interface for our workstations.

We decided to design our own Multibus compatible speech recognition hardware, plugging it into our existing workstations. Because the hardware was actually going to be built it had to be debuggable, work reliably, and be reproducible.

The primary goal of the project was to build a usable system. That is, the software, the hardware, and the user interface should work together to aid the computer's user not slow him down. We wanted to test whether a speech recognizer is a desirable and practical input device for current computer systems. The system was designed so that experiments could be performed to analyze both system components (i.e. the recognition algorithm, the user interface, etc.) and the overall system performance.

INTRO.5. Previous Work

Several commercial speech recognition systems have also been interfaced to workstations (generally an IBM PC or Apple II). These include systems from NEC (model SR-100), Interstate Voice Products (model SRB), Texas Instruments, Keytronics (model KB 5152V), and Votan (model VPC 2000). While the speech recognition algorithms and hardware for these systems differs widely (and are generally proprietary), they are otherwise all very similar. Each system can recognize about 200 words with a latency of between 0.2 and 0.5 seconds. The recognition accuracy quoted for these systems is between

98% and 100% although no independent tests have been performed on these particular systems. The user interface provided by each system is minimal: no feedback is given to the user unless a word is recognized correctly.

The vocabulary for an application is fixed by the application's designer and no new words can be added by the user. This inflexibilty limits the use of these recognizers to few applications in an engineering workstation environment. These recognizers are trained off-line using a special program. Words can be retrained only with the aid of this special program, and because the host computer can run only one program at a time, any modification of templates must also be performed off-line.

The interface provided to application programs is also the same for all the systems. Each word has a corresponding string that is sent to the computer when the word is spoken. This technique is very popular but, as explained earlier, is not very appropriate for speech input.

These speech recognition products are not transparent and complete enough to the user that they eliminate the need for keyboard input. With the exception of a few well suited applications, these systems are very cumbersome to use and have not been a large commercial success.

INTRO.6. Organization of the Thesis

The thesis is split into six chapters. The first two chapters detail the algorithm and hardware design. The third chapter explains the user interface and the interface provided for application programs. The fourth chapter discusses four different interface styles and how the speech recognizer can be used for each of these. The fifth chapter discusses an experiment performed to evaluate the entire recognition system for a particular application. The final chapter contains conclusions drawn from the project.

Chapter 1 - The Algorithm

1.1. Introduction

Currently, there are two types of practical high accuracy speech recognition algorithms: template based dynamic-time-warp algorithms and Markov-model-based algorithms. Dynamic-time-warp algorithms have been in laboratory use since 1970,[3,4,5,6] and have proven useful in commercial applications as well. Markov-model-based algorithms have also been in existence for many years, [7,8] but have only been used commercially with small vocabularies.[9] This is due the the large amount of training, especially initial training, required for Markov-model algorithms. Only recently have practical moderate to large vocabulary Markov based systems been introduced commercially.[10]

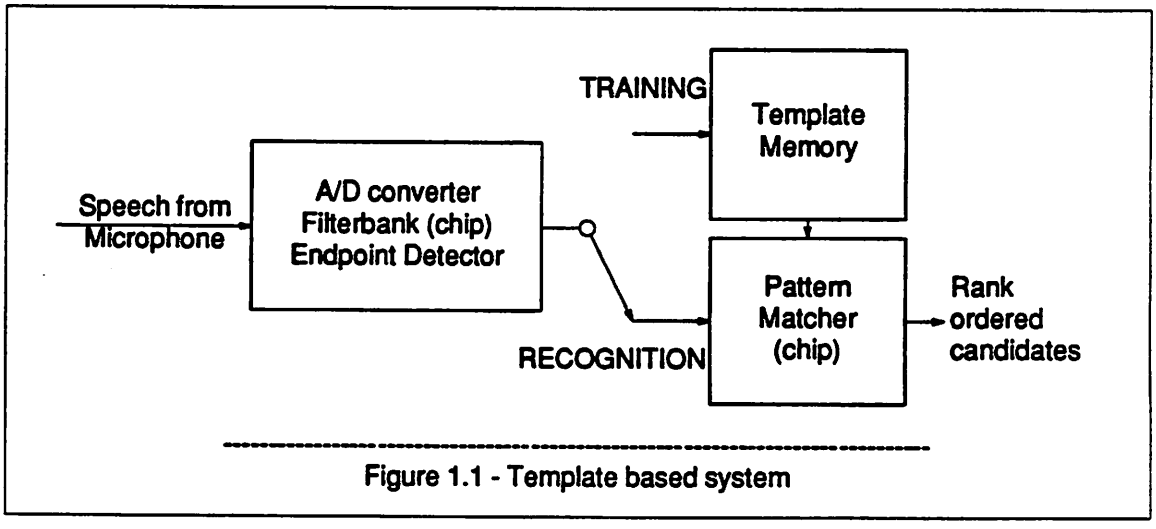


Figure 1.1 - Template based system

Over the past few years, the design of a sophisticated real-time isolated-word speech recognition algorithm capable of recognizing 1000 words has been developed in our laboratory. Since this algorithm is sufficient for the design goals of the Mara system, we decided to use the template-based algorithm. The basic structure of the template-based dynamic-time-warp algorithm is typical of pattern matching systems, representing each word as a collection of templates (normally just one template) which correspond to different pronunciations of that word. The algorithm has two main phases: training and recognition. During training, each word in the vocabulary (that is, the list of words to be recognized) is

spoken a few times by the user and templates are generated for the word by averaging those utterances. During recognition, each template is compared to the unknown spoken word. The vocabulary word template with the smallest word-to-word distance to this unknown word is recognized as the word that was spoken. The output of the recognizer is an ordered list of templates and their scores.

The algorithm is naturally split into two computational sections: front end spectral/word analysis and a word-to-word comparison algorithm. The spectral/word analysis section takes the speech signal as input and first generates a stream of vectors that represent spectral features (i.e. the log power spectrum). It then divides the input stream into words, each word beginning and ending in silence. A word is a time-ordered sequence of feature vectors. A training algorithm takes these words and generates templates. The user is instructed to speak each word a few times and these utterances are clustered into groups corresponding to major pronunciation differences, then each group of words is averaged to generate a central template. The word-to-word comparison algorithm takes as input two words (a template and the unknown word) and generates the smallest possible accumulated spectral distance allowed by a non-linear stretching and compressing of the time axes of the two words. The spectral distance used is the squared Euclidean distance between gain-normalized log-spectral components.

1.2. Front-End Processing

The purpose of front-end processing is to extract words from the incoming speech signal and derive a time sequence of short-time spectra for each word. There are many different spectral representations that are known to work well for speech recognition. We chose a direct spectral representation generated by a filterbank. A pre-emphasis is applied to the signal before the filterbank, one zero at 500 Hz and one pole at 5000 Hz. The filterbank contains 16 channels, each channel consisting of a 4 pole band-pass filter, a full-wave rectifier and a low-pass filter. The 3 dB corner frequencies of the band-pass filters are placed according to the "critical band" theory: linearly spaced filter up to 1KHz, and logarithmically spaced above 1KHz. The three-pole low-pass filter (25 Hz 3dB frequency) windows the speech signal with a 40 millisecond IIR window.

Table 1.1 - Measured Filterbank Specificiation

channel	center freq (Hz)	lower band edge (Hz)	upper band edge (Hz)	Q
1	126	76	227	.83
2	261	277	345	1.5
3	464	362	548	2.5
4	682	564	767	3.4
5	901	767	986	4.1
6	1053	969	1171	5.2
7	1306	1188	1407	5.9
8	1559	1441	1728	5.4
9	1880	1627	2166	3.5
10	2352	2015	2504	4.8
11	2689	2386	2824	6.1
12	3094	2841	3280	7.0
13	3515	3279	3768	7.2
14	3903	3701	4276	5.8
15	4645	4359	5101	6.3
16	5523	5202	5758	9.0

The outputs of the filterbank, all 16 channels, are sampled every 10 milliseconds and log-converted to 8 bits per filter, 0.375 dB per step. Then the log-filters are averaged to form an in-band energy estimate used for energy normalization and end-point detection. Also, the peak filter value is computed for use later.

$$S_i^{\log} = 16 \log_2 (S_i^{\text{linear}}) \quad i \in \{1..116\} \tag{1.1}$$

$$Ave = \frac{\sum_{i=1}^{16} S_i^{\log}}{16} \tag{1.2}$$

$$Peak = \max(S_1^{\log}, S_2^{\log}, \ldots, S_{16}^{\log}) \tag{1.3}$$

The log-spectrum is energy normalized by subtracting the average energy estimate, *Ave*, from each log filter, then quantized to 4 bits per filter, 2 dB per step. This 4 bit quantization was found to be sufficient in the sense that finer quantization steps led to no increase in recognition accuracy.[11] The quantization function is linear between +15 dB and -15 dB, clipping above and below.

$$S_i = Q\,[S_i^{\log} - Ave\,] \quad i \in \{1..16\} \tag{1.4}$$

The result is a frame consisting of the normalized log-spectrum (4 bits/filter), log in-band energy estimate (8 bits), and log peak estimate (8 bits), of comparable size to LPC parameterizations.[12]

The front-end processing then splits the incoming speech signal into words using a 3-level end-point detection algorithm. The algorithm tags each frame as one of:

EP_IGNORE	frame not in a word (energy below low threshold)
EP_START	possible start of word (energy transisition from below to above low threshold)
EP_NULL	frame in a word
EP_END	possible end of word (energy dips below low threshold)
EP_FOUND	word found - use last EP_START and EP_END as real start/end of word (.2 seconds of silence)
EP_SENTCEND	end of sentence was encountered (1.5 seconds of silence)

The end-point algorithm is similar to the one proposed by Rabiner and Sambur[13] and modified by Davies.[14]

The end-point algorithm uses three threshold levels: a low, mid, and high threshold. The low and mid thresholds are set dynamically 3 dB and 10 dB above the background noise level. The high threshold T_{high} is set to a constant value much greater than the noise level. The noise level is estimated using a non-linear filter that is applied to the average in-band energy Ave.

$$\begin{aligned} &\text{if } Ave > Noise \\ &\text{then } Noise = Noise + \frac{1}{64} \\ &\text{else } Noise = \frac{Ave + Noise}{2} \end{aligned} \tag{1.5}$$

The last step in front-end processing is down-sampling the 10 millisecond frames to 20 milliseconds. A "selective" down-sampling scheme similar to "frame repeat" in vocoders is used. Each frame is compared to its previous frame. If the spectral distance is less than a threshold $T_{downsample}$, then

the frame is ignored.

$$\begin{aligned} &\text{if } \textit{distance}\,(S, S^{\textit{previous}}) \geq \mathrm{T}_{\textit{downsample}} \\ &\text{then } S^{\textit{previous}} = S \\ &\text{else } \textit{ignore } S \end{aligned} \tag{1.6}$$

The distance function is defined in section 1.4. The purpose of this down-sampling is to weigh long steady state sounds equally with the transitions between sounds (i.e. steady state sounds should be downsampled). This algorithm has been shown to increase recognition accuracy while reducing computation.[11,15] The threshold is set to achieve a net 20 millisecond sample rate (factor of 2 compression) for a large number of words. Individual words may be downsampled more or less than the factor of two.

The output of the front end is a time series of frames representing a word.

$$(U^n(1), U^n(2), \cdots U^n(n)) \tag{1.7}$$

The notation above denotes a series of frames that form the word U of length n. Each $U^n(x)$ is a spectrum with 16 features written as $U^n_1(x) \cdots U^n_{16}(x)$.

1.3. Training

Template-based speech recognizers must be trained. The training process normally interacts directly with the user, prompting him to speak a particular word over and over. Training can be thought of in two distinct parts, acquiring utterances of a word from the user and creating templates using those utterances. This section will deal with the second problem, creating the templates. The problem of acquiring the words will be discussed in chapter 3.

Given utterances for a word, we wish to generate a set of templates which spans those utterances. There is no guarantee that all the utterances will be from the word to be trained; users do not always say what they are told to say! The training algorithm splits utterances into groups corresponding to distinct pronunciations of the word, then averages each group to create one template per group.

Words are split into groups using the UWA[16] (Unsupervised Without Averaging) algorithm. All word-to-word distance measures are computed to form a matrix. The algorithm proceeds:

1. Find the overall "average" word over all remaining words.
2. All words within some distance threshold T_{train} of this average word are collected.
3. Find the "average" word in this group. Iterate to step 2 until the group remains constant.
4. The words in the group are removed from the list of words to form a pronunciation group. Iterate to step 1 until all words are removed.

The UWA algorithm defines the "average" of a group of words as the mini-max center of that group. The mini-max center is the word whose largest distance to any word in the group is smallest. The output of the algorithm is a set of pronunciation groups. Each group must have at least two members, eliminating templates for misspoken words.

Next, each pronunciation group is averaged to form a template. To average words a target word is chosen, in this case the mini-max average word. All other words are time-warped (described in the next section) to this target word. For each frame of the target word, all corresponding frames of the warped words are averaged to form the template. The average of a set of frames is computed by averaging each feature in the frame seperately. A detailed description of the averaging algorithm can be found in [11]. The UWA algorithm assumes that training is performed in "one shot". That is, all utterances are generated, then templates are computed, and then the utterances are thrown away. We found experimentally that this "one shot" training is insufficient for high accuracy, so a new training algorithm was developed. The new algorithm takes new utterances of a word while the user is actually using the recognizer, and creates new templates adaptively. This algorithm is detailed in chapter 3. The UWA training algorithm is used when a large vocabulary must be trained, normally when the user starts a new application.

1.4. Word-to-Word Comparisons

The underlying computation in a template-based recognition system is the word-to-word distance measure. This distance is computed between a template and an utterance of a word. The smaller the distance, the more similar the words (that is, the distance is an error measure). The word-to-word distance measure is computed as a sum of frame-to-frame distance measures. The frame-to-frame distance we use

is the squared Euclidean distance between the energy-normalized log-spectra generated by the front-end processor.

$$distance(U,T) = \sum_{i=1}^{16} (U_i - T_i)^2 \tag{1.8}$$

This distance measure is known to work well in speaker-dependent recognition systems.[5]

/siiks/ #1	s	→	→	i	i	i	k	→	s	→
/sssiikkss/ #2	s	s	s	i	→	i	k	k	s	s
frame to frame distances	3	4	5	9	6	3	7	3	8	2

Figure 1.2 - Alignment of two utterances of /six/

Many word-to-word distance measures have been proposed, but the most popular and currently the most accurate is the dynamic-time-warp measure.[4,5] This measure is expensive, but works well with limited training. The time-axes of the template and the utterance are stretched to minimize the sum of frame-to-frame distances. For example, in figure 1.2 two utterances of the word "six" are aligned so that the sounds of one utterance ("s", "i", "k", and "s") are compared the same sound of the other utterance. The alignment adds sounds to the utterance by repeating the sound from the previous frame (→). The total distance is the sum of frame-to-frame distances between the aligned utterances. The time-warp algorithm minimizes this distance over all possible alignments using a computionally efficient dynamic programming algorithm. The result is the popular dynamic-time-warp algorithm.

The algorithm implemented in our system evaluates the following dynamic programming recursion equation:

$$d_{i,j} = \sum_{k=1}^{16} (U_k^n(i) - T_k^m(j))^2 \tag{1.9}$$

where U is the *Unknown* word, and

T is the *Template* word;

$$D_{i,j} = \min(D_{i-1,j}, D_{i-1,j-1}, D_{i,j-1}) + d_{i,j} \tag{1.10}$$

$$\text{where } D_{0,x} = \infty; D_{x,0} = \infty \text{ for all } x \text{ except:}$$

$$D_{1,0} = 0$$

Values of $d_{i,j}$ are clipped at 255 to limit the effects of single frame errors. The word-to-word distance is $D_{n,m}$ for an unknown word of length n, and template of length m.

s	3	5	8	50	52	48	32	8	2
k	28	27	31	80	82	7	3	25	28
i	52	60	54	12	3	60	62	39	41
i	40	49	51	6	12	68	69	45	45
i	45	48	39	9	8	78	82	50	52
s	3	4	5	40	39	20	20	10	8
	s	s	s	i	i	k	k	s	s

time →

Figure 1.3 - d matrix comparing two utterances of /six/

The equation is usually thought of in terms of a matrix of frame-to-frame distance measures d shown in figure 1.3. The horizontal axis is the time axis of the unknown word U, and the vertical axis is the time axis of the template T. The algorithm then attempts to find the path that starts from the lower left corner and ends at the upper right corner with the smallest sum. The path may not go backwards and cannot skip either a column or a row. In order to compute the sum, a second matrix of accumulated D values is computed. The smallest sum will be in the upper right corner of the D matrix.

s	171	140	147	181	164	85	69	48	50
k	168	135	139	131	112	37	40	65	93
i	140	108	151	51	30	90	152	191	232
i	88	97	97	27	33	97	166	211	226
i	48	51	46	21	29	107	193	181	193
s	3	7	12	52	91	111	131	141	149
	s	s	s	i	i	k	k	s	s

time →

Figure 1.4 - D matrix comparing two utterances of /six/

1.5. Streaming and Block Algorithms

To compute the recursion equation all the previous row and column values of D must be computed; however, not all of these values must be stored. If the equations are computed in column order then only the previous column must be stored, and similarly for the rows. Selecting between column and row ordered computation is a critical factor in the design of the algorithm. Real time is traveling along the horizontal axis of the matrix, thus the algorithm can compute one column of each template after the front-end generates that spectrum. Therefore, column-ordered computation is a streaming algorithm, where computation is overlapped with data acquisition in real time. On the other hand, a row-ordered system must wait for all frames of a word to be spoken before the word-to-word distance can be computed. Thus row-ordered computation is a block algorithm, where real-time must be defined in terms of "reasonable response time", usually 0.2 to 0.5 seconds after the word is spoken. In connected word applications or applications using long words, throughput for the block algorithm must be greatly increased to maintain reasonable response time, while a streaming algorithm requires no increase in throughput.

The selection between streaming and block algorithms effect both throughput and memory requirements. During recognition, all templates must be compared with an unknown word. Since a streaming algorithm must be able to compute one column of a matrix at the frame rate, the throughput in term of computing equations 2 and 3 must be (assuming 1000 templates with an average length of 25 frames, 0.5 seconds):

$$\frac{1000\ templates \times 25\ \frac{equations}{template}}{20\ milliseconds} = 1{,}250{,}000 \frac{equations}{seconds}$$

When converted to typical general purpose instructions, this becomes 60 million instructions per second. Obviously, this is not possible with available general purpose processors, but can be achieved with a special purpose chip if the proper architecture is used.

A block algorithm must be able to compute all 1000 matrices in the required response time (300 ms. in this example). Thus the throughput required is:

$$\frac{1000\ templates \times 25^2\ \frac{equations}{template}}{300\ milliseconds} = 2{,}083{,}333 \frac{equations}{seconds}$$

The block algorithm requires higher throughput than the streaming algorithm. Note further that if the streaming algorithm were allowed the same response time as the block algorithm, its throughput would decrease by about 40%.

The memory requirements for the system are not small. Both streaming and block algorithms require the same amount of template memory:

$$\frac{1000\ templates \times 25\ \frac{frames}{template} \times 16\ \frac{features}{frame} \times 4\ \frac{bits}{feature}}{8\ \frac{bits}{byte}} = 200{,}000\ bytes$$

The streaming algorithm must store D values for the previous column of all templates in a scratch-pad memory:

$$\frac{1000\ templates \times 25\ \frac{frames}{template} \times 24\ \frac{bits}{frame}}{8\ \frac{bits}{byte}} = 75{,}000\ bytes$$

while the block algorithm stores D values for only one template which requires only about 150 bytes. Since the amount of template memory required forces the use of off-chip memory, the addition of an off-chip scratch-pad memory is not a critical design factor.

The final step in the computation of word-to-word distances is the normalization of the distances. Since each template has a different length, the number of frame-to-frame distances that will be summed is different. This will bias scores for short templates to be smaller than scores for long templates. To eliminate this bias, the sum is normalized by dividing it by the number of frame-to-frame distances that were summed along the path. Instead of the exact path-length, the maximum of the unknown word and template length is used. This normalization factor is easier to compute, and works better than the real path length.[11] The reason for this might be that this factor will give preference to diagonal paths, implementing a sort of slope constraint. The normalization is not performed on the special purpose chip

because the computation is performed only once per template and can be computed in the controlling processor. The normalized scores are then sorted in ascending order and presented to the host computer.

1.6. Evaluation of the Algorithm

Table 1.2 System Performance with TI Database

speaker	errors (out of 320)	% errors
ALK	1	0.31
DFG	0	0.00
GRD	3	0.93
JWS	2	0.62
MSW	0	0.00
RGL	0	0.00
SAS	2	0.62
TBS	0	0.00
CJP	1	0.31
GNL	0	0.00
HNJ	2	0.62
KAB	0	0.00
REH	0	0.00
RLD	1	0.31
SJN	0	0.00
WMF	1	0.31
TOTAL	13	0.25

Table 1.3 Speech Recognition System Comparisons

manufacturer	model	total errors
Verbex	1800	10 (0.2%)
UC Berkeley		13 (0.25%)
NEC	DP-100	60 (1.2%)
Threshold Technology	T-500	73 (1.4%)
Interstate Electronics	VRM	147 (2.9%)
Heuristics	7000	300 (5.9%)
Centigram	MIKE 4725	366 (7.1%)
Scott Instruments	VET/1	646 (12.6%)

The algorithm was evaluated using a standard database collected by Texas Instruments.[17] The database consists of 20 words (the digits and 10 other common words) spoken 26 times each by 16 people (8 male, 8 female), recorded on 1/4 inch audio tape. The speakers were prompted to say each word. The

first 10 words for each speaker were used to train the recognizer, the remaining 16 words were used as test utterances. Table 1.2 shows the results of the experiment. Table 1.3 shows how these results compare to some commercial recognizers. It should be noted that the tests for the commercial recognizers were performed independently, while we performed our own tests.

Chapter 2 - Hardware

2.1. Introduction

There are constraints on speech recognition systems which make them very difficult to implement in general-purpose computers. First, all processing must be performed in real time. Thus one cannot afford page faults or use general purpose time-share schedulers. The recognizer must be running continuously (ie. it cannot afford to drop samples), and requires very small latency for reasonable response times. Second, signal processing tasks performed in the recognizer are CPU intensive. Generally these tasks are best performed with special signal processing CPUs such as the TMS320.[18] Also, some signal processing algorithms require special forms of arithmetic, such as saturating addition, which general purpose computers cannot handle efficiently. Third, speech recognition is considered a peripheral function to the computer; that is, the purpose of the computer is not to recognize speech, but to allow its users to perform tasks such as writing papers, performing simulations, and designing chips. The recognizer is the interface between the computer and its user, and thus it is in a tenuous position because it must be integrated into the user interface of the computer system, but not consume substantial or even moderate computational resources. The proper place for the speech recognizer is as a self-contained unit that connects to the host computer through a reliable bus. The advantages of this scheme are:

- Any computer can act as a host computer.
- The recognition algorithm can be designed independently of the hardware available in the host computer.
- The interface to the recognition system will be well defined and can thus be more easily studied and optimized.
- Special hardware can be used in the speech recognizer independent of the hardware and software in the host computer.

The disadvantages of the this scheme are:

- The system may not be as efficient due to the bandwidth constraint on the bus.

- System development and algorithm development might be hindered by an additional layer of interface.

For our system the advantages far outweighted the disadvantages, so the speech recognition system was designed as a stand-alone Multibus board which could be plugged into any Multibus host computer. Multibus was chosen because the target host computer had a built-in Multibus available for user peripherals, and the bus itself is reliable and has a high bandwidth.

2.2. Hardware Architecture

In order to provide a useful user interface to the host computer, the recognizer must be able to recognize between 500 and 1000 words in real-time. This requires enormous computational power, about 100 million instructions/second for a general purpose computer. This number is far beyond the capabilities of current microprocessors which operate at about 1 million instructions/second. Our solution to this computational problem was to develop special purpose integrated circuits as the computational elements, and use a general purpose microprocessor as a controller.

Given this design approach, the speech recognition algorithm is naturally partitioned as shown in figure 1.1. Data flows from block to block at the frame rate of 50 frames per second on average. Each block runs internally at a much higher rate (2-5 MHz). Thus, each block is implemented as a seperate sub-system, and connected together on a single microprocessor bus shown in figure 2.1. The board contains all three sub-systems: a general purpose system, a front-end spectral analysis system, and a dynamic-time-warp pattern-matching system. All sub-systems run in parallel, communicating to each other through the address and data bus lines provided by the general-purpose sub-system.

The circuit implementation requires many tradeoffs. First, board space is a key issue. There are many advantages of a single board solution over multi-board solutions: less power, less buffering, and most important less space is required in the workstation cabinet. This was very important because the workstation had space for only a single wire-wrap Multibus board. Multibus boards can hold approximately 150 16 pin chips, but much of this space is needed for memory (490K bytes requires 44 chips), the 68 pin 80186, its buffers, and the multibus and serial ports. The remaining space (about 30% of the

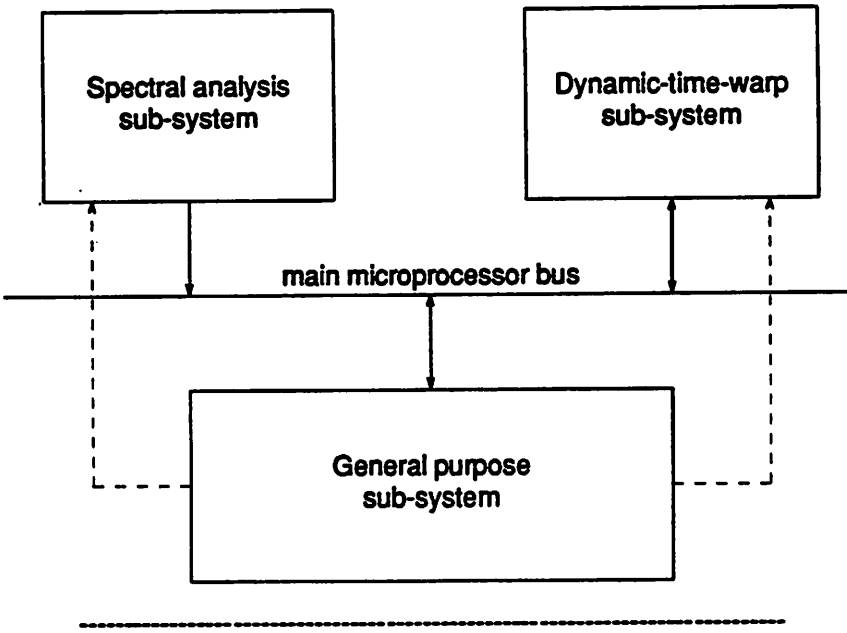


Figure 2.1 - Block diagram of speech recognition hardware

total board space) is left for the special purpose ICs and their associated circuitry. Thus, the special purpose ICs must include much of their interfacing circuitry on-chip, increasing the complexity and design time of the ICs. Second, all the special purpose chips had to be designed and tested within a reasonable amount of time or the system would never be built. Finally, since the board was going to be used for real work, the system had to be reliable. Thus the special purpose IC's had to work reliably at 5 volts, and the board had to function inside a number of different system configurations. The resulting Multibus board is fully packed, drawing 4.25 amps at 5 volts.

2.3. General Purpose Sub-System

The general purpose sub-system was designed to be used for multiple applications including: the speech recognition system, an IC tester system, a data acquisition system and other stand-alone applications. In order to be useful for speech recognition this sub-system had to:

- have 128K bytes of program and data memory,
- be able to address an additional 256K bytes of template memory,
- have a DMA port to read scores from the time-warp chip, and

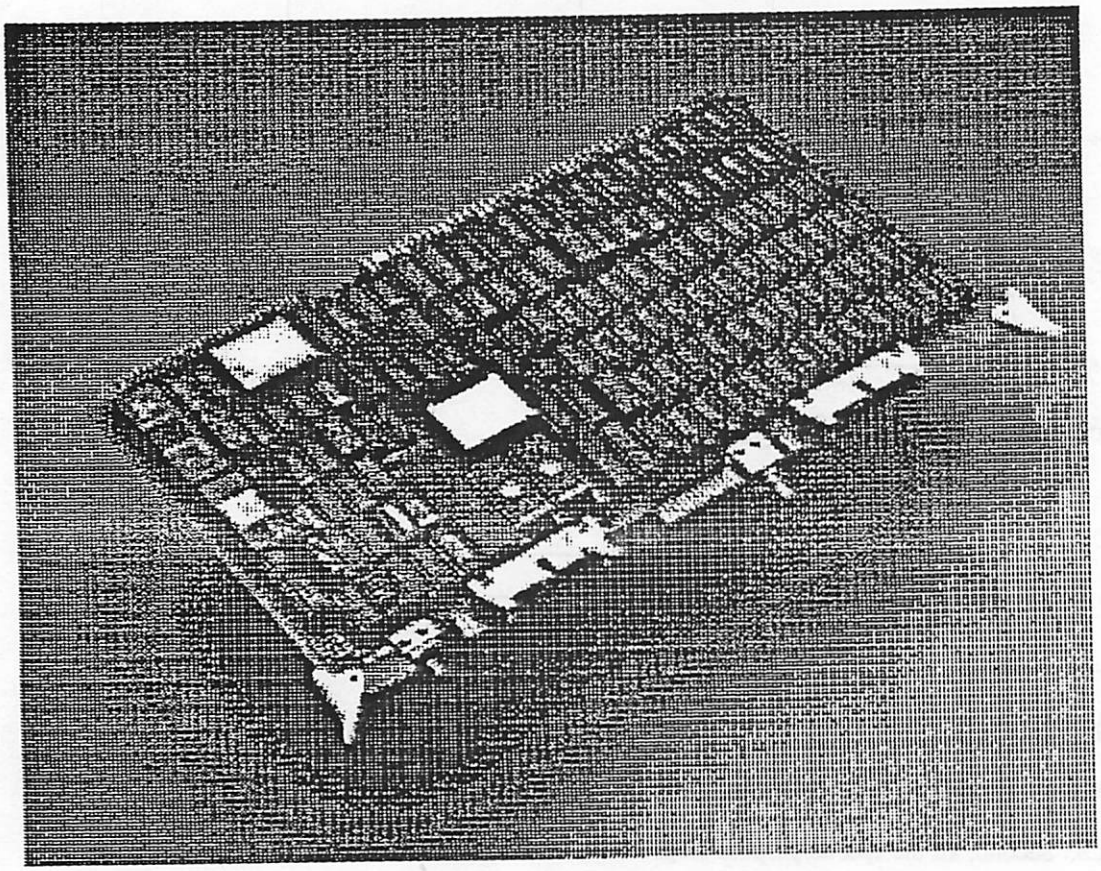


Figure 2.2 - Picture of the Multibus board

- execute divide instructions quickly (for distance normalization).

The Intel 80186 was chosen because in addition to the standard 1 MIPS CPU, it contains many of the extra support functions on a single chip. These include two DMA ports, three counter/timers, four external interrupt channels, and chip select logic. This reduced the size of the general purpose section of the board to less than one third of the Mulitbus board. The rest of the general purpose section of the board contains:

- two RS232C serial ports,
- a Multibus slave parallel port,
- a 64K by 16 bit memory, and

- a 2K by 16 bit EPROM memory.

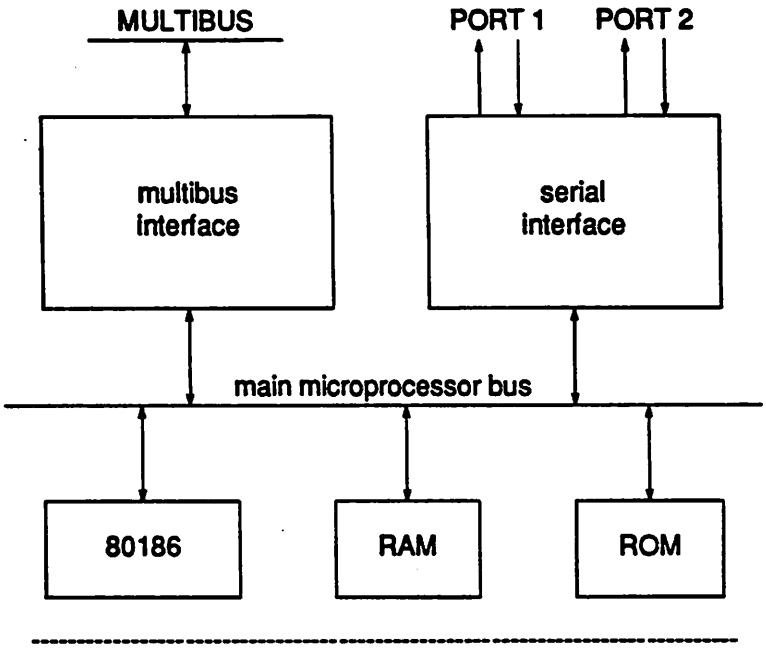


Figure 2.3 - Block diagram of general purpose sub-system

The memory is implemented using 16 standard 64Kx1 dynamic RAM chips. Refresh is performed in software. The two serial ports are used mainly for debugging, but can also be used to add different front-end processors to the board. For example, a digitizing pad can be attached to the board to form a character recognition system. There are three distinct port addresses in the Multibus memory space, one for bi-directional data, one for bi-directional control information, and one that when read by the host, resets the board (jumps to the EPROM monitor). Thus the board requires no external reset switch. The data port uses a fully interlocking handshake, while the control port does not. The internal data and address busses are buffered for use by the other sub-systems.

The EPROM contains a program that will load other programs from the host computer. The EPROM monitor listens on both the Multibus port and one serial port for a download command. After the command is received the follwing data is loaded into memory and executed. This strategy allows quick updating of the recognition program running on the 80186. 80186 programs are written in C, using a cross compiler from MIT.[19] In order to speed development, a simple operating system was written to

allow multiple co-operating processes, each associated with a different phase of the algorithm. This structure greatly simplified the software on the recognition board (see appendix A).

2.4. Spectral Analysis Sub-system

Due to limitations in the IC processing technology available for the digital filter spectral-analysis chip (a 5 volt, 4 micron NMOS technology with only poly-metal capacitors), the analog interface circuits including the anti-aliasing filter, sample and hold, and A/D conversion are performed with separate off-the-shelf integrated circuits. A CODEC (standard telephone A/D converter) could not be used here because the sample rate of 14K Hz was needed. Even so, the analog section of the board is only 5% of the board area. The anti-aliasing filter for the A/D converter are off-board, as are the microphone and its pre-amplifier. This was due to both lack of space, and fear that digital noise would sum into the high impedance nodes of the filter. The spectral analysis sub-system contains:

- an analog buffer,
- analog pre-emphasis,
- a sample and hold, and
- a 12 bit A/D converter.

The required filtering is normally a computational bottleneck for many speech recognition systems. The desired filterbank requires 81 poles of filtering, (i.e. 16 band pass filter channels (4 poles), 16 decimation filters (one pole) and 1 multiplexed low pass filter (two poles)), all operating at a 14KHz sample rate. The computations are reduced dramatically by using a canonical signed-digit coefficient coding scheme that reduces the total number of non-zero digits in each coefficient while only marginally changing the overall filter shape. Details of the chip implementation can be found in Ruetz.[20,21] The result of the coefficient coding scheme is that each multiply becomes one or two "shift and add" operations. Thus no multiply operations are required, reducing the complexity and size of the hardware. The filterbank is implemented on a single special-purpose integrated circuit. The chip takes as input the output of the A/D converter and outputs two channels each sample period. The output is decimated to 10 ms frames using a counter in the 80186 microprocessor. The net data rate is 16 channels every 10 ms. These samples are

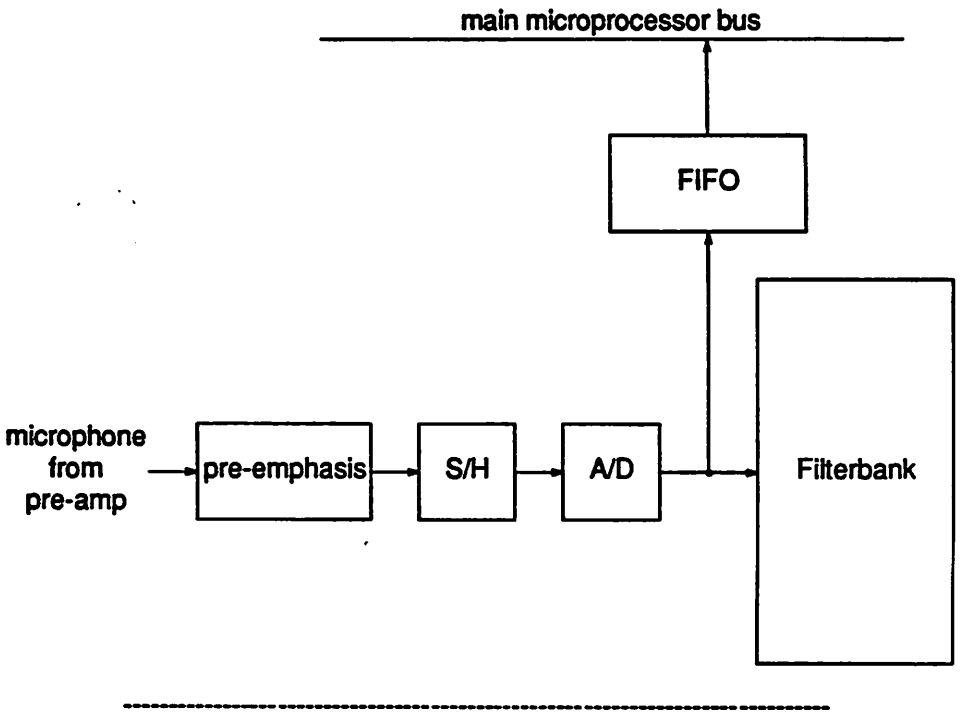


Figure 2.4 - Block diagram of spectral analysis sub-system

loaded into a FIFO and read by the 80186 every 10 ms (the FIFO generates an interrupt after all 16 channels are loaded).

The log-conversion, energy and peak estimates, endpoint algorithm, energy normalization and quantization, and selective down-sampling are all performed in the general purpose processor. Performing these operations in the 80186 requires little compute time, and allows fast and simple modifications to the front-end.

2.5. Dynamic-Time-Warp Sub-System

This sub-system computes one column of the D matrix for all templates, returning the accumulated distances on the top row of each template. This sub-system contains:

- 256K bytes of template memory,
- 96K bytes of scratch-pad memory,
- the dynamic-time-warp chip, and

- FIFOs to read the dynamic-time-warp chip distances into the 80186.

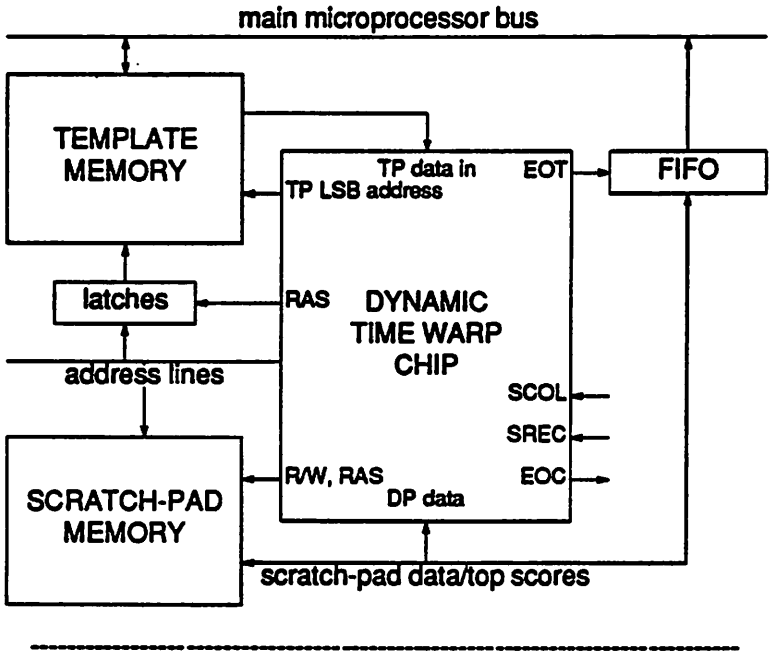


Figure 2.5 - Block diagram of dynamic-time-warp sub-system

2.5.1. Template Memory

The template memory is dual-ported, allowing access through both the 80186 and the dynamic-time-warp chip. The contents of the 16 bit wide template memory is organized as shown in figure 2.6. Address 0 contains the boundary value ($D_{x,0}$) for the unknown word axis of the time warp chip. This value is set to 0 for the first column, and ∞ for subsequent columns. The boundary value is set to other values for connected word algorithms. Addresses 1, 2, and 3 contain no information. Addresses 4, 5, 6, and 7 contain the current 64 bit frame (16 4-bit features) of the unknown word. Addresses 0, 4, 5, 6, and 7 are updated by the 80186 for each new frame. The rest of the template memory contains the actual templates stored sequentially with special frames as separators. A special frame with first word FFFF hexadecimal indicates end-of-template (EOT), with second word FFFF indicates end-of-column (EOC, i.e. end of all templates). The template memory is implemented with 32 standard 64Kx1 dynamic RAM chips, organized as a 64K x 16 x 2 interleaved memory. The memory is interleaved to cut the effective

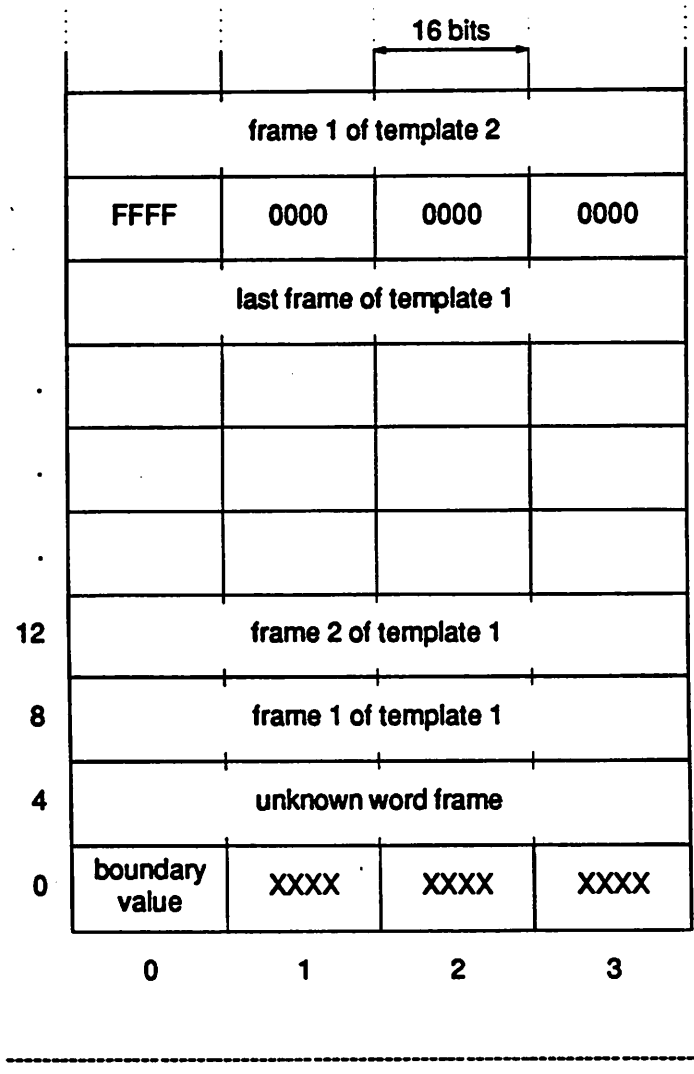


Figure 2.6 - Organization of template memory

cycle time of the dynamic RAM chips in half, below the required 200 ns. The dynamic-time-warp chip reads the template memory sequentially both while computing and idle. This continuously refreshes the dynamic memory.

The 80186 can directly read and write the template memory when the dynamic-time-warp chip is idle. This allows the 80186 to move, add, delete, and update templates quickly. In fact, using the DMA port provided by the 80186, templates can be added at the maximum speed of the bus (2M bytes/sec). The template memory appears as addresses 40000 through 80000 on the 80186 bus.

2.5.2. Scratch-pad Memory

The scratch-pad memory is constructed from 12 standard 16K x 4 dynamic RAM chips. This memory is organized as 24 bits x 32K words, allowing for a 15 bit accumulated distance, 1 bit of slope constraint information, and 8 bits of path length information (used in connected speech). The RAS and R/W signals for the memory are generated by the dynamic-time-warp chip, reducing the amount of extra glue logic on the board. Unfortunately, this memory cannot be read directly by the 80186 which complicates its testing. In the future the dynamic-time-warp chip should be modified to allow direct testing of the scratch-pad memory.

2.5.3. Time-Warp Chip

The heart of the dynamic-time-warp sub-system is the dynamic-time-warp chip. This chip performs the *d* and *D* distance computations, template and scratch-pad memory addressing, scratch-pad memory control generation, and state sequencing. The dynamic-time-warp chip is highly parallel and pipelined in order to compute the distance measure as fast as possible. This technique is well suited to special purpose design because the placement of pipeline registers and parallel units depends on the implemented equations.

The chip operates from a single 50% duty cycle 5MHz clock. Internally this clock is converted to:

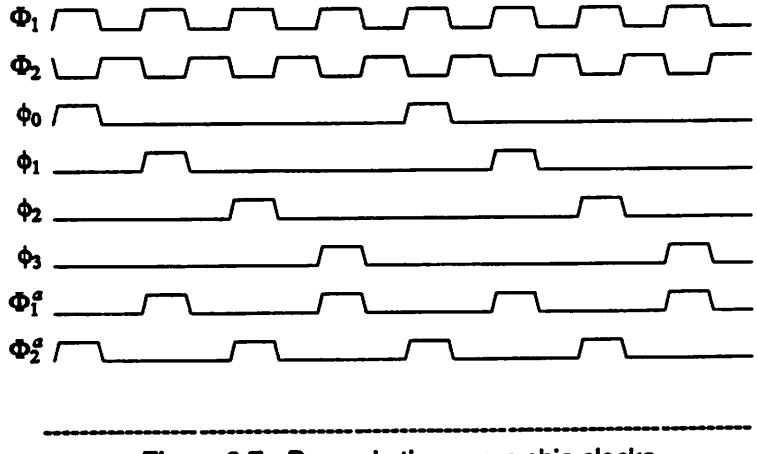


Figure 2.7 - Dynamic-time-warp chip clocks

- a two-phase non-overlapping 5 MHz clock (fast clocks: Φ_1, Φ_2),
- a four-phase non-overlapping 1.25 MHz clock (slow clocks: ϕ_0, ϕ_1, ϕ_2, ϕ_3), and
- a two phase non-overlapping 2.5 MHz address clock (Φ_1^a, Φ_2^a).

2.5.3.1. Circuit Operation

The chip starts in its idle mode, sequentially refreshing both the template and scratch-pad memories. When a start-first-column (SREC) or start-other-columns (SCOL) signal is received, the chip resets its address counter to 0 and reads the template memory sequentially. The first item read, the boundary value, is stored in the 16 bit PRESET register (15 bits plus one bit for slope-constraint). The next three items are ignored. The next four items read are stored in the UNKNOWN FRAME register, a recirculating shift register that outputs 16 bits of the 64-bit frame each fast clock cycle. Next, the first frame of the first template is read from the template memory, requiring four fast clock cycles. A pipeline register at the top of the data path allows maximum delay though the memory (about 160ns out of the 200ns clock). As each 16-bit word is read it is split into four 4-bit nibbles, and subtracted using four one's complement subtractors in parallel from the corresponding unknown frame nibble. These 5-bit differences are then absolute-valued to 4 bits then pipelined. Next, each 4-bit value is squared forming an 8-bit value and pipelined. These four 8-bit values are summed pairwise and pipelined to form two 9-bit sums, then summed again pairwise to form a 10-bit sum. The 10 bit sum is clipped at 255 to 8 bits and then pipelined again. The result is an 8-bit 4-dimensional squared Euclidean distance every 200 ns. The 8-bit values are then summed over the unknown frame (four cycles) using an 8-bit saturating accumulator producing the required 16-dimensional squared Euclidean distance $d_{i,j}$.

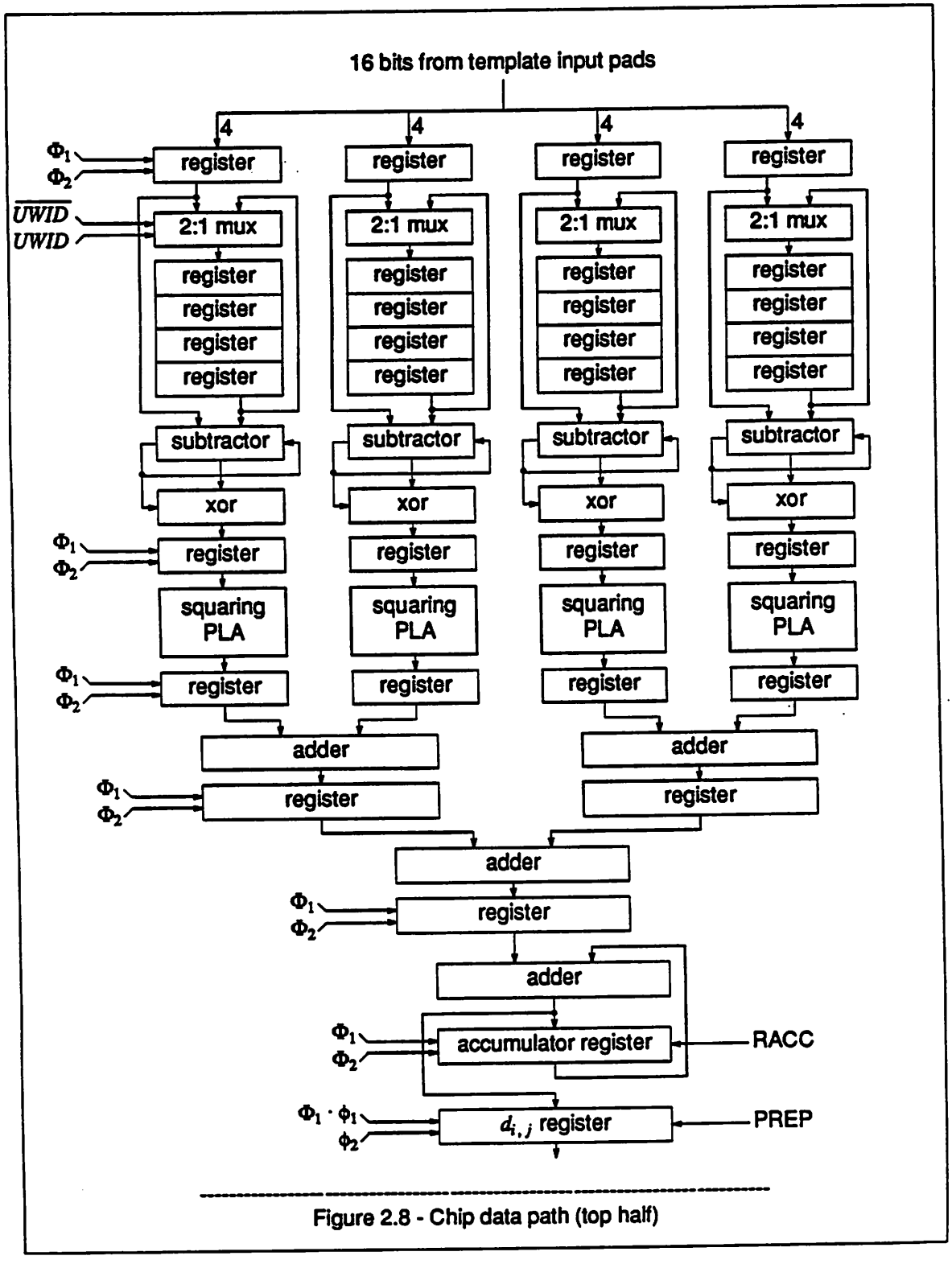


Figure 2.8 - Chip data path (top half)

$$d_{i,j} = \sum_{k=1}^{16} (U_k^n(i) - T_k^m(j))^2$$

The output of the accumulator is loaded into the $d_{i,j}$ register on ϕ_1 allowing for the five pipeline delays through the top half of the data path. The pipeline registers in the top half of the data path are clocked with the fast clock.

While the top half of the data path computes the squared Euclidean distance, the bottom half computes the recursion equation:

$$D_{i,j} = min(D_{i-1,j}, D_{i-1,j-1}, D_{i,j-1}) + d_{i,j}$$

The equation is computed using a data path specifically designed to compute this one function. The data path consists of:

- three 15-bit registers corresponding to $D_{i-1,j}$, $D_{i-1,j-1}$, and $D_{i,j-1}$,
- three comparators to compute $D_{i-1,j}<D_{i-1,j-1}$, $D_{i,j-1}<D_{i-1,j-1}$, and $D_{i,j-1}<D_{i-1,j}$,
- a PLA to select the minimum given the three comparisons above,
- a multiplexor to select one of $D_{i-1,j}$, $D_{i-1,j-1}$, $D_{i,j-1}$, or PRESET,
- a 15-bit saturating adder that adds the output of the multiplexor to $d_{i,j}$,
- a 16-bit PRESET register, and
- an FFFF detector (to detect EOT and EOC).

In addition to the operations above, the output of the $d_{i,j}$ register can be set to zero (NSEL input) and the output of the adder can be set to ∞ (PREP input). The input to the FFFF detector and the 16-bit PRESET register comes directly from the pads. This is a design error that might shorten the amount of time available to read from template memory as this is a critical path of the circuit. The input should come from the pipeline register right after the pads. The output of the FFFF detector goes into the state sequencer.

In order to understand how the recursion equation is computed one must remember that the computation proceeds up a column of the matrix. Thus, $D_{i,j}$ delayed one cycle becomes $D_{i,j-1}$. Boundary values are fixed (to the PRESET), thus $D_{i,j-1}$ and $D_{i-1,j-1}$ can be computed without going off-chip. The only value which must be read from the scratch-pad memory to compute $D_{i,j}$ is $D_{i-1,j}$.

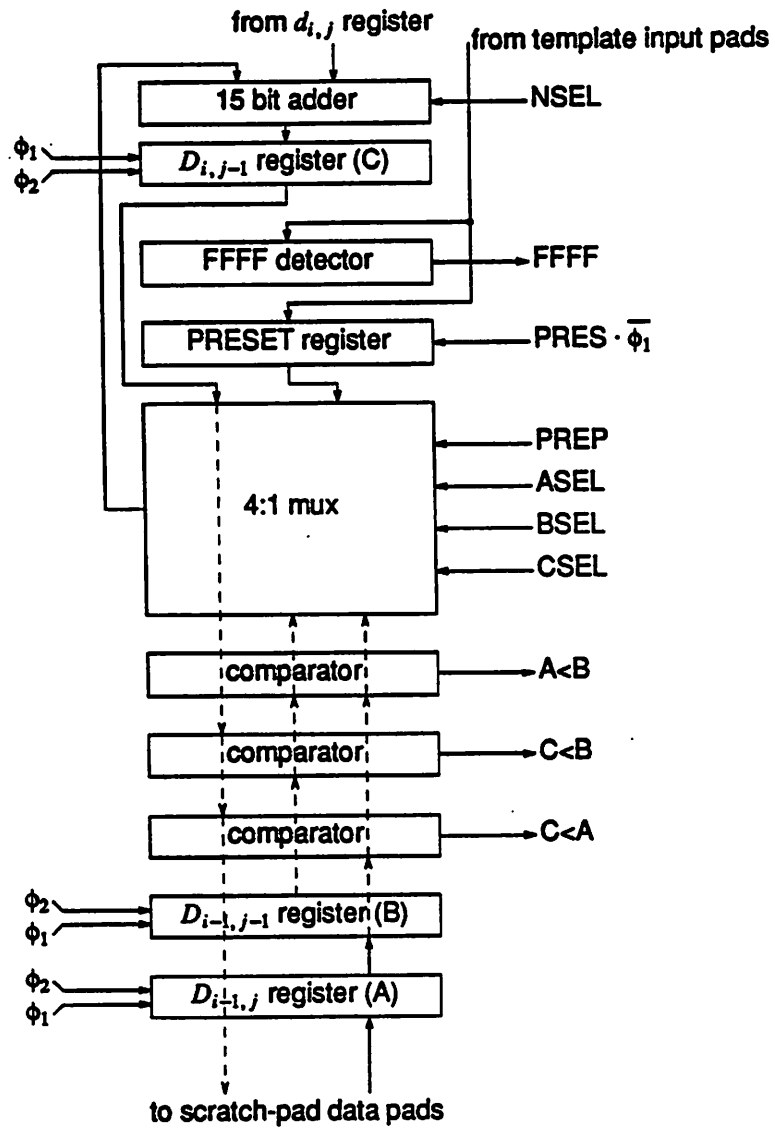


Figure 2.9 - Chip data path (bottom half)

The first row boundary condition (FROW) occurs at the start of a new template, causing the data path to compute:

$$D_{i,1} = min(\text{PRESET}, D_{i-1,1}) + d_{i,1}$$

by loading the PRESET into the $D_{i,j-1}$ register one slow cycle before $D_{i,1}$ is computed (called the PREP cycle). The minimization PLA equations are changed accordingly. FROW is an output of the state

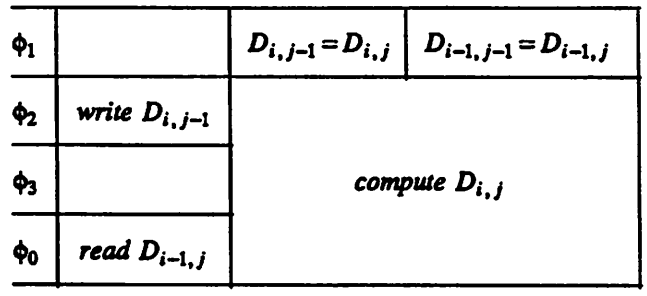


Figure 2.10 - Timing of recursion equation computations

sequencer, indicated by the special EOT flag in the template memory.

The top row distances (at EOT) are loaded into the off-chip FIFO for use by the 80186. During the first column (FCOL) the values read from the scratch-pad memory are ignored by a modification of the minimization equations to disallow recursions past the template word axis. During subsequent columns, the input from the scratch-pad memory is used to compute the recursion equation. Distances in the FIFO are read into the 80186 using a DMA port. When the end of all templates is detected, the chip strobes end-of-column (EOC) and returns to an idle state waiting for the next input strobe.

2.5.3.2. Address Generator

The addressing unit generates both the template and scratch-pad memory addresses. The scratch-pad memory is accessed every two fast clocks, first a write, then a read. The address for the write is one lower than the address for the read (i.e. first $D_{i,j-1}$ is written, then $D_{i-1,j}$ is read). Thus the address counter must count down one for the write, then up two for the read. This is accomplished with a adder/accumlator that selects between -1 and +2. The reset address (RADD) and address clock signals are generated by the slow/address clock generator.

While the scratch-pad memory is accessed every two fast clocks, the template memory is accessed every fast clock. The template memory is also accessed sequentially, not down one up two. In order to share the same address generator, an external latch is required to hold the address generated for the

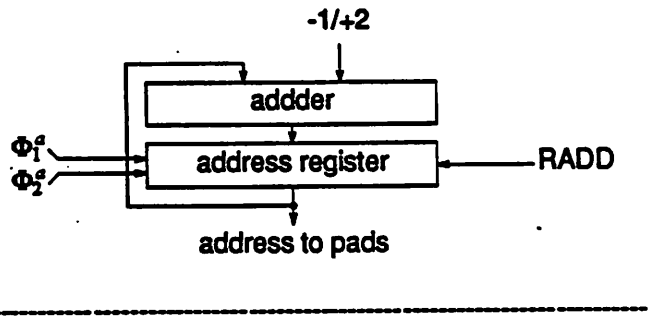


Figure 2.11 - Address counter

scratch-pad read cycle for 4 template memory read cycles. This latch is "free" because there is already a need for external address multiplexing for the dynamic memories (i.e. one can use latching multiplexors). The two least significant address bits of the template memory address are generated by the slow/address clock generator. These merely count from 0 to 3 continuously.

2.5.3.3. Clock Generators and Control Outputs

As mentioned above, the chip requires one clock input, but internally generates 8 clock signals. The two fast non-overlapping clocks are generated using a traditional multistage cross-coupled NOR gate. The output is buffered to drive the required 10pF in 40 ns. The slow clocks and address clocks are generated using a 2-bit binary counter and a decoder. The counter outputs go off-chip directly forming the least-significant template memory address, and are decoded internally (and ANDed by the fast clock to prevent glitching) such that ϕ_1 occurs when the counter output is 0. The clock generator also creates the output control signals RAS and R/W using random logic. Scratch-pad memory write cycles are suppressed when the chip is idle by not asserting the RAS signal (using the WRIH, write inhibit, signal). Finally, the slow clock unit generates the control signals for the address counter: Φ_1^a, Φ_2^a, and -1/+2.

2.5.3.4. Support for Connected Word Operation

Two modifications are required for connected word operation. First, the PRESET is set to any value, as opposed to only 0 and ∞ as in isolated word operation. This is done by the 80186. Second, the projection of the path length on the unknown word axis must be computed. This means that for each

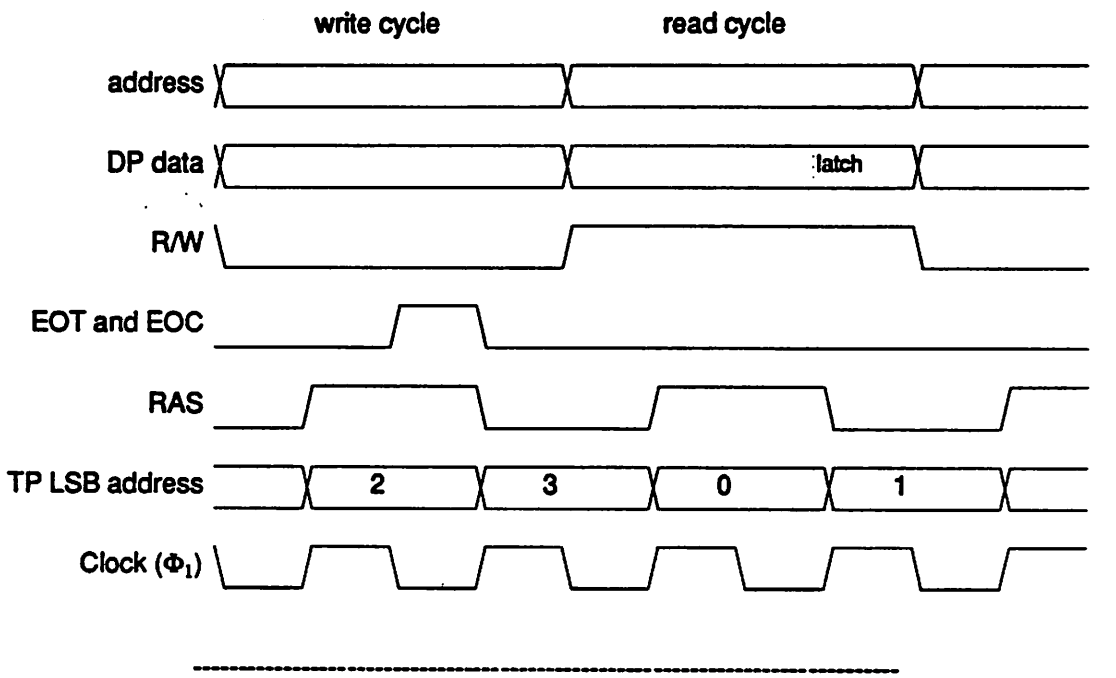


Figure 2.12 - Data, address and control signals (at pads)

$D_{i,j}$, the path length into that element of the matrix must be computed. The path length of $D_{i,j}$ is the path length of the smallest of $D_{i-1,j}$, $D_{i-1,j-1}$, and $D_{i,j-1}$, plus one if $D_{i-1,j}$ or $D_{i-1,j-1}$ is smallest. The path length of PRESET is set to zero on FROW by clearing the path length $D_{i,j-1}$ register. The path-length adder is 8 bits, saturating at 255.

2.5.3.5. Support for Slope Constraints

Many people feel that slope constraints can improve recognition accuracy[6] though our experience has indicated otherwise.[11] A simple type of slope constraint is implemented on the chip. This constraint specifies that if a horizontal or vertical path is taken into one element of the matrix, then the path leaving that element must be diagonal. Note that this form of slope constraint is not the same as the one proposed by Sakoe and Chiba because it does not gaurantee the result will be the path with smallest error (i.e. it no longer computes a true minimization). Never the less, the chip can handle this form of slope constraints if enabled by setting the most-significant bit of the PRESET register (bit 15) to 0.

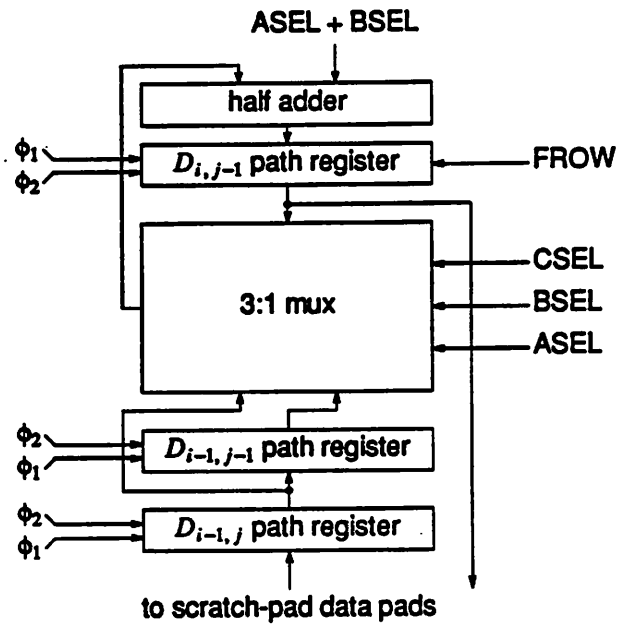


Figure 2.13 - Data path for path length computation

If $D_{i,j}$ is computed from $D_{i-1,j}$ or $D_{i,j-1}$ then the slope bit for $D_{i,j}$ is set low (active). The PLA that computes the minimum for the recursion equation then looks at the slope bit for $D_{i-1,j}$ and $D_{i,j-1}$ and disallows those paths if the bit is 0. The circuitry involved is minimal, requiring one extra bit in the word of the scratch-pad memory, one bit for each of the $D_{i-1,j}$ and $D_{i,j-1}$ registers and a modification to the minimization PLA.

2.5.3.6. State Sequencer

The data path is controlled by a state sequencing finite-state-machine (FSM). The FSM is aided by a circuit that delays input and output signals so that they occur at the correct time on the pipeline stages. The sequencer runs at the slow clock rate, with clocks ϕ_3 and ϕ_0. It has 5 active states: idle, load preset, load unknown word, first row, and other rows. The sequencer has three inputs: SCOL, SREC, and the output of the FFFF detector in the data path, in addition to the four slow clock phases. The output of the FFFF detector is clocked with the appropriate clock phases to determine if EOT or EOC has occurred. The sequencer generates nine outputs: EOC, EOT, FROW (first row), FCOL (first column), UWID (load

unknown word), $\overline{UWID}$, PRES (load PRESET register), PREP (generated one cycle before FROW), and WRIH (write inhibit in idle mode). The signals UWID, $\overline{UWID}$, and PREP are super-buffered due to the large capacitance they must drive.

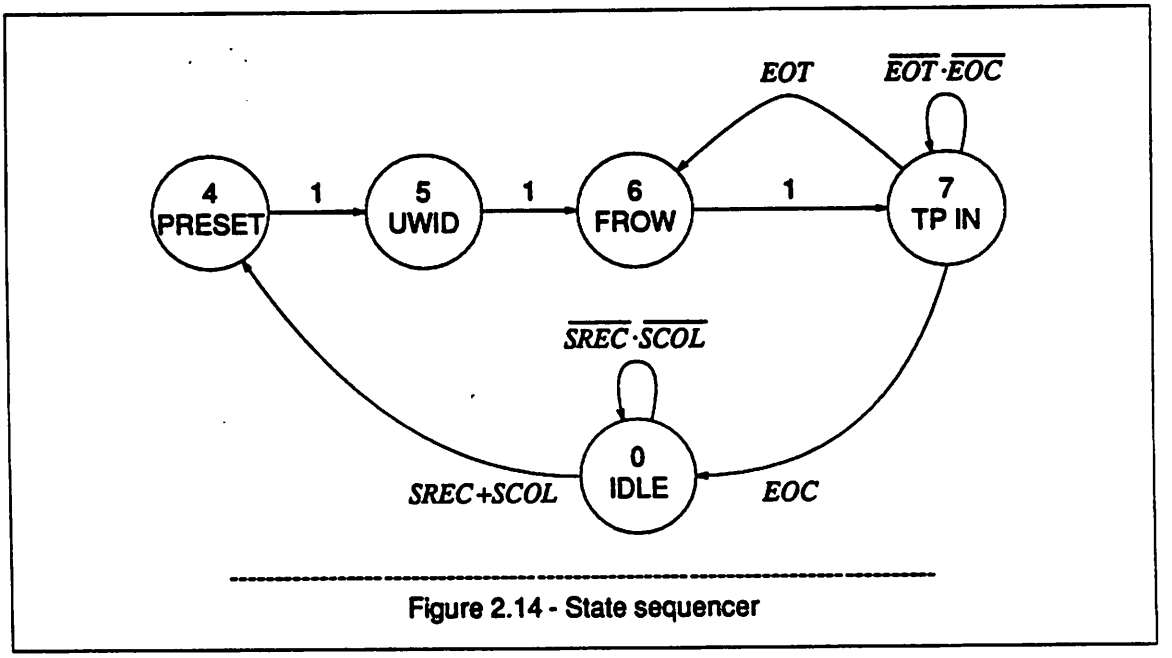


Figure 2.14 - State sequencer

2.5.3.7. Minimization PLA

In addition to the state sequencer above, the data path is also controlled by the minimization PLA. The PLA uses different minimization rules depending on FROW, FCOL, slope-constraints, and PREP (load PRESET into $D_{i,j-1}$ register). The PLA has 9 inputs, 4 outputs (super-buffered to the multiplexor select inputs), and one additional slope-constraint output. The PLA computes the the inverse of the CSEL signal reducing its size substantially. This signal drives an inverting super-buffer.

2.5.3.8. Circuits and Layout

The chip is designed with five basic building blocks:

- a delay register cell,
- two adder cells (even/odd bit slices) with optional saturating logic,
- two comparator cells (even/odd bit slices),

INPUTS				OUTPUTS				
FROW	FCOL	ASLP	CSLP	ASEL	BSEL	$\overline{CSEL}$	NSEL	SETCSLP
0	0	0	0	0	1	1	0	1
0	0	0	1	0	$\overline{C<B}$	$\overline{C<B}$	0	$\overline{C<B}$
0	0	1	0	$A<B$	$\overline{A<B}$	1	0	$\overline{A<B}$
0	0	1	1	$A<B\cdot\overline{C<A}$	$\overline{A<B}\cdot\overline{C<B}$	$\overline{A<B\cdot C<A}$ $+\overline{A<B}\cdot\overline{C<B}$	0	$\overline{A<B}\cdot\overline{C<B}$
0	1	X	0	0	0	1	1	1
0	1	X	1	0	0	0	0	0
1	0	0	X	0	0	0	0	1
1	0	1	X	$\overline{C<A}$	0	$\overline{C<A}$	0	1
1	1	X	X	0	0	0	0	1

Figure 2.15 - Minimization PLA rules

- PLA's, and
- three multiplexor cells (2:1, 3:1 and 4:1).

Some random logic is also used to implement various enable signals, clock generators and clock drivers. The adders and comparators use a non-precharged carry chain that has a 4 ns per bit delay.

The chip was implemented in a 4 micron NMOS process, has an active area of 20,000 square mils, and runs with a 5MHz clock. The chip was fabricated by MOSIS[22] and mounted in a 68 pin lead-less chip carrier.

2.6. Hardware Design Alternatives

Many research and commercial organizations have built speech recognition systems. The resulting

Table 2.1 - Time-Warp Chip Pins

Number of pins	Direction (I/O)	Name	Function
3	I	-	Power, Ground, Substrate
1	I	CLK	Clock (50% duty cycle)
16	I	TPIN	Input data from template memory
24	I/O	DPDATA	Data from/to scratch-pad memory
16	O	ADDR	Address lines (shared) for template and scratch pad memories
2	O	LSB	LSB address lines for template memory
1	I	SREC	Start first column
1	I	SCOL	Start other columns
1	O	EOC	End of column
1	O	EOT	End of template
1	O	RAS	RAS for scratch-pad memory
1	O	R/W	Read/Write for scratch-pad memory

hardware ranges from a few chips to an entire room full of computer equipment. Single-board large-vocabulary systems have become viable recently because of decreasing memory costs and LSI technology. The chips in these systems are often designed specifically for speech or signal processing. A practical speech recognition system must use either a single chip or a chip-set solution; otherwise the size of the hardware required would be prohibitive. A comparison of these speech-processing chips is difficult because each system uses a different speech recognition algorithm and hardware design. The design goals of a given chip therefore vary from company to company.

2.6.1. Filterbank

Filterbank design is a classic problem in signal processing. There are many different hardware configurations to perform filtering quickly and efficiently, implemented in both analog and digital technologies. Solutions to the problem range from single chips, to chip sets, to single boards, to entire computers. Most analog filterbanks use switched-capacitor designs. NEC has produced a 16/20 channel switched-capacitor analog filterbank for their speech recognition system[23] that includes a 9 bit A/D converter. It is not clear whether the chip actually works, or how it integrates into the rest of their system. Other analog filterbanks include the Reticon CP5016 and Interstate Voice Products ASA-16, both single

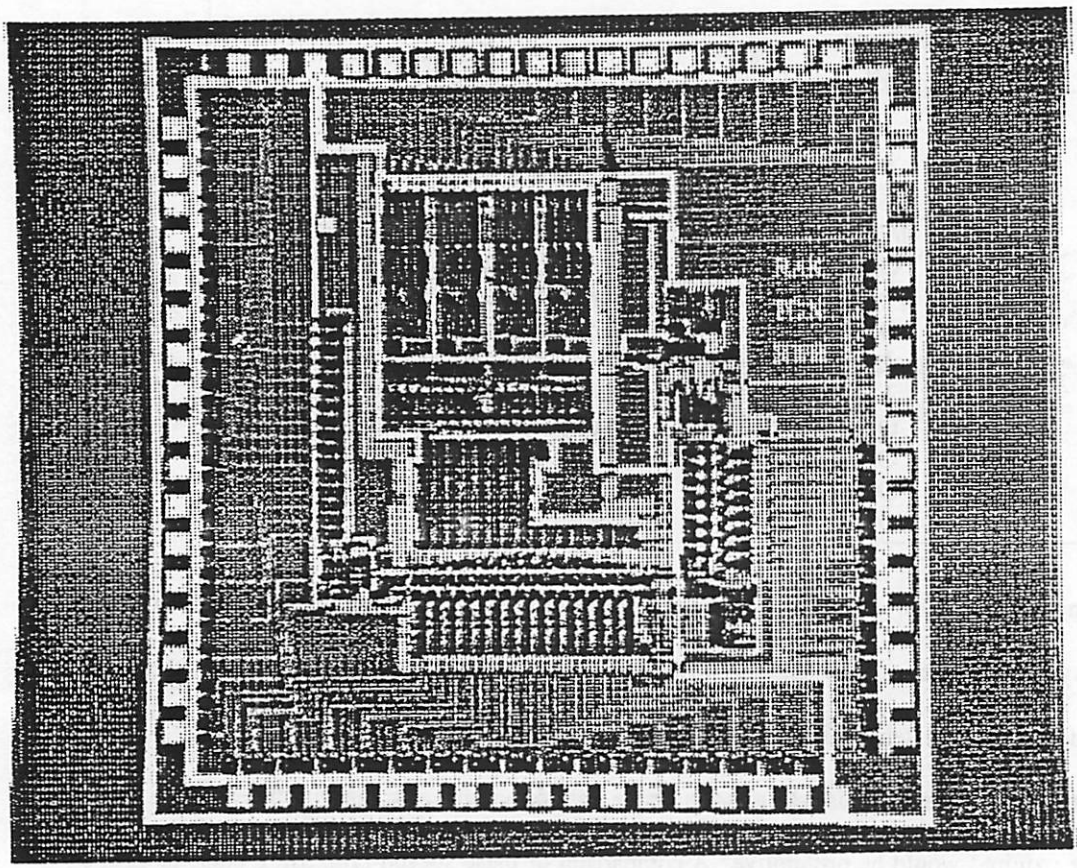


Figure 2.16 - Chip Photo

chip 16 channel filterbanks with no A/D converter. These chips are produced commercially. We initially attempted to use the Reticon chip, but found the Q of the filters to be too low for high accuracy speech recognition. The problem with the current generation of switched-capacitor filters is low power supply rejection. Special power supply filters are required which increases the board space required . The new generation of fully differential filters should solve this problem.

Many digital filter bank chips are also available, but these require more than one chip to perform 16 channels of filtering. The Texas Intruments TMS32010 (a general-purpose array-processor on a chip) can be programmed to form an 8 channel filterbank, thus two would be required for 16 channels. The 320 includes no A/D converter, thus a total of 3 chips would be required. The Intel 2920, an array-

processor with A/D and D/A converters has also been programmed as an 8-channel filterbank at a sample rate of 8KHz. This would be suitable for recognition over telephone lines. Recently, Kurtzweil Applied Intellegence has produced a chip[24] that can perform eight 2nd-order sections with 24 bit coeffecients and 48 bits of accumulation using a bit-serial approach. A 16-channel filterbank can be created from twelve chips. Again, no A/D converter is supplied on-chip. To date, there are no commercial single-chip digital 16 channel filterbanks.

2.6.2. Dynamic-Time-Warp Processor Alternatives

In recent years different architectures have been examined to implement the dynamic-time-warp algorithm. Both multi-chip and single chip implementations have been considered, as well as systolic arrays requiring up to thousands of chips. It is difficult to compare these chips fairly because each implements a different algorithm. Major algorithmic differences include:

- distance metrics (squared Euclidean, Chebyshev, LPC dot product),
- Minimization equation (various weightings, recursion in one dimension only),
- streaming/block computation
- pruning.

A good measure of speech recognition chips is the time and area required to compute the distance metric and minimization equation. Most trade-offs in chip designs come at this level. The Chebyshev distance (the sum of the absolute value of differences) is usually implemented instead of the squared Euclidean distance to reduce hardware. It is easier to take the absolute value than square a number, especially in an architecture with no multiplier. The LPC (Itakura-Saito) distance metric[25] is a different type of computation, the log of a dot product. This metric requires a variable-variable multiplier and a log conversion circuit, increasing chip area, though computation rate through the circuit can be as high as with the squared Euclidean distance if enough parallelism is exploited. Some LPC-based implementations use a simplier distance measure that does not require the log conversion. The minimization equation can also vary between algorithms, but in general the computation is the same: add the smallest of a few numbers to the spectral distance and accumulate. Using the basic equation computation time as a

comparison measure eliminates algorithmic variations such as number of spectra in a word, number of features in spectrum, and number of distances in the minimization.

2.6.2.1. General Purpose Chips

General purpose chips such as the MC68000[26] and the TMS320 have been used in many speech recognition systems. The TMS320 has been programmed to recognize up to 50 isolated words by implementing a simpler version of the dynamic-time-warp algorithm.[27] The chip performs both the LPC analysis and dynamic-time-warp matching. If only the matching is performed, about 100 words can be recognized. If compared using a standard algorithm the TMS320 cannot perform nearly as well, about 35 words in real-time.[28]

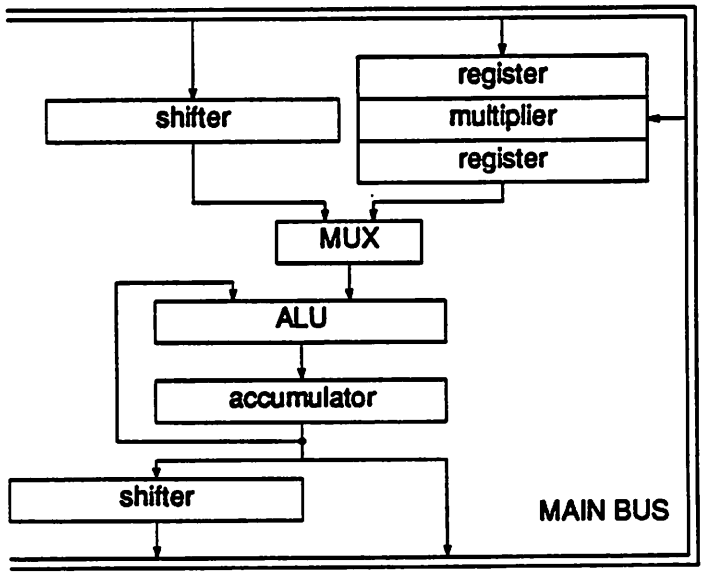


Figure 2.17 - Typical general purpose data path (TMS320)

General purpose architectures use a single arithmetic unit, usually an adder or multiplier/accumulator, multiplexing all computations through the one unit. Parallelism in an algorithm can only be exploited by using many general purpose processors. If an algorithm requires only a few bits for some signals, and more bits for other signals, the arithmetic unit must be as large as the largest bit width. The efficiency of the unit during the small bit width computation is very low. A custom

architecture can have small adders and large adders only where they are required, increasing chip area efficiency.

A general purpose processor can perform any computation, but performs few efficiently. For example, the TMS320 can only perform comparisons pairwise, thus to find the smallest of three numbers 12 cycles are required. In order to operate efficiently, algorithms must be changed to perform only those operations that are efficient. Special purpose architectures can implement any one algorithm efficiently, but cannot perform all algorithms.

2.6.2.2. NEC Chip

In order to increase throughput beyond that of general purpose architectures, NEC designed a chip[29] to perform their version of the dynamic-time-warp algorithm. The chip contains two processors, one to compute the distance metric and one for the minimization. The distance metric processor computes the sum of the absolute value of the differences (Chebyshev metric) between the template and unknown word. The processor is programmable and has a special ALU with two 8-bit adder/accumulators to increase throughput. The minimization processor is also programmable; microcode is down-loaded through a microprocessor interface. The processors have a limited instruction set and data path designed to compute the time-warp equations efficiently in 2.25 μs per equation. The chip also contains enough on-chip scratch-pad and template memory to compute one word-to-word distance. The chip can be programmed to recognize up to 180 isolated words by using pruning techniques or 20 connected words using the standard real-time algorithm. This chip is half way between a general purpose and special purpose architecture, allowing a few different recognition algorithms to be performed.

2.6.2.3. Bell Labs DTWP chip

One Bell approach to speech recognition hardware was to build a non-programmable special purpose chip[30] to implement their version of the dynamic-time-warp algorithm. Their algorithm was developed on general purpose hardware forcing the use of pruning techniques to increase throughput. Pruning techniques such as adjustment windows and global slope constraints reduce the total number of

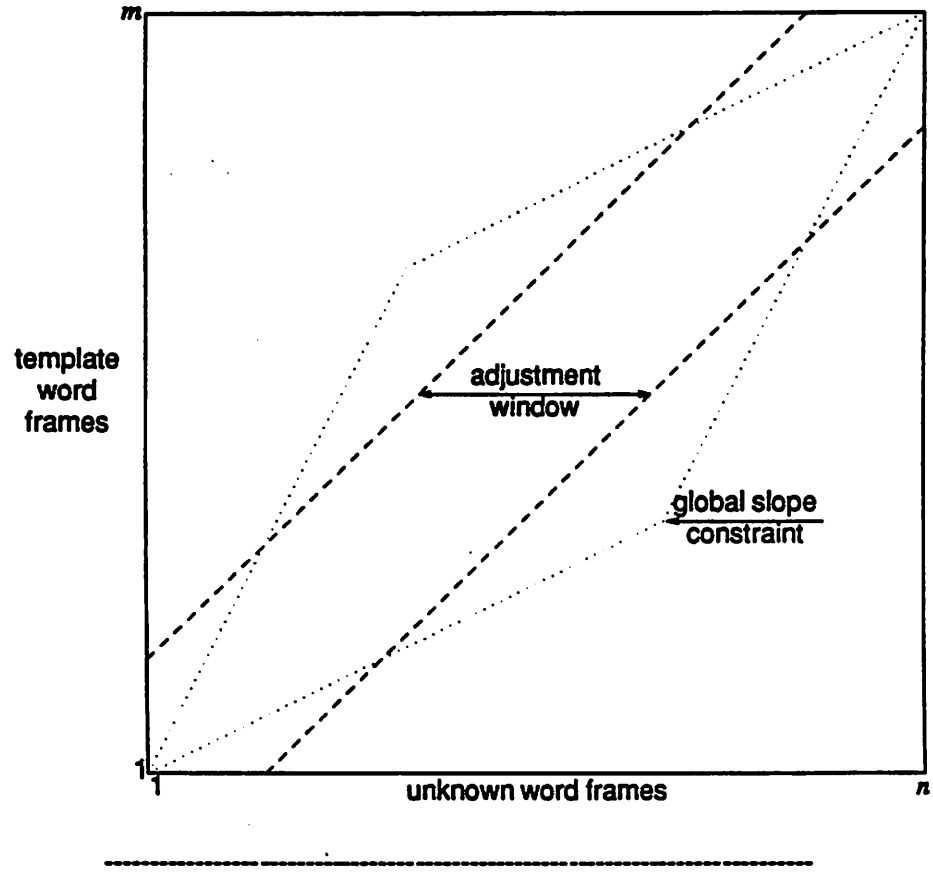


Figure 2.18 - Pruning techniques

distance metrics and minimization equations to be computed. This is acheived by computing the dynamic programming equations only for matrix elements near the diagonal (i.e. near a linear time warp). Values of the D matrix outside the window are set to infinity. Adjustment windows compute in a region parallel to the diagonal, while global slope constraints compute in a diamond shaped region around the diagonal. Both pruning techniques require the entire word be spoken before the pattern matching can start, thus forcing block computation and isolated word operation. Block algorithms then require faster processing to maintain reasonable latency (300 ms latency is used for the comparisons below), reinforcing the need for pruning. We have found that increased pruning reduces recognition accuracy, while complicating the hardware design.

The Bell DTWP chip is designed solely for isolated word recognition. It uses a block algorithm,

comparing a 40 frame off-chip unknown word with a 40 frame off-chip template in 902.5 μs. The comparison is performed in a 11 frame adjustment window along the diagonal of the word-to-word distance matrix. A total of 360 of the 1600 matrix elements are computed, increasing throughput by a factor of 4.4. The distance measure requires ten 250 ns clock cycles for a total of 2.5 μs per equation. A total of 330 isolated words can be recognized with 300 ms of latency. The chip is implemented with a special-purpose architecture including a 250 ns multiplier, a logarithm converter, a 14 word scratch-pad memory and a three input minimization circuit. The chip also performs address generation for the external template and unknown word memories.

2.6.2.4. Systolic Arrays

In addition to single chip solutions, systolic arrays have been developed to perform the dynamic-time-warp algorithm. These include designs from Lincoln Labs[31] and Bell Laboratories.[32] With systolic arrays, each processor (P.E.) performs the basic distance and minimization computation, but many processors are arranged to compute either an entire row, column or matrix simultaneously. The result is high throughput at the cost of many processors.

The Lincoln Labs approach is to meld systolic arrays with wafer-scale integration to produce a wafer that can recognize 3000 isolated words in real-time. While the circuit is still in development, it is interesting to note that the architecture of each individual processor is similar to that described here except parallel arithmetic was replaced with bit-serial arithmetic to reduce size at the cost of speed. Also, very large bit widths are used internally, thus each processor requires more area and runs slower. The wafer contains 63 processors and computes the time-warp algorithm diagonally through the matrix.

The Bell systolic array approach is to design a general purpose processor as the basic processing element. The result is that each distance metric/minimization requires 50 μs, which will be shown to be 62 times slower than the chip described here. The entire system requires 1600 chips to recognize 20,000 words. This is not a chip or board level, but a cabinet level solution. The Bell systolic array is also a block algorithm, although the array can be reconfigured to recognize 500 words as a streaming algorithm with 40 processors and additional scratch-pad memory.

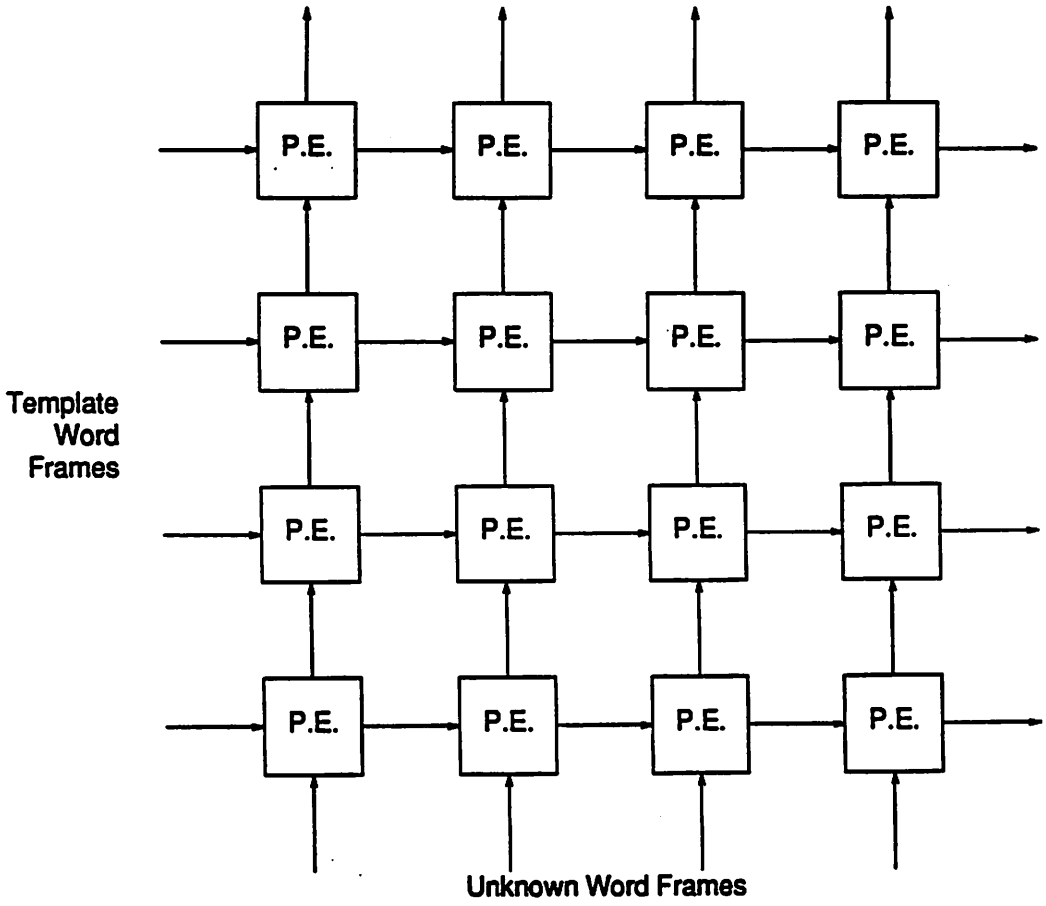


Figure 2.19 - Simple 4 x 4 systolic array

2.6.2.5. Our Approach

Systolic architectures applied to the dynamic-time-warp algorithm exploit parallelism by computing the time-warp matrix using many slow computational elements. This solution requires large interprocessor bandwidth, while simultaneously requiring processing elements which can perform the entire time-warp algorithm including both the distance and minimization equations. The result is that each processing element is large and slow, and many are required for high throughput. Our approach was to design a processing element that can compute the dynamic-time-warp algorithm as fast as possible limited by the number of I/O pins on the chip and the speed of the external template memory. Instead of using many general purpose arithmetic units, each arithmetic unit is tailored to a particular computation.

The bit widths of these units need not be large because the algorithm requires only four bits for each of the 16 features. Many of these units are then connected to compute the required equations in parallel. The result is a chip that performs a distance metric and minimization in 800ns, three times faster than the fastest of the previously described circuits while requiring the least area in low density fabrication technology (4μ NMOS).

The performance of the various chips is summarized below.

Table 2.2 - Time-Warp Chip Summary

	TMS320	NEC	DTWP	Bell systolic	Lincoln systolic	U.C.B.
distance/min time (μs)	25	2.25	2.5	50	10.6	0.8
isolated words	100	180	330	20,000	3000	1000
isolated processors	1	1	1	1600	63	1
connected words	35	20	none	20,000	1000	1000
connected processors	1	1	-	1600	65	1
streaming (real-time)	yes	no	no	no	no	yes
on-chip scratch-pad	no	yes	yes	yes	yes	no
cycle time(ns)	200	250	250	200	62.5	200
chip area (mm^2)	30	37	-	15†	12.5†	13
technology (μ)	3.5 NMOS	2.5 NMOS	2.5 CMOS	3.5 CMOS	5 CMOS	4 NMOS

† Area for each processing element in the array.

2.7. Conclusions

Dynamic-time-warp algorithms have been implemented in chips by many people interested in recognizing more words than are currently possible with general purpose processors. Speech recognition algorithms are normally refined to work well on general purpose machines without the influence of future special-purpose hardware implementation. With general purpose machines, chip implementation issues

such as bit widths and parallelism cannot be exploited, so they are ignored in favor of increasing algorithmic complexity (e.g. by pruning). Chip implementations based on these algorithms require much silicon area and their performance limits their use to moderate vocabulary sizes of a few hundred words.

Special purpose chips can perform operations much more quickly than general purpose chips if the data path reflects the algorithm. By developing the speech recognition algorithm with the hardware implementation in mind, the resulting algorithm can be implemented efficiently in special purpose integrated circuits requiring small silicon area. The development of a special purpose integrated circuit that can recognize 1000 words in real-time, isolated or connected, with a 4μ technology demonstrates the importance of chosing the correct architecture for hardware implementation.

Chapter 3 - The User Interface

3.1. Introduction

One of the major aspects of this project is to identify and study the problems associated with using a speech recognizer as a user-computer interface. Speech recognition systems are normally thought of as a front-end to some underlying application such as an airline reservation system. The design of the application is closely coupled to the design of the speech recognition system. For an engineering workstation environment this approach is too limiting. This chapter treats the speech recognizer as an input device and examines why speech input devices differ from standard input devices. The result of the analysis is a set of feedback techniques which can be used to support the user's interaction with the computer and help increase user productivity.

3.2. Hardware

Computers have many input and output devices, but most of these are not user oriented. Disks, tapes, network links, and modems all either produce or read information that a user normally cannot. Input devices such as card readers and printers deal with user-readable information, but interaction with the computer is not immediate. Most modern computer systems support "on-line" input devices such as terminals. When using a terminal, interaction with the computer is immediate, that is, the computer responds to each character or line as the user is typing. This is the type of system which is best supported by speech recognition. For this reason we will examine speech only in interactive environments.

3.2.1. Output Device

By far the most common user-readable output device for workstations is the screen (e.g. CRT, liquid crystal, plasma). The resolution of screens varies from standard text-only 80 character by 24 line terminal screen to 1024 by 1024 (or higher) pixel text and graphics high-resolution bit-mapped displays. High-resolution bit-mapped displays are used in the current generation of engineering workstations. Bit-mapped displays allow text and graphic symbols to be displayed anywhere on the screen. This output

device also supports windowing systems. A window is a part of the screen that is dedicated to a single application where all interaction between that application and the user takes place through the window. Because the screen has high resolution, many windows can be placed on the screen at one time allowing multiple applications to run at one time.

3.2.2. Input Devices

Engineering workstations generally have two input devices: a mouse and a keyboard. A mouse is a hand-held device which rests on a smooth surface. A cursor on the computer's screen mimics the mouse's movements allowing the user to "point" to positions and objects on the screen. The cursor is a feedback mechanism which allows the user to know where the mouse is really pointing. In addition to pointing, a mouse normally has a few buttons which can be used to select, delete, create or otherwise operate on objects such as windows, text, and menus.

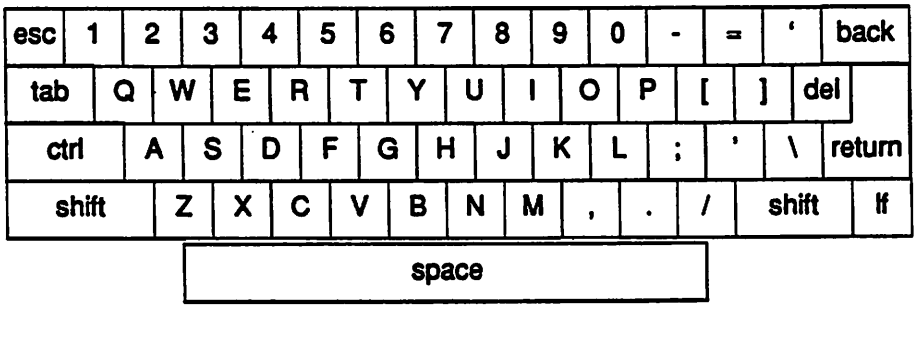


Figure 3.1 - Standard English keyboard configuration

The keyboard has been the predominate input device for computers since the first computers were developed in the late 1930's and early 1940's.[33] A keyboard is a board, about 6 inches by 20 inches (sizes vary) that contains keys (switches) configured as a near-matrix. Each key is associated with a symbol except for a few special keys (shift/control/meta) which modify the meanings of other keys. The keys are arranged in a standard configuration and labeled allowing users to use many different computers without having to relearn key positions. For English keyboards, each letter and number of the alphabet has its own key, and special symbols have their own keys, or share keys by use of the shift key. In

addition to the standard English symbols, keyboards support a large number of control and meta symbols. Typical keyboard alphabets consist of 128 to 256 different symbols. In addition to these symbol keys, many keyboards contain "function" keys. Function keys, normally labeled with special numbers, are not associated with symbols but with commands (functions). There are two types of function keys. The most common send an "escape sequence" to the computer when pressed. An escape sequence is a predefined sequence of characters that start with an "esc" character. The other type of function key is programmable. When pressed it sends a user-definable sequence of keys. Programmable function keys are not as common as escaped sequence keys, and are not supported very well in most operating systems and applications.

A keyboard provides feedback in a few different forms. First, some keyboards "click", making a small noise for each character sent to the computer. Second, most keyboards provide tactile feedback by pressing back against the users fingers when the key is pressed. Third, most keystrokes perform some action on the screen almost instantly after the key is pressed. In most computer systems this instantaneous response, called echoing, is supported at the lowest possible level. Once the user is familiar with the placement of keys on a keyboard, data can be entered very quickly - a rate of more then 360 characters per minute is typical of fast typists.

Both mouse and keyboard input devices provide quick feedback to the user so that errors can be detected early. This feature is essential because user interaction with the computer is not perfect. Sources of error vary greatly, but with keyboards most errors are due to typing mistakes. Other sources of errors are "bouncy keys", where the user presses a key once but the computer thinks the key was pressed many times, and "dirty keys", where a key is pressed but the computer does not detect it. It is the feedback mechanisms described above that inform the user when an error has occurred.

3.3. Speech Input

The user interface requirements of a speech recognition system are very different than those of a keyboard or mouse. These differences stem from both the nature of speech itself and the constraints of currently available recognition algorithms. To create an effective user interface the idiosyncrasies of

speech recognition must be understood and handled correctly at the lowest levels of the interface.

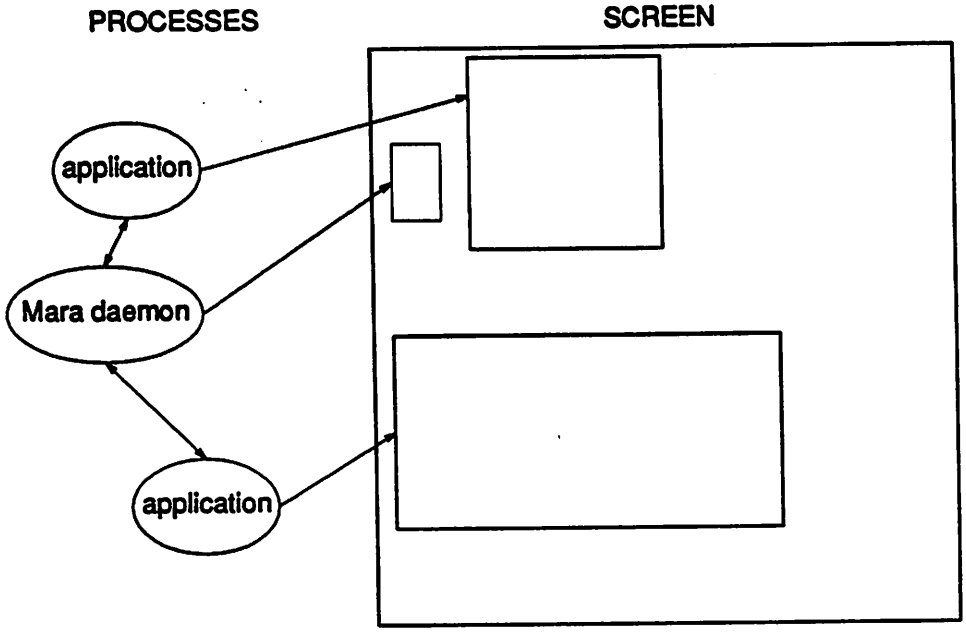


Figure 3.2 - Structure of Mara System

Part of the design of a good speech recognition system is integrating it into the computer system. The speech recognizer should not consume much of the resources of its host computer, but some resources are needed to create an effective user interface. One way to share the recognizer among many different applications and provide a uniform interface to both the application and the user is to implement the speech recognizer as a daemon (server) process. A daemon is a program that handles all interaction between an application (called a client) and a shared resource (called a service). The Mara daemon was developed for this purpose, providing 5 basic functions:

- provide the user with feedback related to speech input,
- train and update templates for words,
- maintain all recognizer related parameters,
- coordinate requests for speech input from many applications in a manner independent of the underlying speech recognition algorithm where possible, and

- allow for the testing, debugging, and evaluation of the speech recognition hardware, algorithms, and the complete system.

The daemon is started by the user at log-in time and claims a small part of the screen (a window) for feedback to the user. Because the user must log into the system, the speech recognizer knows who is talking, and can therefore use the database of pronunciations specific to that user. The rest of this chapter describes the problems associated with speech input devices and how the Mara daemon solves these problems.

3.4. Word Model

A template-based speech recognition system recognizes words based on templates created from previous utterances of a word. Templates must be stored in permanent storage (on disk) to eliminate constant retraining. Storing templates on the host computer's allows them to be shared among many workstations and allows the host computer to create and manipulate templates. In fact, in the current daemon implementation the training algorithms run on the host computer.

The connection between a word and its templates forms the "word model" used by the recognizer. Template-based recognizers can recognize any sound as a word allowing users to speak with their normal pronunciations. But most recognizers, including the one described here, cannot verify that a word has any particular pronunciation. The connection between a word and its pronunciations (templates) must be enforced by the user, an interface problem specific to speech input devices. To show why this is important, consider what would happen if the word "stretch" was trained as "expand". Now if the user wishes to perform the function associated with "stretch" and forgets that "stretch" was trained as "expand", there is no way to tell the user to say "expand" because the templates cannot be played back to user. Thus the user must retrain the word "stretch". Another way to think about this problem is to consider each word as an alias of a function, but one cannot list the aliases. Thus, if a user forgets the alias, it must be destroyed.

One solution to this problem is to develop a speech recognizer that can speak templates back to the user. Our system did not have this capability, so in order to solve this problem the user must be able to

speak a word unambiguously from its spelling. A set of disambiguating conventions was developed to represent words. First, a dictionary of homonyms was used to convert each spelling into a canonical spelling. For example, "2", "to", "two", and "too" all have the same canonical spelling "2". Multiple word phrases are represented as "word_word" and single letters are represented as single letters. Thus the UNIX word "vi" is spelled "v_i". These conventions are not strictly enforced, but are used by all programs written as part of this project. By representing words with an unambiguous spelling, the feedback techniques described later will have real meaning to the user.

3.5. A Virtual Recognizer

When an application program wishes to use speech input it must connect to the Mara daemon. The daemon then provides a "virtual recognizer" to the application. The virtual recognizer is a set of server calls (one can think of them as system calls) that application programs require to use a recognizer. A list of virtual recognizer calls can be found in table 3.1. Each virtual recognizer is independent, but because of the word model above, applications can share words efficiently.

When an application is started it normally loads its vocabulary into the recognizer using the "LoadaSpelling" call. This call returns an integer between 1 and 1000 (called a "uname") that is the internal name assigned by the recognizer for a word. The uname for a given word is the same for all applications, reducing the memory required in the daemon. After words are trained the application indicates where on the screen the application is "active". When a word is spoken the recognizer determines which application receives that word by looking at which window the mouse is over. An application can associate itself with any number of windows by using the "AssocWindow" call. An application can also force the daemon to accept all voice input, independent of the mouse. Finally, the application turns its virtual recognizer on. While the recognizer is on, when a word is spoken in one the application's windows, a message is sent to the application through the recognizer's file descriptor (i.e. like any other input device). The application can then read this input using the "GetaHearing" procedure to determine the action to be taken. The message sent by the daemon contains the uname of the recognized word or a special "reject" uname. If the application wishes to train a new word it must first turn off the recognizer,

Table 3.1 Mara Virtual Recognizer Commands

Type	Meaning	Function
WORD * HEARING *		GetaWord(char *prompt; WORD *word) CompareWord(WORD *w; HEARING *hearing)
int int int int	uname uname uname uname	LoadaRawTp(WORD *word; int flags) UnloadaUname(int uname) LoadaSpelling(char *spelling; int flags) UnloadaSpelling(char *spelling)
int int int int int	param param param param param	MaraParameter(int parameter; int valuetype; int value) SetMaraParameter(parameter; value) GetMaraParameter(parameter) SetDefaultMaraParameter(parameter) GetDefaultMaraParameter(parameter)
int int int int int	flags flags flags flags flags	MaraFlags(int flags; int valuetype; int value) SetMaraFlags(flags, value) GetMaraFlags(flags) SetDefaultMaraFlags(flags) GetDefaultMaraFlags(flags)
int int int int	T/F T/F T/F T/F	ConnectRecognizer(int fatal) OnRecognizer(int graball) OffRecognizer() DisconnectRecognizer() AbortRecognizer()
int int	uname uname	AddMemberToClass(char *class, *member, *cdata) DeleteMemberFromClass(char *class, *member)
int	T/F	AssocWindow(int window)
		FlashString(char *string; int x, y) FlashUname(int uname; int x, y) TpVerify()

then train the word then turn the recognizer back on.

The virtual recognizer also supports debugging and parameter-setting system calls. The "Loada-Word" call allows the application to skip the word model and directly load an arbitrary template into the recognizer allowing low-level tests of the recognizer. Recognition related parameters such as rejection threshold, peak in-word energy threshold, silence between words/sentences etc. can also be set through virtual recognizer calls. Parameters are global and therefore effect all virtual recognizers.

In addition to returning just the best matched word, the recognizer can be placed in "evaluation mode" causing the top few matches to be returned to the application. This mode is normally used to evaluate the recognizer but can also be used to implement a maximum likelihood parser in applications

```
/*
 *    Load the digits, then recognize from the microphone
 */

#include <stdio.h>
#include <mara/mara.h>

#define    TRUE          1
#define    FALSE         0

char *digits[] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "oh", NULL};
char *names[1000];

main()
{
      char **p;
      int  window;
      HEARING *hr;

      ConnectRecognizer(TRUE);
      for(p=digits; *p; p++) {
            uname = LoadaSpelling(*p, 0);
            if(uname < 0) {
                  fprintf(stderr, "Couldn't load %s\n", *p);
            } else {
                  names[uname] = *p;
            }
      }
      sscanf(getenv("WINDOW_ME"), "/dev/win%d", &window);
      AssocWindow(window);
      OnRecognizer(FALSE);
      while(hr = GetaHearing()) {
            printf("you said %s with score %d.\n",
                  names[hr->hr_data[0].hr_uname], hr->hr_data[0].hr_score);
            FreeHearing(hr);
      }
}
```

Figure 3.3 - Sample application: recognize from microphone

that require constraining grammars. The number of matches returned is by the application.

The "Compare" call allows the recognizer to be used as an array processor. That is, the recognizer normally will compare incoming words from a microphone with its templates. The "Compare" call will send a word to the recognizer as though it was spoken, and return the best match. When used in conjunction with evaluation mode, this call allows large databases of words to be compared very quickly (60

```
/*
 *      Load the digits, then compare each disk file specified on command
 *          line with all digits
 */

#include <stdio.h>
#include <mara/mara.h>

#define     TRUE            1
#define     FALSE           0

char *digits[] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "oh", NULL};
char *names[1000];

main(argc, argv)
        int  argc;
        char *argv[];
{
        char **p;
        int  i;
        HEARING *hr;

        ConnectRecognizer(TRUE);
        for(p=digits; *p; p++) {
                uname = LoadaSpelling(*p, 0);
                if(uname < 0) {
                        fprintf(stderr, "Couldn't load %s\n", *p);
                } else {
                        names[uname] = *p;
                }
        }
        for(i=1; i<argc; i++) {
                word = ReadaWord(argv[i], NULL);
                hr = CompareWord(word, NULL);
                printf("file %s sounds like %s with score %d.\n",
                        argv[i],
                        names[hr->hr_data[0].hr_uname],
                        hr->hr_data[0].hr_score);
                FreeHearing(hr);
                free(word);
        }
}
```

Figure 3.4 - Sample application: recognize from disk

words/minute). The word used by the "Compare" call is not a digitized word; it is the output of the recognizer's front-end processor.

3.6. Adding New Words

Most speech recognition systems separate recognizer use into two phases, training and recognition. While this view holds well for studying speech recognizers, it is not the way that recognizers are used. When used, training and recognition often occur at the same time, that is, new words are added while an application is running. Also, from the point of view of the application program, training is a recognizer-dependent feature with which applications should not have to deal, because in the future some recognizers may need no training. An application program really requires only one word-related call such as "LoadaSpelling".

Representing a word by its spelling allows many applications to share the same words without having to retrain words for each application. Since templates are associated with a word, not an application, each application need only send the daemon the spelling of a word to be used. The Mara daemon supports the function "LoadaSpelling" that can be executed by any application program when its virtual recognizer is off. "LoadaSpelling" will train a word if necessary (i.e. if no templates are found on disk), then load that word's templates into the recognizer.

Training requires user interaction, and if the user makes a mistake while training the recognizer, then the recognizer cannot operate properly. In order to aid the user, the training interface should be consistent and error resistant. The training algorithm must balance three considerations. First, the user wants to say each word to be trained as few times as possible. The reason for this is obvious: training is a burden on the user and it does not directly aid an application. On the other hand, a word must be trained well in order for the recognizer to work well. It has been discovered by many researchers that averaging many utterances of a word to form a template increases recognition accuracy greatly, and the more utterances that are averaged, the better the recognition accuracy. The third consideration is that users do not always say what they are prompted to say. This is especially true when the recognizer is being used with an application that requires new words while running.

The Mara daemon trains words by prompting the user to say each word twice. The prompt appears in the middle of the screen in a large white box that pops up only when words are to be trained. These

two utterances are compared and if they are similar (i.e. their word to word scores are small) then the daemon averages the two utterances to form a single template. If the utterances are not similar then the user is prompted to say the word again. The training algorithm then compares all possible pairs of two of these three utterances to find a match. If no match occurs the user is prompted to say the word again. This process repeats until the word is spoken twice similarly. The Mara daemon's training algorithm attempts to balance all three considerations. Thus when the user cooperates completely, he must say each word only twice. If the user makes a mistake or the recognizer makes a mistake then more than two utterances are required.

It is interesting to note that keyboards and mice do not require their applications to load a vocabulary into them (an exception is keyboards with programmable function keys). Instead, these devices have a predetermined set of symbols that they can send to the host. This is one reason why a speech recognizer cannot be thought of as just a "voice keyboard".

3.7. Feedback to the User

Another major difference between speech recognizers and keyboards is that speech recognizers make many more mistakes, and the user does not expect (nor can he predict) when errors are going to occur. Keyboards and mice generally do not make mistakes, BUT, users make lots of mistakes when typing. Thus some of the feedback mechanisms described here apply equally to keyboards and speech recognizers, others are specific to speech. A speech recognizer can make three different types of mistakes:

- Rejection error - the user speaks a word/phrase and the computer does nothing.
- Insertion error - the user says nothing but the computer does something.
- Substitution error - the user says one thing but the computer does something else.

3.7.1. Rejection Errors

Rejection errors are by far the most common sort of errors encountered in normal use. These errors come from many different sources. First, the recognizer might be turned off, broken or not initial-

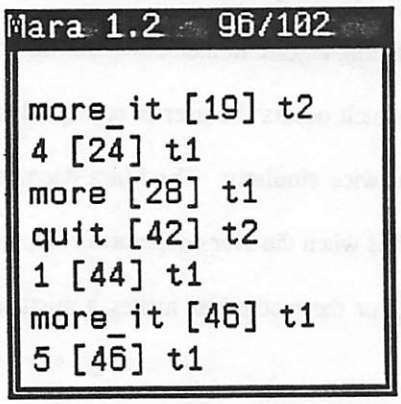


Figure 3.5 - The Mara window

ized properly. To solve this problem the user must receive feedback from the computer indicating that the hardware and software are set up correctly. The Mara system provides such feedback in the form of a VU meter on the screen. The VU meter is updated after each word is detected, causing a gray bar proportional to the log of the peak in-word energy to be displayed in a fixed spot on the screen. Additional VU meter information shows a long term energy estimate and background noise energy estimate so that the pre-amplifier gain can be set correctly. System parameters such as the rejection threshold must also be set properly.

The second source of rejection errors occurs when the user speaks a word that is unknown to the recognizer. While this error is really a user error, it is a very common phenomenon and must be handled by the user interface. The Mara daemon presents an ordered list of the top few recognition possibilities for each spoken word along with the recognition score. This list is called the top words list. Normally if a word is in the recognizer's vocabulary then the word will be in the list. Also, if the spoken word sounds similar to the required word, the user is reminded of the correct word. This feature is especially useful for multi-word commands when the user remembers one word but not the other. The user can also scan the current recognizer vocabulary by typing into the Mara window.

The third source of rejection errors occurs when the user says the correct word but the score of that

word is greater than the rejection threshold. This is the type of rejection error traditionally associated with speech recognizers and can be overcome by simply saying the word again. Another solution is to retrain the word so that its templates yield smaller scores.

3.7.2. Substitution Errors

From the user interface point of view, substitution errors are already dealt with in many systems. A substitution error can be handled just as if the user made a mistake and said the wrong word. An "undo" command will undo the error, and the word can be spoken again as though it was rejected initially. In order to supply the user with a consistent method of undoing errors (since substitution errors can happen at any time), the applications program should have an "undo last word" function and trigger that function when a special word such as "error" or "undo" is spoken. Note that for many programs such "undo" commands already exist, although they are usually implemented by undoing a single keystroke. With a speech interface, errors occur at the word level, so it is important to undo the last word. Often the connection between number of keystrokes and words is difficult and inconsistent. How this problem is handled in the Mara system is discussed in chapter 4.

3.7.3. Insertion Errors

Insertion errors are specific to speech recognition systems. It is very rare to find a keyboard or mouse that sends characters to the host computer without any user actions. A speech recognizer will listen to noise that comes into its microphone and, if it is loud and long enough, interpret it as a word. Thus non-speech sounds such as door slams, breathing noise or telephone ringing. Unintentional speech sounds such as expletives, mumbling, or someone talking in the background may match closely to some word in the vocabulary. If this match has a score less than the rejection threshold then the system will perform some random operation. If the application does not give feedback to the user about each command performed, then the user would have no way of determining what function was accidentally performed without looking at the top words list. Even the top words list does not work well for most noise sources because it is updated with each new spoken word, and if the noise source is interpreted as more

than one word, then the window changes too quickly.

There are two solutions to the insertion error problem. First, the application must inform the user, somehow, about each word that is acted upon. Thus interfaces that allow users to change things that are not visible should be rewritten to perform some function on the screen (for example flash a word on the screen in such a way as not to slow down the interface). The second solution is to turn off the recognizer temporarily when noise is a problem, for example, when someone comes up to talk to the user. The Mara daemon allows the user to turn it off by merely pointing the mouse to a window on the screen that has no application running in it, for example, the background window. Unlike rejection and substitution errors, insertion errors cannot be reduced by retraining unless a template is trained to some specific noise by accident.

3.7.4. Other Feedback

The ordered list of recognition possibilities is a very useful feedback mechanism. It provides complete information about what the recognizer is hearing and how well the reconizer is working. Scores in the Mara window can be examined to determine how to set the rejection threshold, when to retrain a word, and if the recognizer is working well or just marginally so (i.e. if scores are close to or far from the rejection threshold). Another feedback mechanism was also considered, flashing the recognized word on the screen over the cursor, that is, where the mouse points. "Reject" was flashed when a rejection occurred. This flashing slowed down the response time of the system so that the user could not do anything while the word was flashed, and was therefore discarded.

3.8. Debugging

The feedback mechanisms described above provide the user with limited information as to why the recognizer make mistakes. Sometimes the user might like more detailed information about a particular mistake. This is especially true when the user is the trying to debug the system. When requested, the Mara daemon will pop up a window that shows how the dynamic-time-warp algorithm comparing the incoming unknown word with any specified template.

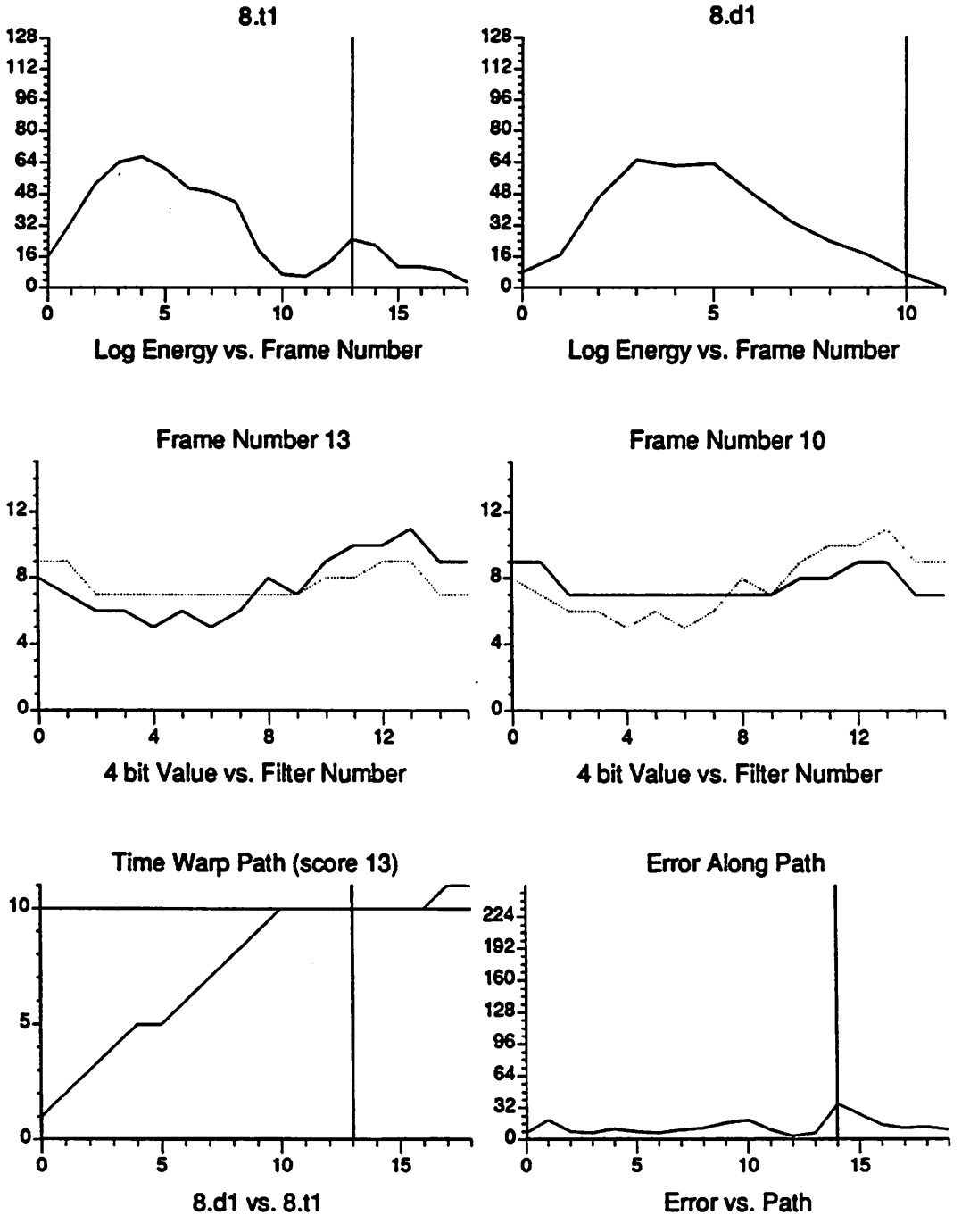


Figure 3.6 - Dynamic-time-warp comparison of template and utterance of 'eight'

The window contains six plots shown in figure 3.6. The plot upper left in the upper left corner shows the energy vs. frame number for the template and the plot to its right shows the same for the unknown word. The two plots in the middle show one frame of the template overlayed with one frame of the unknown and vice versa. The frame numbers of the template and unknown word are selected by moving a highlight in the upper plots or the lower plots. The plot in the lower left corner shows the dynamic-time-warp path for the word to word comparison. The plot next to it shows the spectral error along that path.

With a little practice, these plots can provide much information about why a recognition error occurs. If the peak spectral error occurs at one end of spectral error plot then the error is most likely an endpoint error (such as the one shown in figure 3.6). This can be confirmed by looking at the shape of dynamic-time-warp path near the lower left and upper right corners. If the path is either vertical or horizontal this confirms the error. Other errors can be found by comparing the energy versus time plots looking for missing peaks indicating different stress or missing syllables. Individual spectra can be examined to determine if the front-end spectral analysis is working properly. For example, spectral tilt, missing or shifted filters and flat specta all indicate that the front-end processing is faulty.

3.9. Feedback from the User

The feedback mechanisms describe above solve the problem of telling the user what the recognizer is doing, but do not allow the user to improve recognition accuracy. In order to do this, the user must give feedback to the recognizer informing it when an error has occurred and how to fix the error. The Mara system allows two forms of user feedback: retraining and adaptive training.

3.9.1. Retraining

The idea behind retraining is simple, if a template is "bad", remove it and train a new one. The Mara daemon allows any template to be retrained by pointing to it in the Mara daemon window, then selecting the retrain command from a pop-up menu. The retraining algorithm is the same as the initial training algorithm: the user is prompted to say the word twice similarly then a template is created by

averaging those two utterances.

3.9.2. Adaptive Training

Retraining can solve the problem of completely mistrained templates (e.g. templates that are trained to a telephone ring) but does not solve the most common training problem: the way a person says a word when reading is NOT the same as when speaking spontaneously. One must remember that the recognizer is trained by prompting the user to read a word then speak it. When in use, however, the recognizer hears words spoken spontaneously. For example, the word "erase", when read, is often pronounced with a initial /i/ sound; while when spoken spontaneously, the /i/ sound can change to an /uh/ sound. Another example is that the word "eight" is often missing the trailing /t/ burst when spoken spontaneously but not usually when read.

Another difference between utterances used to form templates and spontaneous speech is due to physical and mental changes in the user. For example, a cold causes severe nasalization of sounds that are normally never nasalized. This causes a decrease in recognition accuracy.

Training is the only knowledge that the system has as to how words are pronounced. In actual use, input to the recognizer is spontaneous, so templates should be created from the recognizer while it is being used. Unfortunately, to use the recognizer it first must be trained. To overcome this problem the Mara daemon uses the initial training algorithm described above to form basic templates, then an adaptive training algorithm attempts for refine these templates. The result is an increase in recognition accuracy without requiring the user to perform tedious, unneeded, and possibly useless training.

The goals of the adaptive training algorithm are to:

- use spontaneous utterances of a word to update existing templates,
- integrate into the system without consuming too many system resources (especially disk space),
- be envoked either automatically or semi-automatically to ease the burden on the user,
- adapt to short and long term voice changes by the user to eliminate periodic retraining, and
- increase recognition accuracy.

3.9.2.1. Design Issues

The first issue in the design of an adaptive algorithm is to identify the input to the algorithm. There are two possibilities: one is to use just the unknown utterance as input, the other is to use both the unknown utterance and a label indicating the word corresponding to the utterance. The first input scheme allows fully automatic adaptation (i.e. no user input is required), while the second scheme requires the user to tell algorithm which word was spoken, and therefore is only semi-automatic. The algorithm implemented in the Mara daemon is semi-automatic. The user can adapt any word by first saying that word spontaneously, then finding the word in the top words list. Very rarely, if it is not in the list, then the word can be typed into the daemon's window. To perform the adaptation all the user must do is press the left mouse button. Adaptation is only performed when the user wishes, reducing the computational overhead incurred by adaptive algorithms.

The second issue is how many templates will be used to represent each word. In general, if one uses more templates per word, the recognition accuracy increases, but the fewer words can be recognized at one time. There are two different minimization criteria that can applied to trade off templates per word versus vocabulary size: for each word, minimize the number of templates for that word; or for all words, minimize the average number of templates per word. The second minimization is very difficult because the vocabulary changes often, therefore the Mara daemon attempts to minimize the number of templates per word.

The third issue is how many extra pieces of information are required to perform the adaptation. This information must be stored on the host computer's disk using some of the host computer's resources. Thus, the amount of extra information should be as small as possible.

3.9.2.2. Previous Works

Few adaptive training algorithms have been proposed in the literature. Lowerre[34] implemented a semi-automatic adaptive algorithm for HARPY.[35] HARPY is not a template-based speech recognizer, but can be thought of as one, if templates are considered to be entire phrases. Fully automatic schemes have been proposed to train sub-word level spectral classifiers[36] and feature classifiers,[37] but these techniques

do not apply directly to word or phrase level adaptation. All of these adaptation algorithms are used to convert speaker-dependent systems into speaker-independent systems by updating speaker dependent information in the system for each new user. A dramatic increase in recognition accuracy occurs when adaptation is performed. Lowerre[34] found that an errors decrease from 7% to 2% when adaptation was performed, and Stern[37] found that errors decrease from 12.5% to 6.2% in their system.

To date, adaptive algorithms update only spectral information in a word but do not change the structure of a word. These schemes, therefore, cannot handle cases where a user speaks a word with an unexpected pronunciation. Lowerre solved this problem by assuming that all expected pronunciations could be computed by applying linguistic principles. Since no linguistic principles are built into the Mara system, it must provide an empirical method of finding unexpected pronunciations.

3.9.2.3. The Algorithm

The adaptive training algorithm has three inputs and two outputs. The first input is the unknown utterance u, the second is the set of templates (T) for that utterance, the third is a set of template considerations (C). Template considerations are averages of utterances that might be templates in the future. Note that because the unknown word is labeled by the user, its templates are known. The algorithm outputs a set of templates and a set of template considerations. The templates are loaded back into the recognizer and the template considerations are stored back on disk.

Each template and template consideration corresponds to a possible pronunciation of the word. The first step in the algorithm is to determine if the unknown is a new pronunciation of the word or just a refinement of an existing pronunciation. This is done by measuring the word-to-word distance between the unknown utterance and each template and template consideration. If a distance is less than the adaptation threshold T_{adapt} then the unknown is averaged into that template. In computing the average, the unknown word is weighted by 3 while the template is weighted by 2 times the number of utterances (N) that were averaged to create that template. The maximum template weighting is 8. The weighting can be thought of in terms of the formula:

Definitions:

u - unknown word
T - set of templates for u
C - set of template considerations for u
L - temporary set of templates and template considerations

Algorithm:

```
any ← FALSE
for all x in T ∪ C
    if distance(x, u) < T_adapt then
        any ← TRUE
        x ← weighted_average(x, u)

if any then
    L ← T ∪ C
  else
    L ← T ∪ C + u

sort L in ascending order

T1 ← L1
if weight(T2) > T_min-weight then
    T2 ← L2

C ← L − T
```

Outputs:
T and C

Figure 3.7 - The adaptive training algorithm

$$T_{new} = \frac{3 \cdot U + 2 \cdot N \cdot T_{old}}{3 + 2 \cdot N}$$

If the unknown word is not within T_{adapt} of any template or template consideration, then it is assumed to be a new pronunciation and entered into the set of considerations. The sets of templates and template considerations are then merged into one set (L).

The last step in the algorithm is to split L into the new sets of templates and template considerations. First L is ordered by weight such that L_1 has the highest weight. L_1 is always put into the set of new templates, and L_2 is put into the set of templates if its weight is greater that the threshold $T_{min-weight}$.

The remaining mumbers of L are placed in the new list of template considerations.

3.9.2.4. Thresholds and Weightings

The two thresholds T_{adapt} and $T_{min-weight}$ (along with the weighting scheme described above) determine how the algorithm responds to short term and long term changes in pronunciations. T_{adapt} can be thought of as the largest possible error between two different pronunciations of a word. Small values of T_{adapt} will force many template considerations with little averaging. Large values will perform much averaging among different pronunciations. Because averaging utterances is known to cause an increase in recognition accuracy, T_{adapt} should be set high. The Mara daemon sets this threshold 20 percent above the rejection threshold.

The $T_{min-weight}$ threshold determines how quickly the system will adapt to new pronunciations. If the threshold is set to 2 then one spontaneous utterance of a word which corresponds to a different pronunciation will cause the algorithm to create two templates. If the threshold is set to 3 or more then two or more utterances would be required to force the creation of a new template. The Mara daemon sets the threshold to 2 for two reasons. First, because the user normally requests adaptation only when the system is not working well, the recognizer should respond as quickly as possible. Requiring two utterances of any new pronunciation would seem like an unresponsive system from the user's point of view. Second, as stated previously, spontaneous utterances often have a completely different pronunciation than read utterances so it is not unreasonable to expect that a single labeled spontaneous utterance corresponds to a new pronunciation.

The weightings were chosen so that the template is a running average of the all utterances that created it. To give more preference to recent utterances the template weight is limited to 8 and the weight of the newest utterance is 3 instead of 2.

3.9.2.5. Experiments

A set of experiments were performed to test the adaptive training algorithm. The database for the experiments was collected from a high fidelity audio recording of a live KIC[38] editing session. Details

about the data collection can be found in section 5.2.1. Two speakers were used for the experiments: steve and peter. The vocabulary consisted of the 100 words shown in table 3.2. The database of spoken words consisted of 2 or 3 prompted utterances of each word (used for training) followed by many spontaneous utterances (1658 for speaker steve, 952 for speaker peter). The distribution of the spontaneous utterances for a single speaker is shown in figure 3.8. Note that many words have a few or no spontaneous utterances while a few words have many utterances. This is typical of recognizer usage in the engineering workstation environment.

Table 3.2 KIC Vocabulary

add_layer	arc	area	attribute	basic
black	blink	blue	bottom	box
boxes	brown	change_layer	chip	colors
contact	copy	delete	deselect	diffusion
dimension	directory	donut	edit	erase
expand	fill	flash	glass	green
grid	hilight	instance	kill	label
last	layer	left	less_blue	less_green
less_red	lyra	menu	metal	mirror_x
mirror_y	more_blue	more_green	more_red	move
no	opamp	outline	p_plus	pan
peek	poly	poly_two	polygon	pop_cell
push	0	1	2	3
4	5	6	7	8
9	90	180	270	45
red	redraw	remove_layer	return	save
select	stretch_box	switch	technology	text
text_two	top	undo	update	visible
well	width	window	wires	write
write_out	yellow	yes	zoom	zoom_full

The first experiment simulated the adaptive training algorithm when used by a conscientious user: one who adapts each unknown word only when a rejection error or substitution error occurs. The test was kept "fair" by allowing only one template per word. Each adaptation averaged the unknown word into the current template creating a new template (i.e. $T_{adapt} = \infty$). The experiment was performed for rejection thresholds between 50 and 85 in steps of 5. The results of the first experiment are shown in figure 3.9, and the same data is presented in table 3.3.

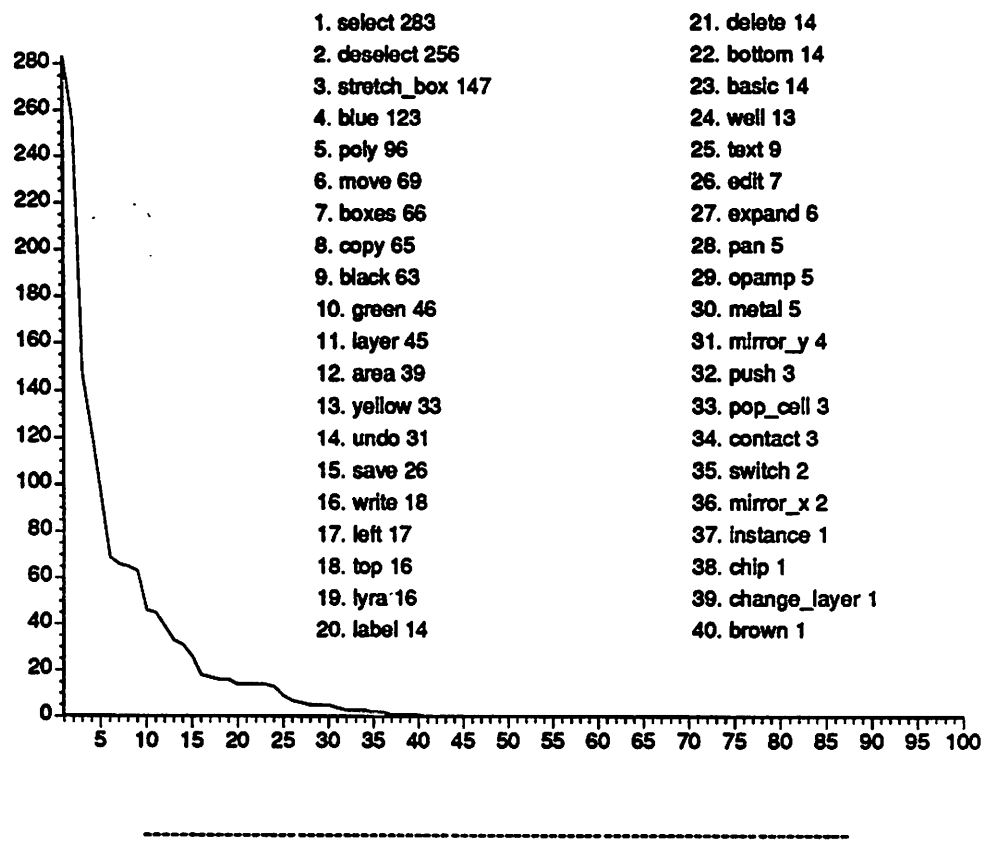


Figure 3.8 - Distribution of spontaneous utterances for speaker steve

In figure 3.9, the right hand side of the horizontal axis corresponds to no adaptation at all. Comparing plots (a) and (b) of the figure, note that as the number of adaptations increases the error rate falls quickly and then plateaus. This suggests that the user need only adapt a word once for reasonable recognition accuracy. In order to measure the performance of the algorithm when the error rate plateaus, the difference between the word to word scores of the unknown word compared to the correct template and the best match among the incorrect templates was computed. The average value of this difference is called the separation and is plotted in Figure 3.9c. It will be shown later that the separation tracks the number of adaptations nicely, but it does not always track the error rate accurately.

The second experiment shows the ability of the adaptive training algorithm to find new pronunciations of a word. For this experiment the rejection threshold was fixed at 50, and T_{adapt} was varied from 60 to 100 in steps of 5. Figure 3.10 (table 3.4 shows the same data) shows the number of words with 2

Table 3.3 Adaptive Training Experiment One: $T_{adapt} = \infty$

speaker	rejection threshold	substitution errors	number of adaptations	separation
peter	no adaptation	33 (3.5%)	0	36.4
	∞	11 (1.2%)	11	39.2
	85	11 (1.2%)	16	40.0
	80	10 (1.1%)	19	41.6
	75	11 (1.2%)	24	42.3
	70	9 (0.95%)	30	43.8
	65	8 (0.84%)	36	44.7
	60	6 (0.63%)	47	45.6
	55	4 (0.42%)	64	46.6
	50	5 (0.53%)	96	48.3
steve	no adaptation	35 (2.1%)	0	31.8
	∞	16 (0.96%)	16	32.9
	85	14 (0.84%)	15	33.0
	80	13 (0.78%)	16	33.0
	75	13 (0.78%)	22	34.5
	70	12 (0.72%)	29	35.9
	65	11 (0.66%)	38	36.8
	60	12 (0.72%)	50	37.4
	55	10 (0.60%)	71	38.3
	50	11 (0.66%)	97	40.0

templates versus T_{adapt}. Note that as T_{adapt} decreases, the number of words with 2 templates remains relatively flat then suddenly increases. The value of T_{adapt} should be chosen at the bend of the plot (about 75).

Allowing multiple pronuncations did not change the recognition accuracy and separation measurably so a different criterion was used to evaluate effectiveness. Words with two templates were compared using the debugging tool described above. Differences between the two templates were classified as shown in table 3.5. The words are listed in the order in which algorithm finds their second pronunciation as T_{adapt} is decreased. Some words generated two templates that were very close to each other, but most words have two distinct templates. This experiment shows that the scores generated from the dynamic-time-warp algorithm can be used to separate pronunciations of a word, and that most words require only one template, but a few words require two templates.

The third experiment examined the relationship between training on prompted versus spontaneous speech. For this experiment, the first two spontaneous utterances of each word were removed from the

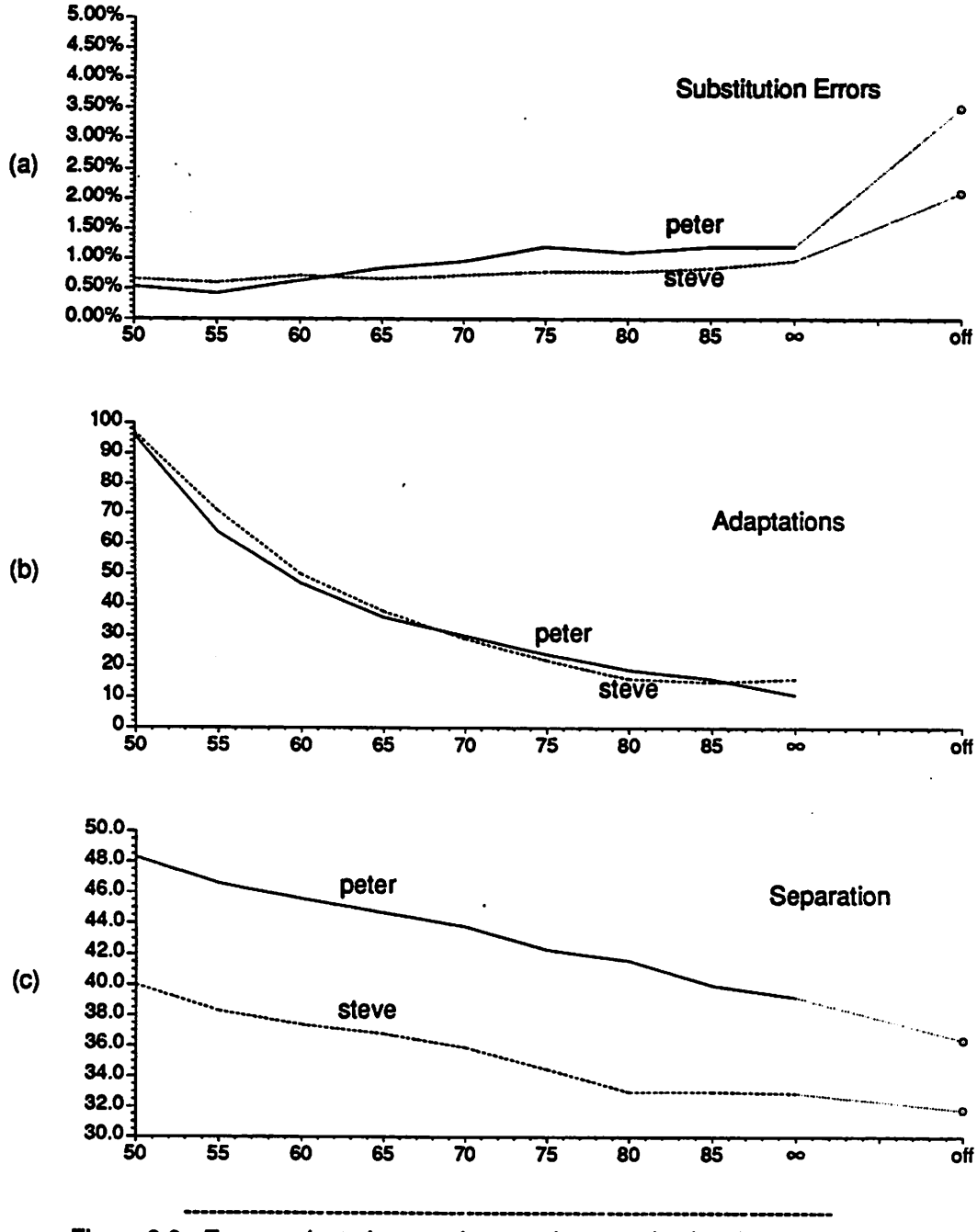


Figure 3.9 - Errors, adaptations, and separation vs. rejection threshold: $T_{adapt} = \infty$

unknown word database. The prompted templates were created by averaging two prompted utterances of a word. The spontaneous templates were generated by averaging two spontaneous utterances of a word. If a word did not have two spontaneous utterances then the prompted template was used instead. Table

Table 3.4 Multiple Pronunciations: rejection threshold = 50

speaker	T_{adapt}	substitution errors	words with two templates
peter	100	5	0
	95	5	0
	90	5	1
	85	5	1
	80	5	1
	75	6	3
	70	6	7
	65	6	11
	60	6	18
steve	100	12	1
	95	12	1
	90	12	1
	85	12	1
	80	12	2
	75	12	3
	70	12	6
	65	12	9
	60	13	18

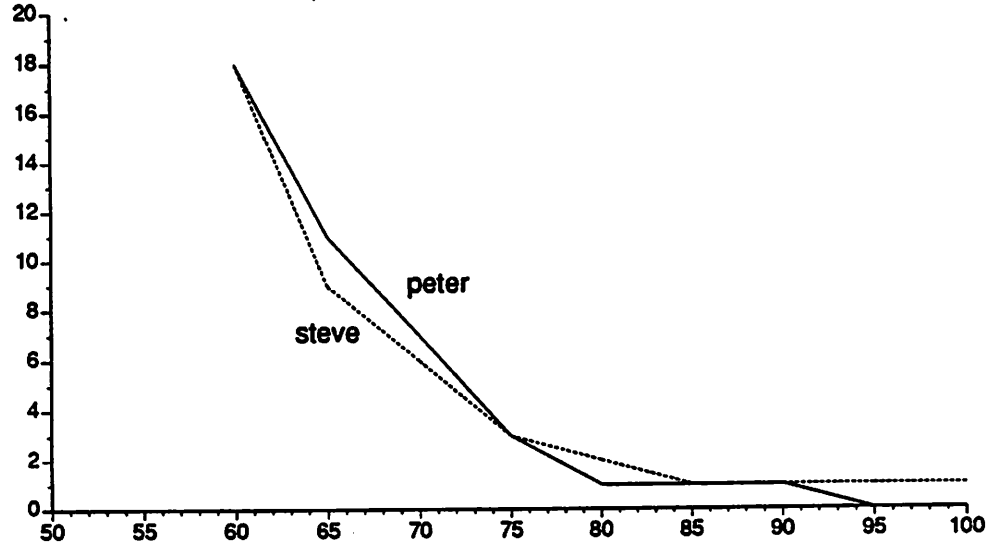


Figure 3.10 - Words with two templates versus T_{adapt} for rejection threshold = 50

Table 3.5 Classification of Multiple Pronunciations

speaker	T_{adapt}	word	difference
steve	100	move	completely different (all sounds)
	80	save	different /a/ sounds
	75	edit	different /eh/ sounds (shifted in frequency)
	70	black	missing /l/ sound
	70	blue	missing /l/ sound
	70	layer	small difference in spectral tilt
peter	90	select	no difference
	75	deselect	no difference
	75	zoom	stressed and unstressed /z/ sounds

3.6 shows the results of the experiment.

Table 3.6 Prompted vs. Spontaneous Training

speaker		substitution errors	separation
peter	prompted	32 (3.5%)	36.4
	spontaneous	12 (1.3%)	45.3
steve	prompted	35 (2.1%)	31.8
	spontaneous	51 (3.1%)	37.4

Note that even though the separation increases for spontaneous templates the error rate also increases. This discrepancy can be explained by plotting the distribution of separations (figure 3.11). Values of separation less than zero indicate substitution errors. While the spontaneous utterances have a higher average separation, the lower tail of the distribution is not shifted. When examined using the comparison tool descibed above, these tails were found to correspond to different pronuncations of the words. For example, figure 3.12 shows the spontaneous templates for the word 'move' compared to a later spontaneous utterance of move. Note that the trailing /v/ sound is completely absent in the template, but not in the utterance. This leads us to conclude that the variation of pronunciations in spontaneous speech is greater than the variation in prompted speech. Thus templates generated from spontaneous speech must be averaged from more utterances or a wider selection of utterances than templates generated from prompted speech. The adaptive training algorithm does exactly that: templates can be generated from any utterance of a word.

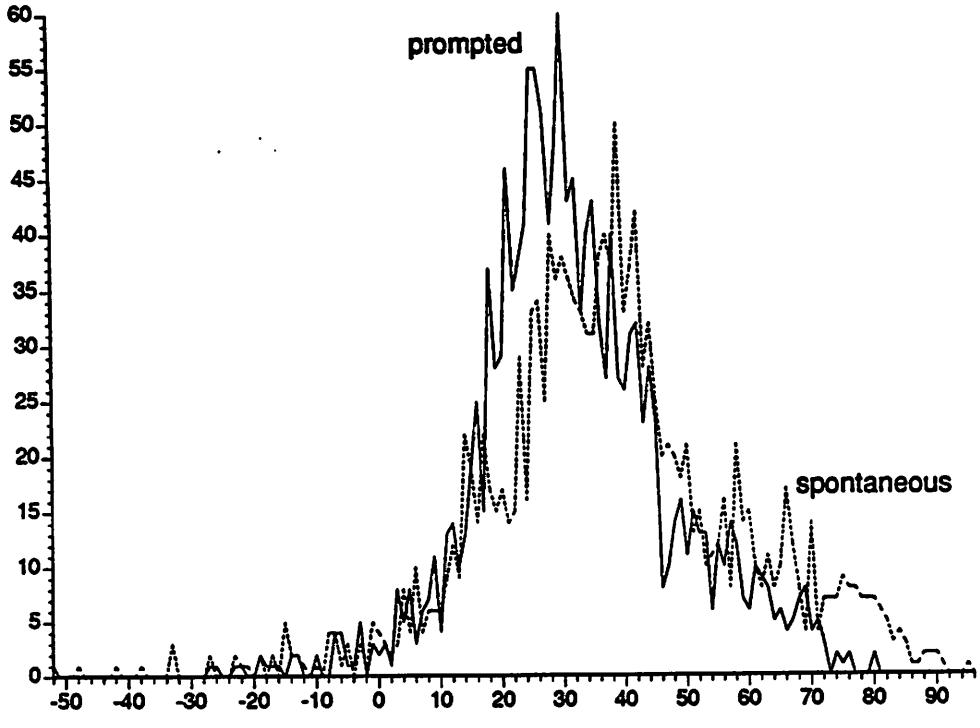


Figure 3.11 - Distribution of separations for speaker steve

It is interesting to note that the machine time required to perform the experiments above was only 7.2 hours because the special-purpose hardware was used. These same experiments, if performed on a computer able to compute 10 word-to-word distances in 1 second, would require 6 days of computer time.

3.9.2.6. Conclusions

The adaptive training algorithm was treated favorably by users for three reasons. First, recognition accuracy increases when adaptation is performed. As one can see from the experiments above, recognition accuracy increases very quickly with each new adaptation. Thus a little extra work provides the user with a dramatic increase in accuracy. Second, the algorithm can sucessfully find new pronuncations of a word allowing the user to speak more naturally. This feature is important for template-based systems because each word is trained by example, and an initial template does not necessarily correspond to a

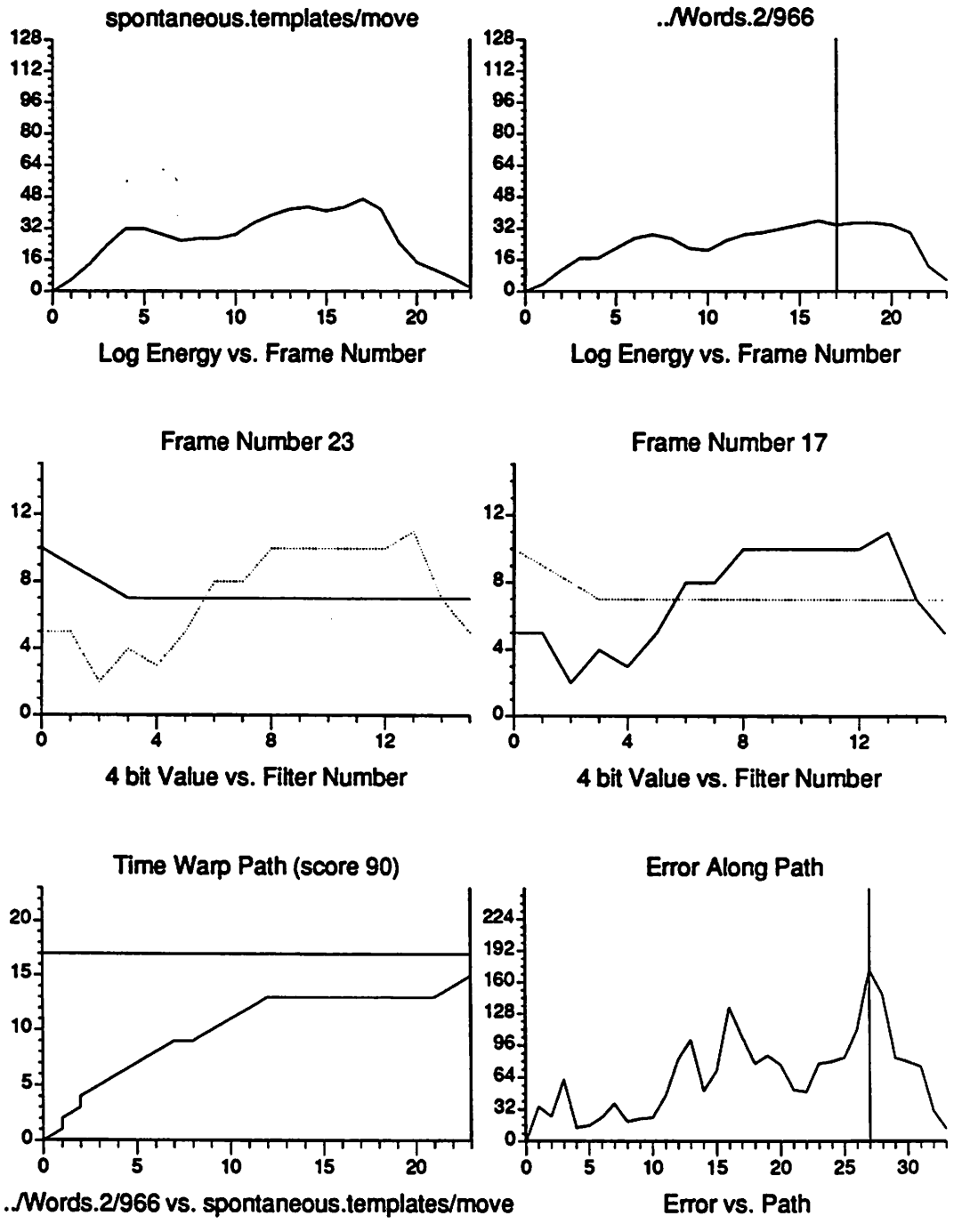


Figure 3.12 - Spontaneous template for word 'move' compared to later spontaneous utterance

natural pronuncation. Finally, from the user interface point of view, the adaptive training algorithm gives the user a chance to react positively when the recognizer makes a mistake. Before the adaptive algorithm was developed, users would react to a recognition error by either talking louder or enunciating clearer. This reaction is unfortunate because it does not increase recognition accuracy. In order to increase accuracy the user must speak to match the template, which was not necessarily created from well enunciated utterances. The adaptive training algorithm allows the user to react by telling the recognizer that it made a mistake, allowing the recognizer to correct that mistake in the future.

Chapter 4 - Interface Styles

4.1. Introduction

The final system must use speech effectively without limiting the types of applications and the effectiveness of those applications that currently run on engineering workstations. This chapter discusses how speech input devices can be used with four different styles of user interfaces: text/data entry, menu-based interfaces, text-based command interfaces, and pointing interfaces. These styles cover most user-computer interaction in use today. Experimental and application specific interface styles such as natural language interfaces are not covered.

4.2. A Practical Consideration

There is one crucial practical consideration in the design of a speech input device: speech is a newcomer to the input device field and existing applications are tightly coupled to other input devices, especially keyboards. The first concern of any speech input designer should be: "can I run existing applications?". The answer had better be "yes!" otherwise the speech recognizer will not be too useful. It is unreasonable to expect other people to change their applications to suit a speech recognizer especially when speech input devices are not generally available. Speech input devices must be retrofit devices that can talk to any application by simulating an existing input device (i.e. a keyboard).

While UNIX provides a mechanism to simulate keyboard input (tty/pty devices), the SUN window system used as part of this project does not supply an equivalent for mouse input or window-driven keyboard input. The SUN window system tags each event (each keystroke and mouse action is called an event) with the x and y location of the mouse when the event occurred and a time stamp to allow applications to detect double-clicking of mouse buttons. Because the window system is implemented in the UNIX kernel and the kernel was constantly being updated during this project, we felt that it would be too difficult to make substantial changes to the window system. Instead, the Mara system was retrofit to the window environment by supplying the applications programmer with a library of routines that provide speech input events much like the SUN window system provides keyboard and mouse events. This

scheme works reasonable well, but a more integrated scheme would be better. Most speech interfaces, however, use the standard tty/pty terminal emulator for speech input.

4.3. A Model of Speech Input

Before examining different styles of user interfaces, the speech recognizer should be modeled in terms of the speed, type, and flexibility of input that it can supply to an application. A speech recognizer can be thought of as a keyboard with only function keys. The labels on these keys can be changed, and new keys can be created at any time. The difference between a keyboard and a speech recognizer is that a speech recognizer has no alphabetic keys. Words are not created by concatenating letters, but by adding new function keys. The alphabetic keys on the keyboard are normally used to both create new items (files, text, commands, etc), and reference old items. This is not the case with a speech recognizer. Speech recognizers cannot create new items spontaneously, but must be told the name of an item before it can be referenced.

To make a speech recognizer more like a keyboard it must be able to take speech as input, then create strings of characters as output, all with no predetermined vocabulary. This task is far beyond the capabilities of current speech recognition systems and is not being considered as the goal of any major speech project.

The speed of speech input is limited by the speed that the user can speak, the processing power in the speech recognizer, and constraints placed on the way the user must speak. The major constraint placed on the user in a speech recognition system is whether words must be spoken in isolated or connected form. Isolated word recognizers require the user to pause briefly before and after each word. Connected word recognizers require no pauses and thus the user can talk faster. Even for isolated word recognizers such as Mara, the user is not significantly slowed by the speech recognizer. The most significant deterrent to high speed speech entry is the user: the user cannot think as fast as the speech recognizer can recognize. Gould[39] showed that in a letter dictation application users were not significantly slowed down when speaking words in isolation. The Mara speech recognition system can recognize between one and two words per second, although we will see later that the user typically

speaks much slower.

The slow speed of the input device when combined with its moderate sized vocabularies suggests that, for a workable interface, each word must perform the equivalent of many keystrokes. Speech commands should be macro commands, combining many basic functions into one word.

4.4. Text and Data Entry

Text and data entry entails the transcription of large amounts of text into a computer for processing at a later time. The creation of a new letter, paper, or computer program starts with a long monologue by the user. During this monologue, the thoughts of the user are transcribed into a computer-readable form. Normally the transcription is performed by the user in the form of typing. The transcription process requires a simple interaction with the computer: feedback from the computer is in the form of a display of the transcribed material. Depending on the application, vocabulary sizes for a transcription interface might be very small (e.g. 10 digits) or very large (e.g. the entire English dictionary). For a memo-writing application, IBM[40] has found that a 93% coverage of the vocabulary can be achieved with 5000 words.

The expected vocabulary sizes for computer programs was determined by examining the number of names in a typical set of programs. The set used was the standard UNIX programs including those in "/bin", "/usr/bin", and "/usr/ucb". A total of 133 programs were examined. The average number of names in these programs is 109, the median is 80, and the range of values is very broad, from 9 names to 424 names. Only 10 programs contain more than 250 names, 3 programs more than 300 name, indicating that many programs can be entered by speech even if all names must be trained.

Text entry interfaces also require some editing capabilities. Simple commands such as "delete the last word" and "delete the last sentence" are sufficient for transcription. More complex editing features are part of either the menu-based or text-based command interface style.

Speech recognition systems have long been thought of as the perfect input device for text entry. Efficient speech input would eliminate the need for a user to learn to type efficiently. Unfortunately, due to the limits on vocabulary size and accuracy of current speech recognizers, entry of English text through speech input has not been successful. Even so, it is with respect to the text entry interface style that most

speech recognition systems are evaluated.

To increase recognition accuracy for English text entry, a language model of English is usually applied to the output of the recognizer. The output of a recognizer is an ordered list of word possibilities and corresponding scores. Given that some words are more likely than others, the recognizer may choose the second word in the list if its score is close to the first word and it occurs more often in English text. This process can also be applied to word pairs, and triples. Bahl[41] has proposed such an English language model for an office dictation speech recognition system using probabilities derived as a weighted sum of single, double, and triple word probabilities. For a 2000 word vocabulary, the word error rate decreased from 20.3% to 2.5% when the language model was used.

Text entry in an engineering workstation environment is normally much more structured than just English text and therefore does not require a language model. Instead, text entry should be coupled with an editor to provide an efficient method of creating structured text. The next section describes such an editor for 'C' programs.

4.5. Menu-based Interfaces

Menu-based interfaces are a popular interface style and are currently associated with the phrase "user-friendly". These interfaces are used to enter both commands and data. The idea behind a menu interface is simple, all commands and data are presented to the user in a list called a menu. The user then selects one of the items in the menu either by entering a code associated with the item (usually a single keystroke) or by pointing to the item with the mouse. If the item is a command then it is executed, if data then the data is entered.

There are many variations on this basic theme. Sometimes menus become too long to display on one screen. To solve this problem the menu is split into sub-menus requiring the user to first select the correct sub-menu, then select the correct item. Another limit that forces the use of sub-menus is the shortage of keys on a keyboard. Often keyboard-based menu systems such as editors do not display the menu on the screen, but the menu can be displayed at the users request (a "help" feature). Some menus are displayed in a fixed place on the screen, others "pop-up" when and where required.

Menu systems usually provide feedback to inform the user which command is currently being executed. Feedback forms range from highlighting or blinking the item on the screen to displaying the item in some obvious spot on the screen.

In addition to commands and data, menu interfaces are also used to change and display the state of an application. For example, for a drawing application state information might include such items as current color, line style, and fill pattern. For state information menus, feedback is usually provided to inform the user about the current state of the application.

Speech input is well suited to menu-based interfaces. A simple speech menu system would associate each word with a menu item. Because menus are normally predetermined (i.e. the commands are fixed by the application), the vocabulary for the speech recognizer can be selected to avoid confusing words, increasing recognition accuracy. Often, speech menus are not displayed because the association between a command and a spoken word is easier to remember than between a command and a single keystroke. For example, the commands "move to previous line, character, and word" might correspond to the keystrokes "e", "h", and "a" as they do in the text editor WordStar,[42] while those same commands could correspond to the spoken phrases "previous line", "previous character" and "previous word". Even with a speech recognizer as an input device, a "help" feature is always desirable.

4.5.1. A C Program Editor

A simple text entry interface and a menu-based interface can be combined to form a powerful speech editor for structured text. One common example of structured text entry in an engineering workstation environment is computer programming. Computer programs are written in a language that has a syntax which can be exploited to aid the user in transcribing a program. Syntax-directed editing has been previously examined in the context of keyboard and mouse input. Such editors are called structure editors because they know about the structure of the data that is being edited. A complete description of the structure editors can be found in [43].

The speech-driven editor developed for this project is not a real structure editor, instead it borrows some ideas present in structure editors to help the user write a program more efficiently. In particular, the

editor automatically indents a program as it is being edited, and has a simplified input vocabulary that is not "character-oriented" instead it is "function-oriented".

The editor is written in the extension language provided by a version of the EMACS[44] text editor. The language was complete and powerful enough so that no changes to the editor were required.

4.5.1.1. Indentation

The editor keeps track of the indentation level for all program statements, not just the current indentation level. When a new statement as added, its level is computed by searching backwards for the start of its corresponding statement block. Commands that generate new statement blocks set the indentation for all statements in the block. Labels are shifted left by one half of the statement block indentation.

4.5.1.2. Cursor and Regions

The editor has its own cursor that is controlled independently from the mouse cursor. The user can move the editor cursor to the character in the text nearest the mouse cursor by clicking the left button on the mouse.

In addition to a cursor, the editor has an invisible marker that is used to define a region. The marker is set either by using the "mark" command, in which case the marker is placed at the cursor's location, or with the mouse by pressing the left mouse button then moving the mouse. In this case the cursor is placed where the mouse button is lifted and the marker is placed where the button was first pressed. Regions are used to select text to be deleted, moved or otherwise edited.

4.5.1.3. C statements

There are five executable statements in the 'C' language: "while", "if", "for", "switch", and "if_else"; and there is one corresponding word for each of these. Each statement contains one expression and one or more statement blocks. When one of the words above is spoken, the editor first opens the expression for editing, then places the token */*next-statement*/* at each statement block yet to be edited. For example, the while statement produces the following text:

```
while(^) {
        /*next-statement*/
}
```

The '^' character indicates the location of the editor cursor after the command is executed. After the expression is entered, the command "next" is used to jump forward to the next token, then a new line is opened above for editing. A semicolon is appended to each 'C' statement automatically. For example, after the word "true" and "next" are spoken for the example above, the following will appear on the screen:

```
while(TRUE) {
        ^;
        /*next-statement*/
}
```

When a statement block is finished, the "done" command is used to remove the */*next-statement*/* token and proceed to the next token.

Statements can also be inserted between two line by moving the cursor to the line after where the statement is to be inserted, then issuing the "add_statement" command. This command inserts a */*next-statement*/* token and opens a new line for editing.

Declaration statements use a different token than executable statements: */*next-declaration*/*. The "add_declaration" command is the equivalent of the "add_statement" command for declarations.

Structure definitions require one additional command "new_struct" to set up the structure definition braces. Labels are emitted by the command "label", and case labels are emitted by the word "case".

4.5.1.4. Expressions

Expressions are entered from left to right using the words in table 4.1 with two notable exceptions. Parenthesized expressions and subscripts are entered as balanced pairs. The command "expression" enters "(^)" and "subscript" enters "[^]". The command "move_outside" jumps the cursor forward just outside an expression or subscript. For example, the expression "(a+b)*c" is entered with the word sequence "expression a plus b move_outside times c". This method of entering expressions forces expressions to be balanced at all times. Parentheses and subscripts can be placed around an expression

Table 4.1 'C' Editor Text Entry Vocabulary

Word	Text	Word	Text
and	&&	break	break
char	char	comma	,
continue	continue	decrement	--
dot	.	double_quote	"
equal	=	equals	=
extern	extern	increment	++
int	int	is_equal_to	==
is_greater_than	>	is_greater_than_or_equal_to	>=
is_less_than	<	is_less_than_or_equal_to	<=
minus	-	not	!
or	||	plus	+
points_to	->	register	register
return	return	semicolon	;
single_quote	'	space	
star	*	static	static
struct	struct	times	*

by putting a region around the expression, then issuing either the command "expression_around" or "subscript_around".

4.5.1.5. Variable and Procedure Names

Most variables and procedure names are not part of the standard vocabulary but must be added on an as needed basis. Each new variable and procedure name is first created by typing it on the keyboard. The commands "train_variable" and "train_procedure" are then used to load the word into the recognizer's vocabulary. These commands inform the recognizer (through the byrne shell described later) to associate that word with either the variable or procedure word type (described later). When a variable is spoken it is entered as though it was typed, while a procedure is entered then appended with the string "()". When a procedure is defined the "new_procedure" call is used to generate the following:

```
...()
{
        /*next-declaration*/

        /*next-statement*/
}
```

The phrase "main_procedure" generates a procedure definition with *argc* and *argv* parameters already

defined.

A set of standard variable and procedure names is included in the basic editor vocabulary. These are shown in table 4.2.

Table 4.2 'C' Editor Standard Variables and Procedures

Variables			
Word	Text	Word	Text
0	0	1	1
2	2	3	3
5	5	6	6
7	7	8	8
9	9	arg_c	argc
arg_v	argv	false	FALSE
i	i	j	j
k	k	null	NULL
standard_error	stderr	temp	temp
true	TRUE		
Procedures			
f_print_f	fprintf	exit	exit
main	main	print_f	printf
print_usage	printusage		

4.5.1.6. Editing Commands

The most important editing command is "delete". The editor knows the meaning of most symbols and knows how expressions and statements are constructed, so the "delete" command can be intellegent. For example, when a left parenthesis is deleted the corresponding right parenthesis is also deleted. The delete command also deletes entire variable names and special symbols.

Entire lines can be deleted with the "cut_lines" command. The lines to be cut are those defined by the current region. Lines must be balanced, that is, the indentation level of the lines must never precede the indentation level of the first line, and the first and last lines must be indented to the same level. The deleted lines are placed in a special buffer so that they can be put back into the program at a later time with the "put_lines" command. When lines are replaced they are automatically reindented. The "select_lines" command performs the equivalent of the "cut_lines" command except the lines are not removed from the text.

Regions within a line can also be cut and put back using the "cut_region" and "put_region" commands. These commands check to make sure that expressions in the region are balanced.

Other miscellaneous editing commands include "indent_more", "indent_less", "move_inside", "delete_useless_line", and "insert_blank_line". The "delete_useless_line" command is used to get rid of lines with only a semicolon that are created when editing is performed out of sequence.

In addition to editing commands, the editor also has standard searching, read/write, movement, and undo commands.

Table 4.3 'C' Editor Commands

beginning_of_file	case	compile
cut_lines	cut_region	default
define	delete	done
end_of_file	expression	expression_around
for	if	if_else
include	indent_less	indent_more
insert_file	label	main_procedure
mark	move_inside	move_outside
new_procedure	new_struct	next
put_lines	put_region	quit
replace	search	select_lines
select_region	subscript	subscript_around
switch	train_procedure	train_variable
visit_file	while	write_file

The entire standard vocabulary for the editor is 103 words. In addition each program requires one word per variable and procedure. For an average program this comes to a total of 211 words.

Figure 4.1 shows an example program and the word sequence that created the program. The text inside the '/' characters was typed, not spoken.

4.5.2. Connected Words

Connected word recognizers would probably not improve the performance of menu-based interfaces very much. Each word or phrase is associated with a function and feedback is on the word or phrase level. When using recognizers that make mistakes, the user normally waits for the feedback from one word before the next word is spoken. In such a case, the user pauses between words eliminating the

Program Listing

```
main(argc, argv)
        int  argc;
        char *argv[];
{
        int  i, limit;

        if(argc < 2) {
                printusage();
        }
        limit = atoi(argv[1]);
        for(i = 0; i < limit; i++) {
                printf("%d %d\n", i, i*i);
        }
}
```

Word Sequence

```
main main_procedure
int i comma /limit/ train_variable done
if arg_c is_less_than 2 next
printusage done
limit equals atoi argv subscript 1 next
for i equals 0 semicolon i is_less_than limit semicolon i increment next
print_f /"%d %d\n"/ comma i comma i times i done
done
```

Figure 4.1 - Example program and corresponding word entry sequence

need for a connected-word recognizer. A connected-word recognizer would however improve the performance of a combined text-entry/menu-based interface because the user does not normally wait for text to be displayed during text entry.

4.6. Text-based interfaces

In addition to menu-based interfaces, many applications specify commands as text. This is especially true for keyboard based systems such as UNIX. Text commands differ from text entry in four key ways:

- The command language is not English and often uses special symbols heavily. Commands are not word oriented, but character oriented.
- Text commands often manipulate items such as file names that are not predetermined but are

created by the user for reference at a later time.

- The last word in a command cannot always be determined by the command, but is usually indicated with a special key (the return key).
- Feedback supplied for text commands is normally a transciption of the character sequence of the command. Because commands are not made of words but characters, one cannot read the command unambiguously.

These differences conspire to make text commands very awkward for speech input.

4.6.1. The Byrne Shell

One example of text-based commands in UNIX is the shell command processor. The shell is an interactive command interpreter that is used to run application programs, manipulate files, browse through data files, and control running applications. The structure of a shell command is linked closely to the structure of the UNIX operating system, and is detailed in [2].

Adding speech input to the existing shell program would have been to expensive with respect to development time. Instead, a new shell, called the Byrne shell was developed to interface to both an existing shell command interpreter and any application that requires a command interpreter. The Byrne shell converts speech commands into character strings and sends those strings to either an application or a shell for interpretation. The application is connected through a tty/pty device pair allowing any existing application to accept speech input that can accept keyboard input (except those application that use window system interface directly).

Along with managing speech input, the Byrne shell provides a graphical multiprocess management scheme using icons and a history mechanism that allows the viewing of the last screen of each application.

4.6.2. Speech Input

The speech input mechanism is word-oriented. This means that feedback is provided to the user as a sequence of words, not word meanings. For example the command "ls a/b" is shown as "list b in a".

Note that "b in a" could also be spoken as "a slash b". The command for either the current application or the Byrne shell is displayed in a special window called the command window. The words in this window can be edited using a built-in mouse-driven cut and paste editor or with the word commands "delete" and "delete_all".

4.6.2.1. Grammars

Commands are entered as a string of words but interpreted according to a grammar. The grammar converts a command word string into a character string suitable for an application. Each application has its own grammar as does the Byrne shell itself. The Byrne shell currently uses a simple precedence parse like that described in [45].

A grammar consists of a set of word types (terminal symbols), productions (reduction rules), and no reduce rules (precedence table). The entire precedence table is not computed automatically, but must be supplied by the application in the form of no reduce rules. Each no reduce rule specifies an "equals" precedence for a pair of symbols. A better approach would be to compute the required parsing table directly from the grammar productions.

Each word in the Byrne shell has both a type and a value. Types are the terminal symbols for the grammar and are either local to an application or global so that they can be referenced by the user. Values are character strings that are used as the data to form the character string sent to an application. Each production specifies both the reduce rule to convert a handle into a non-terminal symbol and the corresponding semantic operations to create the character string.

Figure 4.2 shows the grammar for numbers. Statements in a grammar file specify either a production, no reduce rule, or word. Productions consists of the left-hand-side non-terminal symbol, the character operation string, then the handle. Character operation strings are coded as a string of characters where each field in the form "%<number>" is replaced with the string for the appropriate handle component. The leftmost handle component is "%0". Thus the operation "%00" for the handle "ty_number" appends the character "0" to the string associated with "ty_number". Words are specified as word, type, and value. In the case of the number grammar, words are global, meaning that all numbers should be

```
#
# Standard Number grammar
#

production number "%0"          digit;
production number "%0%1"        ty_number number;
production number "%00"         ty_number;
production number "%0%1"        number number;
production number "%0"          teen_number;

no_reduce_rule ty_number digit;

global_word 1 digit 1;
global_word 2 digit 2;
global_word 3 digit 3;
      ...
global_word 0 digit 0;
global_word oh digit 0;

global_word 20 ty_number 2;
global_word 30 ty_number 3;
global_word 40 ty_number 4;
      ...
global_word 90 ty_number 9;

global_word 10 teen_number 10;
global_word 11 teen_number 11;
global_word 12 teen_number 12;
      ...
global_word 19 teen_number 19;
```

Figure 4.2 - Byrne shell grammar for numbers

interpreted the same by all applications.

A command is parsed successfully if it terminates with only the non-terminal "terminate".

4.6.2.2. Words

Words come from two different sources: the application and the user. Words that come from the application are either hard-wired into the grammar file of the application as terminal symbols or included as members of a local or global type. Words that come from the user must be a member of a global type. Global types include things such as "files", "variables", "procedures", and "users". Some global types are

specific to an application others are used in all applications. A type is made global with the "type" statement in a grammar file.

Words that are added by the user are global, that is, they apply to all applications. These words are organized into projects. When the Byrne shell is started, a project name is specified on the command line. The project name corresponds to a file in a project directory that contains all the user-specified global words for a particular project. As a given project proceeds its project file grows. When a new project is started, the Byrne shell's vocabulary starts from scratch.

New words are added to the shell by typing into the command window. A special word window is then popped up over the command window and the user can enter the new word, value and type. The shell will then load the new word and update the current project file.

4.6.2.3. End of Command

With speech input it is difficult to determine the end of a command. Some applications, especially menu-based applications, require only one word per command. Other applications such as the shell require the user to specify the end of the command because it cannot be determined by looking at the word sequence. In order to reconcile these problems the Byrne shell has two end of command indicators: SEOC and EOC. These indicators are appended to each command string before it is parsed. If the grammar can reduce the command to a single "terminate" non-terminal then the command is processed. SEOC is used for menu-based systems, and EOC is used for standard command systems. Both can exist in the same grammar file, that is, some aspects of the application interface might be menu-based, others command based.

The SEOC indicator is appended and parsed after each word is spoken. If the parser is successful then the resulting character string is sent to the application and the command window is cleared.

The EOC indicator is appended and parsed only when the end of a sentence is detected (about 1.5 seconds of silence after the last spoken word). Even this scheme did not work well to detect the end of a command so it was modified to include a "stop/go" indicator. The problem with detecting the end of a command occurs when the recognizer makes a mistake, or more commonly when the user has to think

about the command for more than 1.5 seconds. In either case, the shell will parse an incomplete or erroneous command.

The "stop/go" indicator appears in the information window of the Byrne shell as either "stop" or "go". When in the "stop" mode, the end of sentence is ignored. The indicator changes from "go" to "stop" whenever a rejection error occurs or an editing command is used. This allows the user to say "uh" or some other untrained word and have the recognizer stop attempting to interpret the command. The indicator can be changed from "stop" to "go" by saying the word "go". When the command window is empty, the indicator always shows "go".

The advantage of this scheme is that if the user knows what he is going to say and the recognizer works perfectly then there is no need to say "go". If the user has to think, create a command with the built-in editor, or not wait 1.5 seconds then the user must say "go".

4.6.2.4. Some Examples

The description of the interface can be clarified by a few examples. First, the grammar for the 'C' program editor described above is shown in figure 4.3. Notice that most commands are members of type "immediate" which, according to the grammar, uses the SEOC indicator to send the command immediately to the application. The only words that are processed by the shell are file names, which end with an EOC indicator.

The grammar for the mail application is shown in figure 4.4. The mail application has short commands that end with an SEOC, and long commands that end with an EOC. Figure 4.5 shows the same grammar in syntax chart form.

4.6.3. Implementation

The Byrne shell runs under the SUN window system. The shell is started with the command "byrne <project name>". The shell starts by reading the master file, then the project file. An application file is read only when its application is executed. The format of the application, master and project files is the same: each statment in a file contains a list of strings terminated with a semicolon. Characters

```
production terminate "%0"   immediate SEOC;
production terminate "%0"   immediate EOC;
production terminate "%0"   variable SEOC;
production terminate "%0"   variable EOC;
production terminate "%0\emake-expression\r"     procedure EOC;
production terminate "%0\emake-expression\r"     procedure SEOC;
production terminate "%0\r" full_file EOC;

type immediate variable procedure;

include full_file.grammar;

#
# Basic emacs editing words

word quit               immediate "\exexit-emacs\r";
word mark               immediate "\exset-mark\r";
word compile            immediate "\exnew-compile-it\r";
word search             immediate "\exsearch-forward\r";
word replace            immediate "\exquery-replace-string\r";
word beginning_of_file  immediate       "\exbeginning-of-file\r";
word end_of_file        immediate "\exend-of-file\r";
word visit_file         immediate "\exvisit-path-file\r";
word insert_file        immediate "\exinsert-file\r";
word write_file         immediate "\exwrite-file\r";

#
# Other word files

word load_c_words    immediate
"\exload\r~kavaler/emacs/mara_c.ml\r";
```

Figure 4.3 - Grammar for C program editor (emacs.app)

between a '#' character and the end of a line are ignored to allow comments. The first string in the statement is the command, the rest of the strings are the data for the command. Strings can be placed in double quotes to include spaces and special characters.

4.6.3.1. Windows

There are four separate windows for each Byrne shell: the information window, the icon window, the command window, and the application window. The information window contains one line for text messages to the user and one line to display the state of the shell. State information for the shell includes

```
include          full_file.grammar;
include          number.grammar;

production number      "%0"          anything;
production terminate   "%0\n"        number SEOC;
production terminate   "%0\n"        number EOC;
production terminate   "%0\n"        immediate SEOC;
production terminate   "%0\n"        immediate EOC;
production terminate   "%1\n"        read number EOC;
production terminate   "s %1\n"      save full_file EOC;
production terminate   "s %1\n"      save into full_file EOC;
production terminate   "%0\n"        long_command EOC;
production terminate   "%0 %1\n"     long_command number EOC;
production terminate   "%0 %1-%3\n"  long_command number up_to number EOC;
include          mail.unix;
production terminate   "%0\n"        a_command EOC;

word read              read          read;
word save              save          save;
word into              into          into;
word up_to             up_to         up_to;
word remove            long_command  "d";
word preserve          long_command  "pre";
word letters           long_command  "h";
word all               number        "*";
word quit              immediate     "q";
word x                 immediate     "x";
word help              immediate     "?";
word respond           immediate     "R";

include special.words;
```

Figure 4.4 - Grammar for mail application (mail.app)

the command window indicator, the "stop/go" indicator, the update button, and the current directory.

The command window indicator shows which command is being displayed in the command window:

- the shell command, (a white crystal ball),
- the application command (a black crystal ball), or
- a previous command (a pair of glasses).

Normally the shell command is displayed. If an application is running, its command is displayed. The

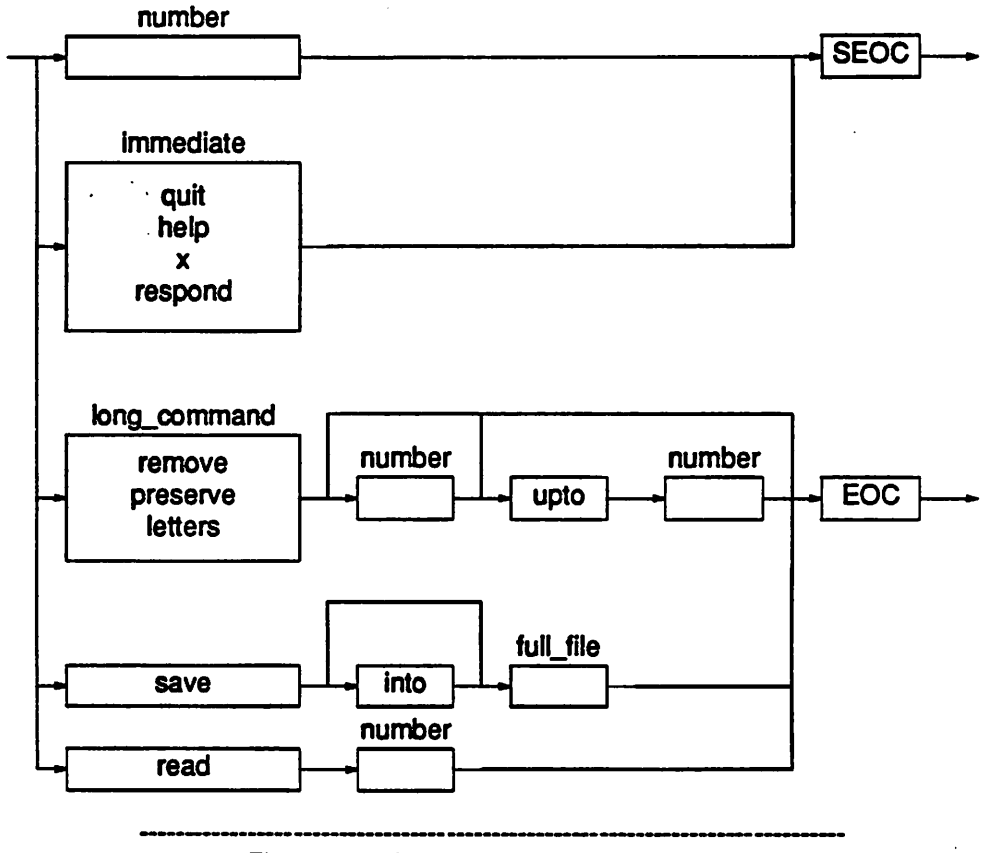


Figure 4.5 - Syntax chart of mail grammar

shell command can be displayed when an application is running by pointing to the icon window. If the right mouse button is pressed over an icon, then the command that created that icon is displayed. The update button, when clicked, rereads all the project, application and master files, to allow the user to change these files while the shell is running.

The icon window shows the history of executed applications as a list of icons. When a new application is started, its icon is shifted into the icon window from the left. One icon is always highlighted (in reverse video) and the application for that icon is shown in the application window. Only one application can be viewed at a time. If an application is running then the label RUN is placed over its icon. An icon is selected by clicking the left mouse button over the icon.

The application window shows the last screen full of data for each application. If the application is

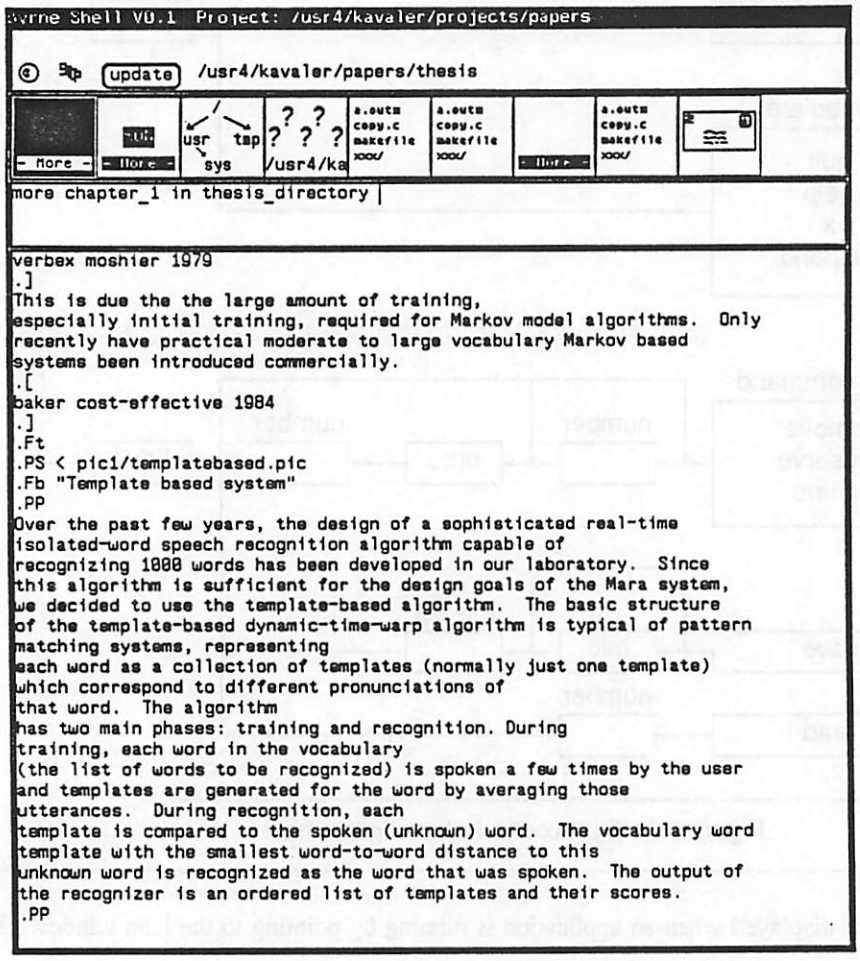


Figure 4.6 - Byrne shell windows

running then the application window is alive and the application can change the window as though it is a terminal (a terminal emulator). After an application dies, its last screen image (called its corpse) can still be viewed but not modified.

4.6.3.2. Master File

The master file has three functions: to control the paths prepended to each application and icon file, to specify the applications and their corresponding icon and grammar files, and to define the Byrne shell grammar. Figure 4.7 shows an example master file.

```
include_path . ~/projects ~kavaler/mara/byrne/byrne_lib /usr/local/lib/byrne;
icon_path . ~kavaler/mara/byrne/byrne_lib/icons /usr/local/lib/byrne/icons;

application emacs     emacs.icon      emacs.app emacs;
application mail      mail.icon  mail.app   mail Mail;
application msgs      msgs.icon  msgs.app  msgs news;
application shell     shell.icon  shell.app  csh sh;
application more      more.icon  more.app  more;

include          shell.unix;
include          mail.unix;
include          number.words;
include          file.words;

include          aliases;
include          ~/.byrne;

include          browse.project;
```

Figure 4.7 - Example master file

One "application" statement is required for each application that needs a grammar. The statement is in the format:

application <name> <icon file> <grammar file> <other name> ... ;

The other name fields associate a shell command with an application. The icon and grammar files are read according to the include_path and icon_path lists. For the master file example shown, the grammar for the Byrne shell is split into smaller files and included using an "include" statement.

The master file is called "Masterfile" and normally lives in a special library directory. A user can customize the master file by putting a private copy in the directory "~/projects".

4.6.3.3. Grammar Files

Each application has its own grammar file. The grammar is defined by the following statements:

- "production <lhs> <operation> <handle>;" - described earlier.
- "no_reduce_rule <symbol> <symbol>;" - described earlier.
- "word <word> <type> <value>" - in an application file the word is local to the application, in a

project file the word is global.

- "global_word <word> <type> <value>;" - a global word.
- "type <type> ... ;" - a global type definition.
- "include <file> ... ;" - include another file, this is especially useful for sharing common subgrammars.
- "special_word <word> <function>;" - The Byrne shell has a few special functions that can be associated with a word. Special functions include such things as "go", "delete", "delete_all", etc. A complete menu of these functions is provided when the right mouse button is pressed in the command window.

Project files live in the directory "~/projects", while most other files live in a library directory.

4.6.3.4. Application Interface

An application can add new words or change its grammar while it is running. To do this the applications sends a special escape sequence to its terminal emulator:

\ePW<word>|<type>|<value>\e\

(\e is the escape character) will add a new word to the global vocabulary just as though it was typed into the command window. This feature is used by the 'C' program editor to add new variables and procedures.

The sequence

\eIW<file>\e\

will read the given file as though it was included in the original application file (i.e. an include statement).

4.6.4. Shell Usage

The standard usage of the Byrne shell differs from a normal shell. The Byrne shell is controlled mainly by a combination of mouse movements and speech commands. The mouse is used to select things on the screen and speech is used to give commands to those things. To illustate this type of interface consider browsing the UNIX file system. For keyboard input the user would perform an "ls"

command, then "cd" into a directory and "ls" again. When an interesting file appears on the screen after an "ls", the "more" command is used to show it on the screen. A typical session might proceed as follows:

```
ls
cd /usr
ls
cd local
ls
cd lib
ls
cd byrne
ls
more Masterfile
more emacs.app
```

Notice that the vocabulary for this application is constantly increasing. In fact, every command except the "ls" command requires a new word. For keyboard input this is no problem, but for speech this is a big problem because the cost of adding a new word is high.

This problem is solved in the Byrne shell which allows the user to point to a file name generated by the "ls" command and select the name by double-clicking the left mouse button. The selected name can then be inserted into the "cd" or "more" command with the special word "it". The word "it" inserts a word with type "anything" into the command. The shell grammar converts the type "anything" into the type "file" using the statement: "production file "%0" anything;". Other applications, such as the mail appliation, convert "anything" into a different type.

The "cd" then "ls" command sequence is so common that a "go_to" command was introduced to combine the two. This style of interface reduces the vocabulary for the recognizer enormously. A word is no longer required for each file name. Words are only required for those file names that referred to often.

Another Byrne shell feature eliminates the need for multiple references for an appendixed file. The C compiler, for example, will takes as input a file appended with ".c" and creates a file with the appendix ".o". If the user wishes to reference both the ".c" and ".o" files, two words are needed. The Byrne shell uses the global type "only_appendix" to strip the current appendix from a file and appendix a new string.

For example, the word "dot_oh" will change the file "main.c" to "main.o". Thus the user can define the word "main" to be type "file", value "main.c". The file "main.c" can be referenced with the single word "main", and "main.o" can be referenced by the two words "main dot_oh".

4.6.5. Connected Words

The shell interface would not need to be changed substantially to support connected-word recognition. First, the 1.5 second delay to detect end of sentence might be shortened, but because the user must think about the command to be executed the "stop"/"go" indicator is still necessary. The word oriented interface is also needed because the user is still speaking words.

The main benefit of using a connected-word recognizer would be to elimate the isolated word speech constraint on the user, providing a more natural interface. It is not clear that the resulting interface would be any faster because most commands take longer to execute than they take to speak.

4.7. Pointing Interfaces

The final interface style used by many applications is the pointing interface. We have already seen menu-based interfaces that use a mouse to point to a command, and some applications also require data entry in the form of coordinates on the screen. For example, the integrated circuit layout editor KIC[38] uses the mouse to place boxes corresponding to mask layers on the screen. Another use of a pointing interface is in video games where the mouse is used to control a player's position on the screen. Feedback for a pointing interface is usually a cursor indicating either a single point or the region that is being pointed to.

Speech is particularly inappropriate for pointing interfaces. Commands such as "up", "down", "right", and "left" are cumbersome to speak in quick succession and are slow compared to a mouse. Specifying a number with each command will speed the interface, but it also requires the user to estimate a distance on the screen accurately. Interfaces that use pointing should either be rewritten to keep pointing information to a minimum, or to use the mouse for pointing related information and the recognizer for all other information.

4.8. Speaker Independence

In this chapter we have been considering different interface styles and how a speech recognizer can be used for each style. One aspect of speech recognizer performance that has not been considered is speaker independence. Most existing speech recognizers only work well for the speaker who trained the recognizer. A speaker-independent recognizer would recognize words from anyone. The problem with these recognizers is that words cannot be added to the recognizer's vocabulary while the recognizer is running. In fact, words must usually be spoken by many people before they can be trained to work in a speaker-independent fashion. For the four interface styles detailed above this restriction limits the use of speech recognizers ONLY to interfaces with predetermined vocabularies. In an engineering workstation environment there are few programs that have a predetermined vocabulary. Some programs, however, might have a large predetermined vocabulary and a small programmable vocabulary.

Speaker-independent recognizers must be able to add new words while they are running if they are to be generally useful. The words that they add need not be speaker-independent. In fact, for the engineering workstation application there is no real need for speaker independence because the computer can keep track of who is using the recognizer very easily.

Chapter 5 - Evaluation of the System

5.1. Introduction

The Mara system consists of many components that work together to allow a person to use a computer by voice. System components such as the hardware, the speech recognition algorithm, and the adaptive training algorithm have been evaluated separately, but the system as a whole must also be evaluated. Evaluations of speech recognition systems have generally concentrated on the recognition algorithms leading to erroneous assumptions about the relative importance of various system components. For example, most commercial speech recognition systems claim 99% accuracy or more, yet these systems have had little or no commercial success. One reason for this discrepancy is that these commercial systems do not perform as well in the field as they do in the laboratory. The reason for this is that the evaluation of speech recognizers is normally performed with data-bases that do not reflect how the system is used. For example, data-bases are normally collected by prompting a user while in real life the user speaks spontaneously. Other performance issues such as user acceptability and user productivity are completely ignored. The result is a speech recognizer that works well in the laboratory but not at all in the field. The final goal of this project is to develop a method of evaluating speech recognition systems that reflects how they are used.

5.2. System Performance

The development of the user interface and its associated feedback techniques was a key factor in increasing user acceptability and system performance of speech input devices but a more formal experiment was also needed to quantify these factors. The experiment compared the same editor being used with two different input devices: a mouse and the speech recognizer. A menu-based VLSI editor called KIC[38] was used for the experiment.

KIC uses the mouse in two different ways: as a pointing device to select coordinates for geometric manipulations and as menu device to select commands from fixed menus on the left side and bottom of the screen. The total number of command menu items is greater than can fit on the screen at one time.

To solve this problem the menu is split into four sub-menus where only one is on the screen at a time. Each command menu contains a command to go to another sub-menu. During normal use a single menu, the "select" menu, is used most of the time. The menu on the bottom of the screen is used to select the current color (layer).

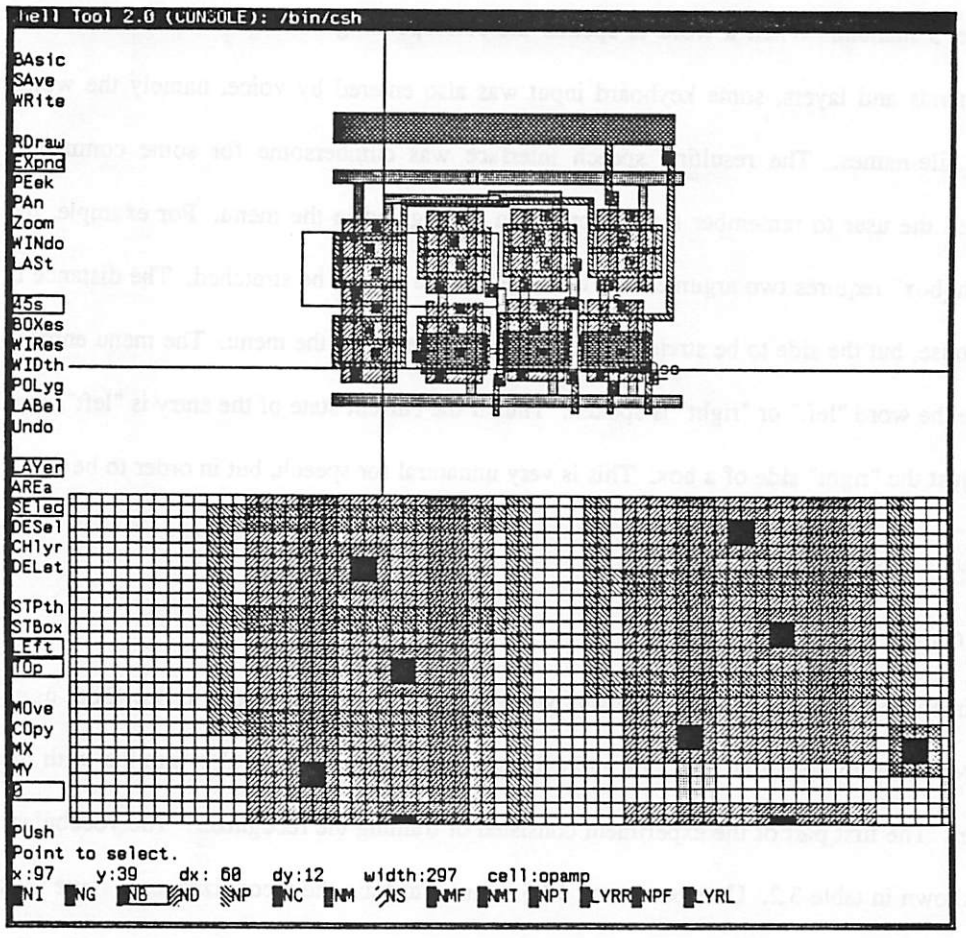


Figure 5.1 - KIC screen

The basic object that KIC can edit is a box. Boxes are placed on the screen with the command "boxes" followed by two mouse clicks indicating the diagonal corners of the box. Boxes can also be moved, stretched, copied, and deleted. For example, a box (or boxes) can be moved with the command sequence: "select" - click over the box or boxes to move - "move" - two mouse clicks indicating relative

movement - "deselect". The screen is updated after any change is made to the layout. In addition to these simple editing commands, other commands are available to check for layout rule violations, handle the cell hierarchy, and read and write files. The editor has been in use for many years and is very robust.

In order to be "fair" to the experiment, speech was retrofit to the editor in a straight-forward fashion: each command and color had an associated word. The words used for commands were taken from the KIC user's manual. When a word is spoken the corresponding command was executed. In addition to commands and layers, some keyboard input was also entered by voice, namely the words "yes", "no", and file-names. The resulting speech interface was cumbersome for some commands because it required the user to remember state information highlighted in the menu. For example, the command "stretch_box" requires two arguments: a distance and the side to be stretched. The distance is given with the mouse, but the side to be stretched is specified by a entry in the menu. The menu entry is toggled each time the word "left" or "right" is spoken. Thus if the current state of the entry is "left", saying "left" will adjust the "right" side of a box. This is very unnatural for speech, but in order to be fair it was not changed.

5.2.1. The Experiment

The experiment involved two users (steve and peter) performing normal editing tasks. Both users had used KIC previously to layout at least one entire integrated circuit and were also familiar with the speech recognizer. The first part of the experiment consisted of training the recognizer. The vocabulary used by KIC is shown in table 3.2. User peter took 12.5 minutes to train the recognizer, user steve took 14 minutes. The users were then given a specific circuit to edit. The circuit was a single stage of a switched capacitor filter. The circuit was laid out twice, once with the recognizer and once without the recognizer. The users were instructed not to race through the session but to go at a normal pace. The experiment stopped when the circuit was validated by the layout rule checker. Layout took place one hour per day over four days for user peter and six days for user steve. All editing sessions were recorded digitally using a PCM recorder and standard Beta format VCR for use as a speech data-base.

The purpose of the experiment was to determine the relative speed gained or lost by using the recognizer. Because the users were editing the same circuit twice it was unfair to just measure the total time required to perform all editing. Instead, the time between commands and the time between words was used as a normalized measurement. The time between commands and words has many advantages as a measure:

- delays caused by excessive thinking by the user or the computer can be eliminated by throwing away delays greater than a few seconds,
- many data points can be acquired, enough to estimate both the average delay and the shape of the delay distribution, and
- delays for different combinations of inputs can be measured allowing a more complete analysis of why the interface works well or poorly.

5.2.2. Results and Discussion

User actions were split into two types: pointing actions (called points) and command actions (called commands). Delays were split into four categories: point-to-command, point-to-point, command-to-point, and command-to-command. The one exception is the command "deselect" which is first a command, then a point. Thus the sequence "deselect" - "boxes" contributes a point-to-command delay instead of a command-to-command delay. This change forces the re-draw time associated with the "deselect" command, into the same category as other commands that redraw the screen.

Table 5.1 KIC Experiment With and Without a Recognizer

speaker	with/without recognizer	c-to-c	p-to-p	p-to-c	c-to-p	total	
steve	with	1.70	1.66	2.56	0.87	1.69	delay (sec.)
		13.2%	27.2%	29.4%	30.2%	1706	commands
	without	1.75	1.74	2.58	1.82	2.01	delay (sec.)
		13.9%	27.3%	29.5%	29.4%	1360	commands
peter	with	1.97	1.45	2.51	1.01	1.67	delay (sec.)
		9.5%	27.6%	29.9%	33.0%	784	commands
	without	1.47	1.30	2.18	1.46	1.62	delay (sec.)
		11.1%	32.8%	28.4%	27.7%	500	commands

Table 5.1 shows the results of the experiment. Each pair of rows in the table indicates the average delay in seconds and percentage of actions for each category along with the total average delay and total number of user actions. The average delay was computed by ignoring detectable think and redraw time. That is, delays greater than five seconds were ignored. This table shows much information about how the recognizer is actually used, and how it can be made more efficient.

The command-to-command delays accounted for only between 10 and 15 percent of the total number of delays. While one user had equal command-to-command times, the other could use the mouse menu faster than the speech menu. Even so, the overall speed of the system is not primarily determined by the speed of the recognizer alone. The three other delays: command-to-point, point-to-point, and point-to-command account equally for remaining 90% of the total delay. Because point-to-point times are not dependent on the recognizer, and command-to-point times are determined mainly by the time required to redraw the screen and think about the next command. The main recognizer related factor in determining speed is the point-to-command time. Both users experienced a large speed-up in the point-to-command delay when using the recognizer. This leads us to conclude that the speed-up gained by using the recognizer with the KIC application is primarily due to overlapping of speech commands with pointing actions. The overall speed gained by using the recognizer (total average delay) was about 15% for user steve and no gain for user peter. If just the command-to-command and command-to-point delays are considered, that is, those delays directly attributable to the recognizer, then user steve experienced a 38% speed gain and user peter a 16% gain.

Some of the speed lost in command-to-command delays was due to rejection and substitution errors. While the recognizer performed well during all experiments, the rejection threhold was lowered and raised during the experiment to simulate different rejection error rates. The result was that a few session contained many rejection errors slowing down the user. Table 5.2 shows the distribution of substitution errors and rejection errors each day for both users.

It is also interesting to look at the distribution of delays for each category. These plots are shown in figures 5.2 and 5.3 for speaker steve. The dashed lines show the delay distribution without the recog-

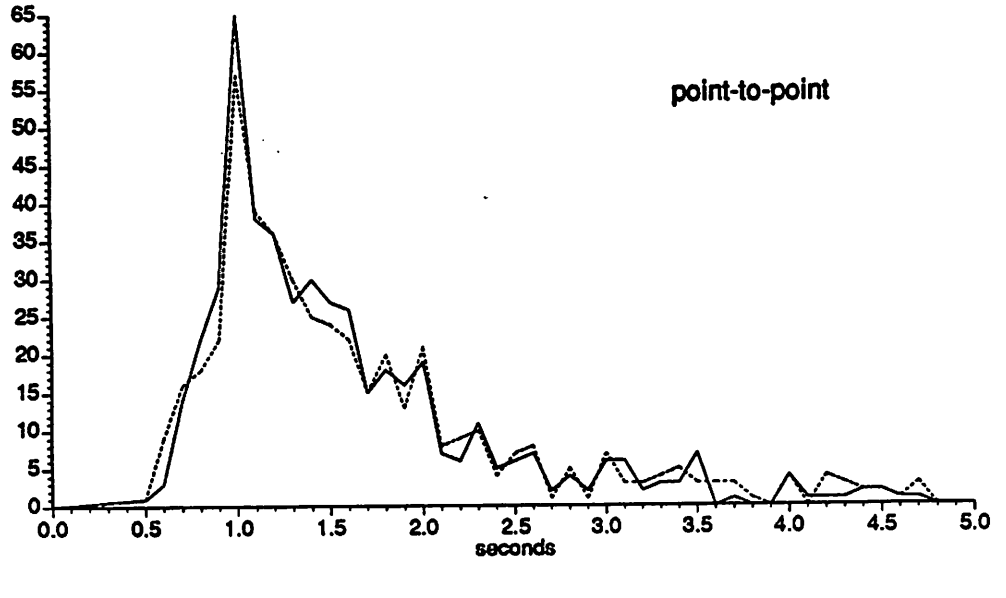
Table 5.2 Errors During KIC Experiment

speaker	day	substitution errors	rejection errors	total errors
peter	1	1	11	12 (3.1%)
	2	5	30	35 (8.2%)
steve	1	6	34	40 (7.7%)
	2	8	2	10 (2.0%)
	3	1	24	25 (4.1%)

nizer while the solid lines are with the recognizer. The plots are normalized so that the area under each plot is 500.

Notice that the distributions in figure 5.2 are identical with and without the recognizer while the other two distributions, command-to-point and command-to-command have different shapes. This supports the claim that only the command-to-point and command-to-command distributions depend on the recognizer. There are a few anomalies in the plots that also deserve an explanation. First, the distributions tend to peak near intervals of one second. This is not because of user actions, instead the measuring technique used depends on the internal clock kept by the UNIX operating system. This clock seems to change slowly at second intervals. This hypothesis was verified by feeding a series of beeps into the recognizer with a flat delay distribution and measuring the resulting interbeep delay. The shape of this distribution also peaked near second intervals.

The second anomaly is in the command-to-point plot. Notice that when the recognizer is used the distribution has only one peak, while when the recognizer is not used it has two peaks. This can be explained by examining the types of pointing that are performed in KIC. Each pointing action requires either fine alignment for specific coordinates or gross alignment to select existing boxes. When the recognizer is not used the mouse comes from the command menu and then must either be finely or grossly aligned. Fine alignment takes longer than gross alignment therefore two distinct peaks appear in the distributions. When the recognizer is used, the mouse is normally aligned while the command word is being spoken, thus the delay required for alignment is mainly a function of the recognizer latency. This explanation was verified by examining which commands contributed to the two peaks.



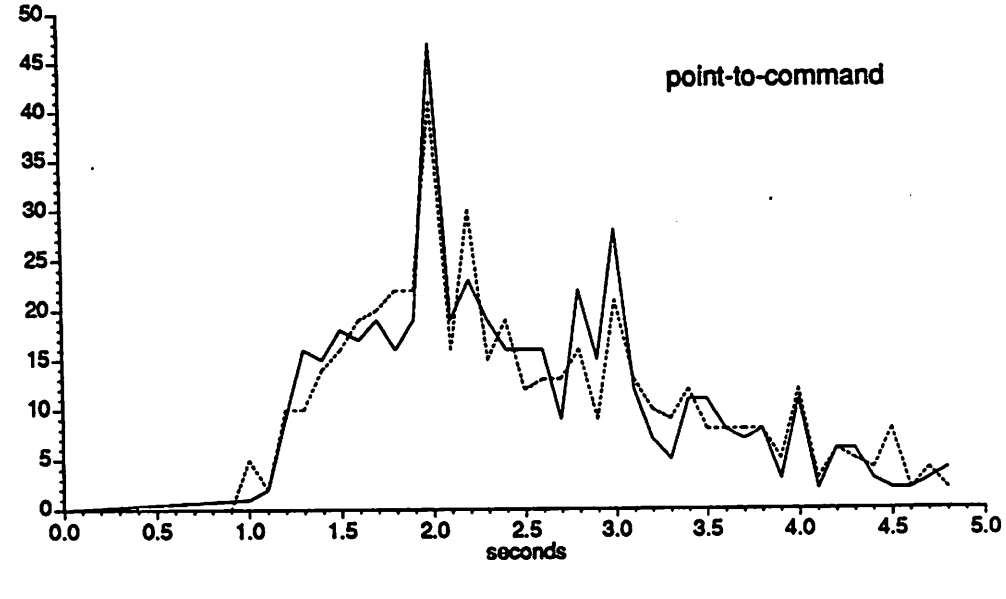


Figure 5.2 - Delay distributions for user steve

The third anomaly is in the command-to-command plot. The distribution without the recognizer is flatter than the distribution with the recognizer. Because each word is equally easy to speak, one would expect that the distribution with the recognizer should have a distinct peak. On the other hand, using the mouse command menu requires the user to move the mouse different amounts depending on the com-

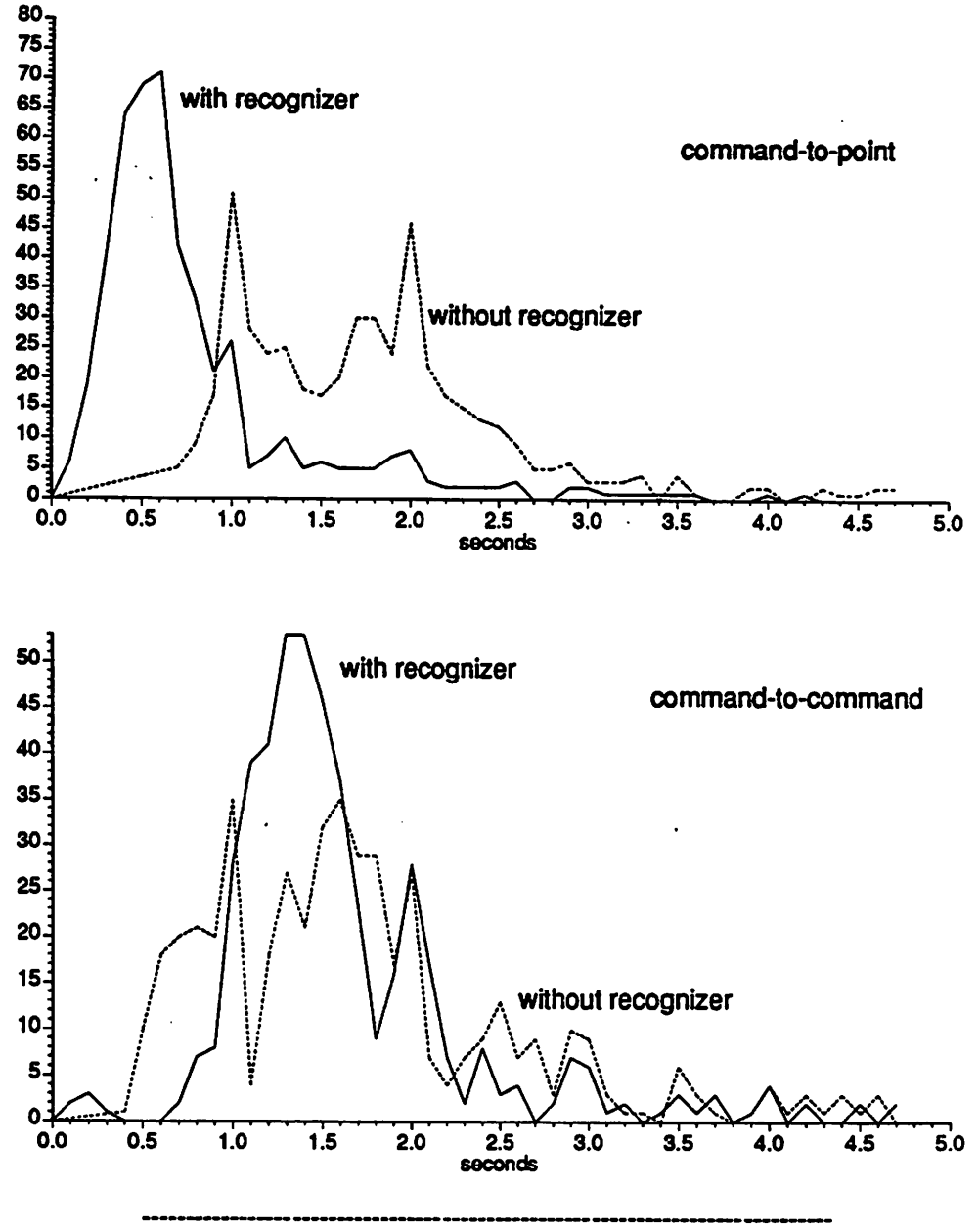


Figure 5.3 - Delay distributions for user steve

mand. In addition, the user might forget where a word is placed in the menu and must search for it. Thus the distribution without the recognizer is flatter than the distribution with the recognizer.

5.2.3. Conclusions

The speed-related performance of recognizers is usually quantified by two measures: words recognized in a second (call this the speed) and latency. The importance of each of these is application dependent. For the KIC editor, an application typical of most menu-based applications, the latency of the recognizer has been demostrated to be more important than the speed. In fact, only 10 to 15 percent of the total interaction delay is due the speed of the recognizer. The rest is due to the interaction between the recognizer and the other input devices. This would indicate that in order to speed the interface the recognizer must have a small and predictable latency, as is the case with the speech recognizer presented here. Another possible solution is to integrate the mouse into the speech recognizer and delay the mouse action until after a word is spoken. This solution would allow mouse actions to overlap speech commands, decreasing the command-to-point delay even further.

The speed gained by using the speech recognizer is not signicantly large to alone justify its use for this application.

5.3. A Long-Term Evaluation

One user, steve, used the recognizer with KIC to layout an entire chip (a video rate 8 bit A/D converter). The chip was part of a Ph.D. project. The recognizer was used a total of 144 hours over one month. Although the error rates were not measured during this time, some of the delays were measured. Figure 5.4 shows the word-to-word delay distribution. Again note the peaks near the second intervals.

The distribution rises sharply at about one second then falls as a decaying exponential. The average value of the distribution is 7.2 seconds indicating that the recognizer is not being used constantly. The decaying shape indicates that the recognizer is used in bursts, that is, the user speaks a few words then waits for the response. In order to speed the interface the curve must be shifted to the left suggesting that a connected word recognizer would speed the interaction. But, the connected word recognizer would only have to handle strings of two or three words, not long phrases or sentences, to be useful for this application.

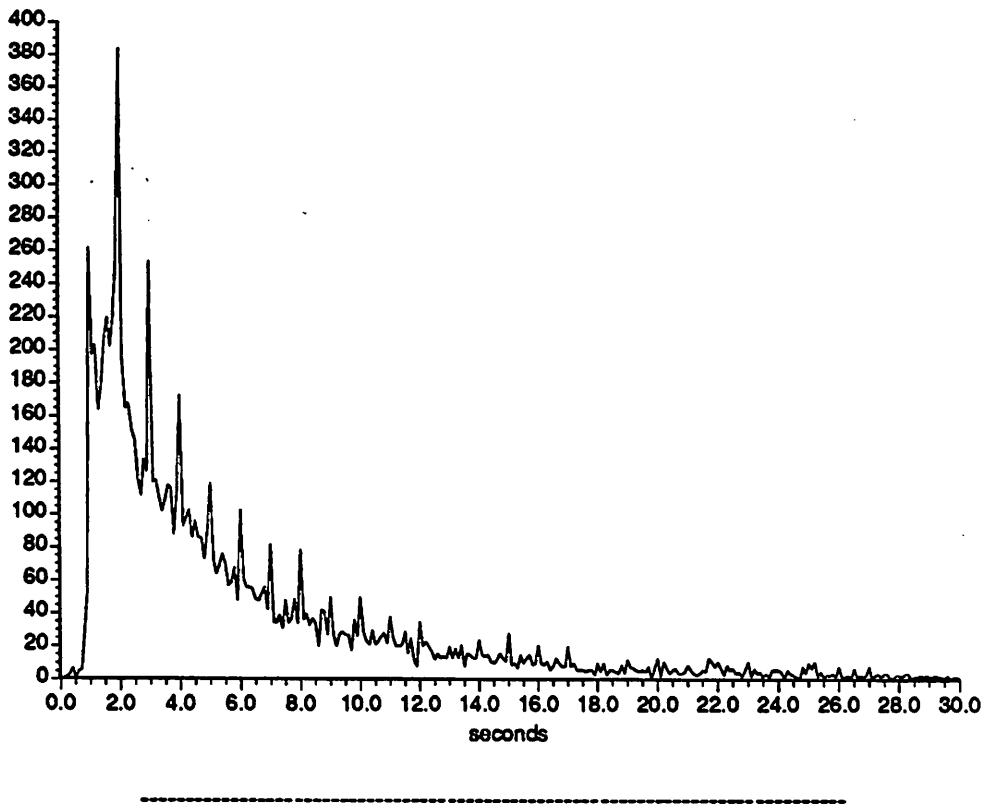


Figure 5.4 - Distribution of long-term word-to-word delays for KIC

The user noted that although the speed gained by using the recognizer was difficult to judge, the strain associated with laying out the circuit was much less when the recognizer was used. This descrease in strain allowed the user to edit the chip longer each day and make fewer mistakes. Overall he evaluated the recognizer's contribution positively and would recommend its use for chip layout.

Chapter 6 - Conclusions

6.1. Conclusions

The predicted boom in speech input devices is normally billed as providing a natural interface to computer systems. In reality, these speech devices must be integrated into the computer system as one input device of many. It is unreasonable to expect all existing non-natural computer interfaces and applications to be completely rewritten specifically for speech input. Instead, the speech recognizer must be integrated into existing systems and applications.

Additional hardware must be added to most computers to recognize speech. This hardware must be flexible enough to recognize at least a few hundred words with high accuracy and little latency. For this project special purpose integrated circuits were designed in a 4μ NMOS process that allowed one thousand words to be recognized in real time. The architecture of the chips is tailored to a version of the dynamic-time-warp algorithm allowing enormous throughput in a relatively old fabrication process. The chips are part of a single board speech recognizer that connects directly to the internal I/O bus of an existing engineering workstation.

Issues not normally considered part of speech recognizer design such as feedback techniques and interface styles must be part of any speech input device for computers. Feedback techniques can be used both to increase recognizer performance and to aid the user. An adaptive training algorithm linked to the display of top recognition candidates has been shown to increase recognition accuracy from 97.5% to 99.5%. Increased recognition is not the only benefit from feedback techniques. By presenting the user with a list of the top candidates for each unknown word, the user can determine how well the recognizer is working and be reminded of the applicable vocabulary for an application. Such feedback must be in the form of words, not just action.

Three interface styles - text entry, menu-based commands, and multi-word commands - have been examined for use with speech input devices. While text entry, also called dictation, is a traditional model for a speech input device, it is a very poor model of how speech recognizers are used in a workstation.

Menu-based and multi-word commands are more typical of computer interface styles. These styles require different recognizer capabilities than text entry, namely, words must be added on a regular basis. These words can be categorized into types for use by a input parser to convert word phrases into computer commands. Associating a word with a particular equivalent string is not sufficient for speech commands.

Traditional performance measures used to evaluate speech recognizers do not work well when applied to speech recognition in the engineering workstation environment. Speaker dependence issues are not as important as being able to add new words on the fly. A usable system must, of course, run in real-time, but it must also provide a small and predictable latency. Connected word recognition would relieve some constraints on the user and might speed the overall interaction a little, but without the corresponding natural language interface, connected speech alone will not make a recognizer more natural than an isolated word recognizer. For an application particularly suited to speech input - a menu-based application - no significant gain in interaction speed was realized when speech input was used. However, users did indicate that speech input was less straining and caused fewer errors than a mouse-based menu interface.

6.2. Future Directions

There are three aspects of the system that need further attention. First, the algorithm should be extended to recognize connected speech. While the hardware supports connect word operation, a real-time algorithm has not been implemented on it (although one has been implemented on past versions of the hardware). Connected speech, if applied with a language model, would probably allow the recognizer to be more robust while still maintaining high accuracy.

The speed with which templates are loaded into the recognizer should be increased so that no noticeable delay occurs when a large vocabulary is needed. Currently the main source of delay is in the network file system provided by SUN. Another source causing delay is synchronous template by template loading. If a "load multiple templates" command were implemented, the system would be faster.

Finally, the Byrne shell should be evaluated and extended to provide more sophisticated speech input for applications. In fact, the Byrne shell grammars should be implemented at a lower level in the recognizer allowing the recognizer to make use of the language model implicit in the Byrne grammar files.

In addition to these changes, addition interface styles need to be examined. In particular, natural language interfaces and application specific interfaces should be developed.

Appendix A - Software

The software described herein can be found relative to the directory "~kavaler" on the SUN workstations with server "zeus".

A.1. Recognizer Board

The software for the recognizer board is written in C with a few a assembly language routines to access 80186 registers and instructions that do not have corresponding C instructions. The software was designed in 4 levels, the lowest three are in the library "cc86mit/lib186". The lowest level routines are in the directories "gen" and "sys". The "gen" routines are the basic C library routines such as "strlen", "strcmp", and "malloc" that are normally supplied with any C compiler. The "sys" routines deal with the special 80186 instructions such as interrupt instructions and I/O instructions. Interrupts must be handled by special routines that first save the state of the processor then call the appropriate C routine. Table A.1 lists the low level routines.

Table A.1 Low Level Library Routines

atoi(p)	atol(p)
calloc(num, size)	cfree(p, num, size)
free(ap)	index(sp, c)
IoOut(port, data)	lblt(to, from, len)
MemFill(paddr, n, d0, d1, ...)	MemIn(paddr)
physaddr(pntr)	qsort(a, n, es, fc)
realloc(p, nbytes)	Refresh()
sbrk(incr)	SetInt(type, paddr)
splow()	splx(status)
strcat(s1, s2)	strcatn(s1, s2, n)
strcmpn(s1, s2, n)	strcpy(s1, s2)
strlen(s)	strncat(s1, s2, n)
strncpy(s1, s2, n)	swab(pf, pt, n)

In order to aid the development of the system, a simple operating system was designed for the board. The lowest levels of the operating system are in the "stdio" and "multibus" directories. The "stdio" directory includes standard I/O procedures such as "putchar", "printf", and "getchar". The "scanf" function is not currently supported, instead the function "getaline" can be used to read. The standard I/O

devices (stdin, stdout) normally talk to the terminal port of the 80186 board. Calling the procedure "sio_new(&sio_multibus)" will change the I/O devices to talk on the multibus to the SUN workstation. The SUN must be running the program "mbhost" to emulate a terminal. The routines in the "multibus" directory handle the low level handshake between the 80186 and the SUN workstation. A device driver in the SUN workstation handles the other side of the handshake. From the 80186 point of view data is sent and received as messages of arbitrary length. On the SUN side, the "read" and "write" commands can be used to receive and send messages to the 80186.

Table A.2 Mid-Level Library Routines

fprintf(iop, fmt, args)	fputc(c, fp)
fputs(s, iop)	getline(st, length)
printf(fmt, args)	putchar(c)
puts(s)	putw(i, iop)
MBGetWord(type)	MBIntOn(mask)

The highest level of the operating system is in two directories: "opsys" and "messages". The basic operating system on the 80186 has a simple multiple cooperating process structure. Processes are created as functions that get called twice. The first time a function is called it returns the size of its run-time stack, the next time it should never return. Processes are scheduled by a round-robin scheduler that must get called explicitly whenever a process is busy-waiting or otherwise idle. The scheduler is called "Dispatch". The scheduler can also be called as the result of an interrupt. The main interprocess communication facility is the queue. Queues are created once with a fixed size entry and a fixed number of entries. The queue get and put routines call "Dispatch" automatically instead of busy-waiting. The procedures lock and unlock can be used to lock a shared resource.

The speech recognizer program is in the directory "mara/mara86". There are two version of the hardware: the first version is for the prototype board in the machine "robert" and all other boards are version two boards. The software for version two boards is in "mara/mara86.v2" which is just a copy of the "mara86" directory with a change to the "makefile" to compile the correct version.

The recognizer program has nine processes:

Table A.3 High-Level Library Routines	
Dispatch()	InitDone()
LinkObject(Objects, object)	Lock(x)
MGGetWord(type)	MesgFlush()
MesgIn(p)	MesgInit()
MesgOut(data, length)	MesgQOut(p)
NewMgInQ(type, q)	ProcKill(Processes)
ProcRet(p)	ProcStart(Processes)
RelinkObject(Objects, func)	SendObject(Objects, obj, len)
SendQueue(q, len)	SetObject(Objects, object, pers)
UnlinkObject(Objects, object)	Unlock(x)
getlq(q, b)	igetq(q)
iputq(q)	killq()
menu()	pfree(p)
putlq(q, b)	resetq(q)
shell(prompt, commands)	

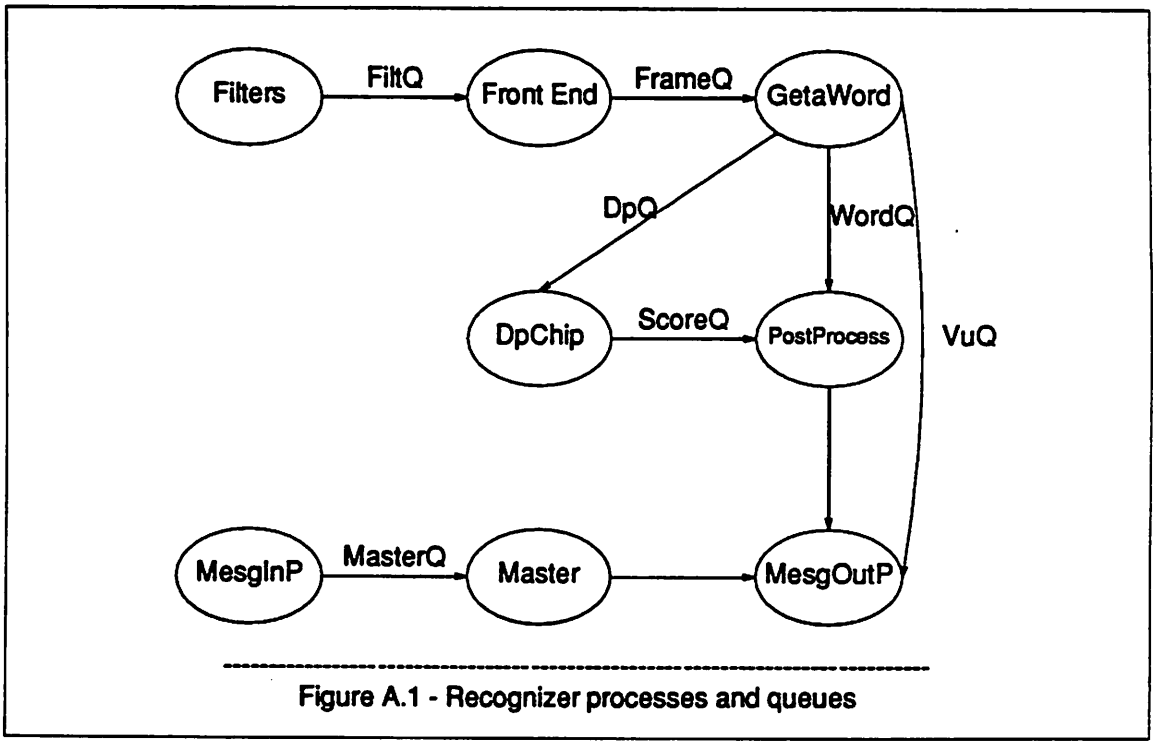


Figure A.1 - Recognizer processes and queues

- Filters - handles the filter interrupt and lowest level filter queue. The output of this routine is the raw filter values as they come out of the filterbank chip. The output queue is "FiltQ".

- FrontEnd - handles the front end processing of "FiltQ" including log conversion, end-point detection, energy normalization, down-sampling, and data compaction. The output of this is the queue

"FrameQ".

- GetaWord - converts the incoming "FrameQ" into word objects. A word object is a large array of frames that is reference-counted to allow words to be passed to other routines inside the program without having to make copies of the word. The GetaWord routine puts a pointer to the word object in the "WordQ", and the incoming frames are also sent to the "DpQ". This routine also sends the VU-meter message to the SUN.
- DpChip - sends frames in the "DpQ" to the dynamic-time-warp chip for processing. This process is triggered by a done interrupt from the chip. The output of the chip is placed in the queue "ScoreQ".
- PostProcess - reads scores from "ScoreQ" and word objects from the "WordQ", finds the best matches, and sends the resulting scores and matches to the SUN.
- Master - handles request from the SUN. Requests are read from the "MasterQ" and include things such as setting parameters, loading words, and reading back words.
- Message In - receives messages from the SUN and places them on the MasterQ.
- Message Out - sends messages to the SUN.
- Debug - used to debug the system.

A.2. Miscellaneous Programs and Changes

Programs can be downloaded from the SUN to the 80186 board in two ways: through the serial port using the program 86ldr or through the multibus using the program mbldr. Both programs take the name of the 80186 file (in .com format) to download. The SUN program mbhost can also be used to download a program. In addition to downloading a program, mbhost hosts the terminal I/O for the 80186 through the multibus.

Along with the standard 80186 program, a special program was written to help debug the hardware. The program is in the directory "spuds/wreck". Two versions of the programs are compiled; the only difference between them is the input and output devices they use. The program "wreck-bus.com" uses the SUN program "mbhost" as its input and output device while "wreck-term.com" uses a standard terminal (i.e. the serial I/O port). The wreck program calls various routines to test the special

purpose chips, the template memory, the local memory and the timers.

Two modification to the standard SUN operating system were required for the recognizer. First, a device driver was added to the kernel and accessed as "/dev/sp0". The source to the driver is in "local/spuds" along with documentation of the hardware and software interface. The driver and the corresponding multibus hardware on the 80186 board can be tested using the programs provided in "spuds/drtest".

The second modification requires a change to the SUN window system. A new window ioctl call was added so that the Mara daemon could determine the exact location of the mouse (i.e. window number and x and y coordinates). The source for modified versions of the window device (SUN versions 1.3, 1.4, and 2.0) is in "local/sunwindowdev_<version>".

A.3. Mara Daemon and Libraries

The software for the Mara Daemon lives in the directory "mara/mara". The daemon is really two programs, one just reads from the 80186 board and formats the data for the main program. Two programs were needed to use the "select" system call for synchronization. The daemon itself is written in a straight forward manner. Connections are made by clients through the UNIX domain socket "/tmp/mara".

The library routines described in the User's Manual (appendix B) are in the directories "mara/libmara" and "mara/libmarawindow".

Appendix B - Mara User's Manual

B.1. Purpose of Speech Recognition

Most papers on speech recognition usually start with a statement like "Speech has long been thought of as the ultimate man-machine interface..." The purpose of this project is to develop a system based on current speech recognition techniques that verifies that premise. It is true that many people think they would like to "talk" to their computer instead of typing, but to date no such system really exists. Current attempts at using speech recognition have been in either highly automated environments such as assembly lines, mass data entry, or in hands busy environments, where the use of a keyboard severely restricts the speed and accuracy of data entry. To date, people are not really "talking" to their computer, they are entering data using their voice.

The Mara system is an attempt to integrate a template based speech recognizer into a single user engineering/programming workstation. In this application there are three components of importance: the computer and its programs, the speech recognizer, and most importantly the computer's user. The Mara system addresses the needs of ALL THREE components in an attempt to build a very high accuracy, easy to use, user environment based on speech input (and hopefully output too).

B.2. Hardware Description

The Mara hardware consists of a Multibus PC board, a backplate with connector, a BNC cable, a pre-amplifier, and a microphone. No other hardware is required, but one can connect two terminals to the PC board if desired (or one terminal and one computer). The microphone should be connected to the pre-amp, which in turn is connected to the PC board through the BNC cable and through the backplate.

The Mara board contains two special purpose integrated circuits developed at Berkeley. One performs spectral analysis, the other computes the Dynamic Time Warp algorithm. There is also an Intel 80186 microprocessor on the board used to manage templates and perform higher levels of the speech recognition algorithm. The board also contains 2 serial channels for terminal communications, a Multibus slave port for high speed communication to a host computer (i.e. a SUN workstation), 128K bytes

of memory local to the 186, 256K bytes of template memory, and 96K bytes of scratch pad memory used in the time warp algorithm. The rest of the board consists of various buffers, latches, and glue logic.

B.3. What the Programmer Sees

B.3.1. System Organization and Standard Usages

The Mara System has five software components:

1. The PC board program - mara86.com
2. The Mara Daemon - mara
3. The Low Level Recognition command library - libmara.a
4. The Standard library - libmara.a
5. Support libraries for various applications - libmarawindow.a

The system was split this way to provide maximum flexibility and still let users share important recognizer resources.

The Mara system was designed to let multiple UNIX processes share the resources on the PC Board (1). Each Unix process can connect to the Mara Daemon (2) and obtain a "virtual recognizer". The connecting process becomes a "user" of the speech recognizer. The Mara Daemon manages the sharing between these processes as well as providing high level support for the recognition system. All virtual recognizers look identical to each user. To ease communications with the Mara Daemon a set of library functions was written that correspond to the lowest level virtual recognizer commands (3). There is also a Standard library (4) that allows any UNIX process to perform common speech recognition functions (i.e. compare two words, average a set of words, file words using a standard dictionary, etc.) locally.

The PC board program is written almost entirely in C. It uses an 8086 cross compiler written at MIT as part of their portable C compiler project. The Mara Daemon and libraries are written entirely in C. The daemon can run under the SunWindows package or on any standard terminal. Of course, Mara

under SunWindows is a much better system.

B.3.1.1. Terms

Before going any further we must define some of the terms used in speech recognition, and some of the structures and abstractions used by my programs. Any word that is in capital letters is the name of a 'C' structure type or a constant. These structures and their corresponding constants and macros can be used by including the file <mara/mara.h> (see Relevant Files).

An 'utterance' is a WORD that is acquired by the system as the result of a person speaking a word. Note that 'WORD' refers to the structure described below, while 'word' refers to what we normally think of as a word.

A 'template' is a WORD that is hopefully a good match to most utterances of a given word. Often a word needs more than one template corresponding to different pronunciations of that word. A template may be an actual utterance of a word, or might be an average over many utterances, or may even be synthesized from allophones.

A '**uname**' (short for 'universal name') is the fundamental low level name given by the Mara Daemon. Since a given word might require more than one template, and since the Mara Daemon may change a word's templates in a given session, a unique identifier not directly associated with a template must be used for each word loaded by each user. This identifier is the **uname**. The **uname** for a word is assigned by the Mara Daemon and is a value between 0 and 999 inclusive. In order to save memory and simplify operations there is only one **uname** per word. Thus two users that load the same word will get the same **uname** for that word.

B.3.1.2. Structures

```
FRAME {
        unsigned char fr_ave;
        unsigned char fr_max;
        unsigned char fr_data[NUMFILTS/2];
};
```

A FRAME is a sample of the speech spectrum. Fr_ave is the average value (0-127) of the spectral components in the speech spectrum. Fr_max is the maximum value (0-127) of the spectral components. Fr_data is the actual spectrum quantized to 4 bits per feature, 16 features (NUMFILTS). The data is coded so that the low nibble of fr_data[0] is the lowest frequency filter, and the high nibble of fr_data[0] is the second lowest, etc.

```
WORD {
    short wd_type;
    short wd_length;
    short wd_flags;
    short wd_elements;
    short wd_ex3;
    FRAME wd_frames[MAXWORDLEN];
};
```

A WORD is a time ordered collection of FRAMEs. Wd_type indicates the condition of the algorithm when this set of frames was acquired (i.e. down-sample threshold, shape of filters, etc.). Wd_length is the number of FRAMEs in the word. Wd_flags contains various flags that indicated information about how this word was formed and how it should be used:

WDF_UNPROMPTED	word was acquired without use of a prompt.
WDF_AVERAGE	word was created by averaging utterances.
WDF_SPEAKERINDEP	word is a speaker independent template.

Wd_elements is the number of words that were used to create the WORD. This field is only applicable when WDF_AVERAGE is set. The function WordElements(word) will return a value of 1 if WDF_AVERAGE is not set, and wd_elements if it is set. The field wd_ex3 is reserved for future expansion. Wd_frames is the time ordered array of FRAMEs. The size of a WORD in bytes can be obtained by using the macro WordSize(word). The size of the header information is the constant WordHSize.

```
HEARING {
        short hr_mgtype;
        short hr_idtype;
        short hr_id;
        short hr_place;
        short hr_x;
        short hr_y;
        short hr_unlen;
        short hr_numscores;
        struct {
                short hr_uname;
                short hr_score;
                short hr_condition;
                char *hr_cdata;
        } hr_data[1];
};
```

A HEARING is the result of comparing a word with all templates. It is sent by the daemon to a user if a word is heard, or as the result of executing a CompareWord function. Hr_mgtype is set to one of the following constants:

HR_ERROR — an error occurred while trying to process this HEARING.

HR_REJECT — this word should be rejected.

HR_OFF — turn off the recognizer now.

HR_ENDOFSENTENCE — end of sentence has occurred.

HR_EVAL — this HEARING contains information about the top hr_numscores words that were trained by this user (EVALMODE).

HR_NORMAL — this HEARING contains information about only the top word that was trained by this user (in data[0]).

Hr_idtype and hr_id together form a unique identifier for the word corresponding to this HEARING. Hr_place is the number of the topmost window under the cursor when the word was spoken. Hr_x and hr_y are the x and y coordinates of the cursor in the screen's coordinate system. Hr_unlen is the length in FRAMEs of the word. Hr_numscores indicates the number of valid score/name pairs in the hr_data field. Each entry consists of four subfields hr_uname, hr_score, hr_condition, and hr_cdata. Hr_data[0] is

the best (lowest score) match of the word with all templates currently trained by the user. Hr_data[1] is the second lowest score, etc. Hr_score is the actual score, and hr_uname is the uname of the word. Hr_condition can be ignored for now. Hr_cdata is the data associated with hr_uname if hr_uname is really a class. The size of a HEARING in bytes can be obtained by using the macro HearingSize(hearing). The size of the header information is the constant HearingHSize.

B.3.1.3. Word Model

At the core of the template sharing mechanism is the 'Word Model'. This abstraction allows one to store and retrieve WORDs (utterances and templates) without knowing the details of the underlying filing system. The model consists of a set of library routines to store and retrieve WORDs, a 'Dictionary' of words and their possible spellings, and a UNIX directory (with sub-directories). If a word is not in the dictionary then the Word Model will create a temporary entry for that word.

A word consists of three pieces of information: a canonical spelling (canspell), a set of other spellings, and a 'template group' name. The dictionary is a set of entries of this kind. A word is referred to by one of its spellings. For example, the word 'six' is referred to by the character string 'six'. Many words have different spellings even though they are pronounced the same way (e.g. 1/one/won; right/write; to/two/too/2 etc.). In order to handle these cases consistently there is concept of canonical spelling (canspell). A word can be referred to by any of its known spellings, and the Mara system will derive the correct canspell (using the Dictionary). All information about a word is associated with its canspell, not a particular spelling. The canspell is really a 'C' structure that contains information about the word. In order to let ALL programs share the same WORDs, each WORD is filed in a particular place on disk. The 'template group' specifies a sub-directory into which this word should be stored. Its use is optional, but can speed operations by shortening the average length of directories.

The 'word model' must be set up by creating a directory "templates" in ones main directory. The "templates" directory should contain the file 'Dictionary', the dictionary of words. The directory should also contain a directory for each template group.

B.3.2. The Virtual Recognizer

A typical process will first connect to the daemon, obtaining a virtual recognizer. The process will then load the set of words particular to its own application. Next the process might enable or disable various options. The process must also inform the daemon when this virtual recognizer should receive HEARINGs. Finally the process tells the recognizer to start recognizing. If the process wishes to add templates, or change options it must first turn the recognizer off, then make the changes, then turn it on again. In order to let the user run an evaluation of the raw speech recognizer each virtual recognizer can be placed in EVALMODE. Normally, the HEARING sent by the daemon to the user is just a single uname (or reject). But a process might want to know the top few candidate words that could have been recognized. The recognizer can be placed in EVALMODE, which will send back a HEARING with the top SCORESNEEDED score/name pairs filled in.

In order to simplify interaction with the Mara Daemon, a library of standard C procedure calls was written. If one uses this library then only one recognizer connection can be established per process. This may change in the future, but I doubt it.

B.3.2.1. Standard Commands

A virtual recognizer is in one of three possible states at all times: disconnected, off, or on. In the disconnected state the daemon does not even know of the existence of the UNIX process that wants to use the recognizer. The 'off' state is entered when a process wants to give commands to its virtual recognizer. When the recognizer is in the 'on' state, a process may not send commands to the daemon. In this state recognition results (HEARING structures) are sent from the daemon to the process indicating that a word was spoken. Transitions between these states are handled by the procedure MaraGotoState(), or by the use of easy to remember macros: (Easy to remember, not to type).

```
OnRecognizer()
OffRecognizer()
ConnectRecognizer()
DisconnectRecognizer()
```

The first thing that a process that wishes to use the recognizer should do is call:

```
ConnectRecognizer(fatal)
    int fatal;
```

If fatal is TRUE then the process will abort (call exit(-1)) if it cannot connect to the daemon. If fatal is FALSE the process will return TRUE if the connection succeeded, FALSE if it didn't succeed.

Next a process will normally load WORDs to be recognized (templates). There are two different ways to load templates.

```
LoadaRawTp(word, flags)
    WORD *word;
    int flags;
```

LoadaRawTp is used if you have a WORD structure that you want to use as a template.

```
LoadaSpelling(spelling, flags)
    char *spelling;
    int flags;
```

LoadaSpelling is used if you want to load all the templates of a given canspell of the given spelling. The latter call allows multiple processes to share the same spellings. If this call is used, only one set of templates for each spelling is loaded into the template memory on the PC board. Thus one is encouraged to use LoadaSpelling whenever possible.

The flags field is set to a bit vector of the following options:

LDF_GLOBAL	this word is a global word (i.e. recognize it regardless of where the mouse points).
LDF_RETAINED	retain templates for this word in template memory even if no users refer to this word.
LDF_NAME	this word is a name for this user. Redirect input to this user if name is spoken. Redirection is active until end of sentence is encountered. A HEARING corresponding to the user name will NOT be sent to the user.

The value returned from both LoadaRawTp and LoadaSpelling is the **uname** of this word, or MERROR if an error occurred and the word could not be loaded. Possible errors include: a global word was already trained by another user, no more template memory, or templates for this word could not be

found or created. A word can be loaded twice, in which case the same uname will be returned both times it is loaded.

If a user wants to delete a word one of the following commands can be used:

```
UnloadaUname(uname)
    int uname;
```

or

```
UnloadaSpelling(spelling)
    char *spelling;
```

The action of these two operations is obvious. In both cases the uname of the deleted word is returned, or MERROR if an error occurred.

Each virtual recognizer has a set of user definable parameters and flags. To set (or get) a parameter one uses:

```
MaraParameter(parameter, value_type, value)
    int parameter;
    int value_type;
    int value;
```

There are two different types of parameters, user and global. Local parameters are specific to one particular user, while global parameters are the same for all users. In general, global parameters should not be changed. Currently the following parameters are defined:

User:

PRM_REJECTTHRESHOLD	rejection threshold
PRM_SCORESNEEDED	maximum number of score/name pairs to return when in EVALMODE

Global:

PRMG_REJECTTHRESHOLD	rejection threshold
PRMG_SILENTGAP	size in frames of maximum intraword gap
PRMG_DOWNSAMPLETHRESH	down-sampling threshold

PRMG_HIGHENERGYTHRESH	minimum peak energy for a word
PRMG_SENTENCEGAP	number of frames of silence between sentences
PRMG_ADAPTRADIUS	determines radius of similarity for adaptive training algorithm

Value_type indicates the type of operation to be performed on the parameter. There are four different possibilities:

PMT_SETTODEFAULT	set parameter to default value
PMT_GETDEFAULT	return default value for this parameter
PMT_GET	return current value of this parameter
PMT_SET	set parameter to specified value.

Instead of using MaraParameter, the user should use one these four convenient macros:

```
SetMaraParameter(parameter, value)
GetMaraParameter(parameter)
SetDefaultMaraParameter(parameter)
GetDefaultMaraParameter(parameter)
```

In addition to user definable parameters, each user has a set of flags that it can set.

FLG_EVALMODE	put virtual recognizer in EVALMODE
FLG_FLASHSPELLING	flash (under SunWindows) the recognized word
FLG_FLASHREJECT	flash (under SunWindows) 'REJECT' if the word is to be rejected.
FLG_SENDEOSENTENCE	send end of sentence HEARINGs if set.
FLG_TRAINONLOAD	when a word is loaded an no templates exist for the word, then one is trained.

The function call is in the form:

```
MaraFlags(flags, value_type, value)
    int flags;
    int value_type;
    int value;
```

Flags is a bit vector of the flags that one wants to change. Value_type is as above for parameters, and

value is a bit vector of new values. Again, the user should use on the four macros:

```
SetMaraFlags(flags, value)
GetMaraFlags(flags)
SetDefaultMaraFlags(flags)
GetDefaultMaraFlags(flags)
```

For example, to clear both flashings use:

```
SetMaraFlags(FLG_FLASHSPELLING|FLG_FLASHREJECT, 0);
```

The ultimate goal of the any speech recognizer is to send a message to the computer that a particular word has just been spoken. In the virtual recognizer system proposed, this task is complicated by the fact that all virtual recognizers are running simultaneously. But a spoken word should only be sent to ONE virtual recognizer. The daemon picks the correct recognizer to get a given "word was spoken" message (HEARING) using the following algorithm:

```
If any user has grab_all set
      then send HEARING to that user
else
if user name was spoken in the recent past
      then send HEARING to that user
else
if the top word is global (LDF_GLOBAL)
      then send HEARING to the user that loaded that word
else
      determine the window that the mouse is in
      send HEARING to the last user associated with that window
```

If Mara is being used outside the window system then the HEARING will be ignored if grab_all is not set, or the top word is not global. A particular window is associated with a user by using the command:

```
AssocWindow(window)
      int window;
```

Window is the number from the device /dev/win#. This number can be obtained by extracting it from the environment variable WINDOW_ME, or, if the window's file descriptor is open (i.e. in a tool), by using the call fdtonumber(fd). Mara maintains a stack of associations for each window. If a user dies the window will revert to its previous owner. I hope that this scheme is sufficient. AssocWindow returns TRUE if it can make the association otherwise it returns FALSE.

Now, after all templates are loaded, and all options are set correctly the user can turn on the recognizer. While on, the recognizer can not load words, set options, or perform any other commands. The macro to turn on the recognizer is:

```
OnRecognizer(grab_all)
    int grab_all;
```

If grab_all is true then the daemon will send the results of ALL words that it hears to this user. Grab_all should only be used to run through test tapes or similar applications. OnRecognizer returns TRUE if the recognizer was turned on, otherwise it returns FALSE.

While the recognizer is on, the user should wait for input from the file descriptor "mara_fd". For example, one might use the select system call with input_bits equal to (1 << mara_fd). Note that the value of mara_fd should never be changed.

If input is available from file descriptor "mara_fd", the user can read the HEARING with the command:

```
HEARING *GetHearing(hearing)
    HEARING *hearing;
```

GetHearing fills the HEARING structure passed to it with the HEARING being sent from the daemon. If hearing is NULL then a new HEARING is created using malloc(). GetHearing returns a pointer to the HEARING, or NULL if an error occurred. The HEARING returned by this call is encoded as explained in the previous section of this manual. Remember, the daemon will only return uname's associated with words that the user has loaded. A HEARING's memory can be freed by calling:

```
FreeHearing(hearing)
    HEARING *hearing;
```

If the HEARING was not created using malloc(), then one should instead call:

```
FreeHearingData(hearing)
    HEARING *hearing;
```

FreeHearingData will free all the cdata sub_fields for re-use.

Finally, if a user wants to turn off the recognizer it should call:

OffRecognizer()

Any words that are spoken from this point until the next OnRecognizer command will be ignored. If a user wishes to disconnect from the recognizer it should call:

DisconnectRecognizer()

If a process dies without turning off and disconnecting then Mara Daemon will delete all of the users loaded templates gracefully. Thus, IT IS NOT NECCESSARY TO DISCONNECT BEFORE EXITING A USER PROGRAM. Another way to disconnect the recognizer is to call:

AbortRecognizer()

This call can be executed at any time.

B.3.2.2. Classes

The above scheme lets a user associate each word with a meaning, but provides no method of sharing meanings among users. Classes let users associate an arbitrary string with a given word. The string is the same for all users, and is remembered from session to session. For example, a file-name class might associate with each word a file name. Thus regardless of which window one is in, when the word is spoken the window can use the file name.

By convention (enforced), all class names start with a "#" character. To load a class one should call LoadaSpelling() with the name of the class instead of the name of a spelling. At that point all words in the given class are active. The **uname** returned by LoadaSpelling is the **uname** of the class. When a HEARING is received, that same **uname** will be in the hr_uname subfield, and hr_cdata will be set to the string.

Members can be added to class by calling:

```
AddMemberToClass(class, member, cdata)
        char *class;
        char *member;
        char *cdata;
```

AddMemberToClass will add the spelling "member" to the class "class", and associate the data "cdata" with this spelling.

```
DeleteMemberFromClass(class, member)
    char *class;
    char *member;
```

This procedure does the obvious thing. Class associations are stored on disk in the file "Classes" in ones templates directory. The file is written at strategic points, and thus should always be up to date. If you wish though, you can call:

```
DumpClasses()
```

and force the file to be up to date.

There are a few more things that are important to know about classes. First, all class members are always loaded in template memory even if no user has loaded that class. This is done to simply things. Thus do not put all words into a class. Second, a word can only be a member of one class at a time. To move a word from one class to another first delete it from the old class, then add it to the new class. Third, a word can be loaded as a class member and at the same time be loaded as a spelling. This is how the "#names" class does its work. The spelling takes precedence over the class membership (for a given user). Fourth, if you have been paying attention then you will realize that the class argument of the DeleteMemberFromClass function is redundant (since each word can only belong to one class). This argument is provided as a check, and must be set correctly.

B.3.2.3. Other Commands

In addition to the above, there are a few more commands that are very useful.

```
WORD *GetaWord(prompt, word)
    char *prompt;
    WORD *word;
```

GetaWord tells the Mara Daemon to print "Please say 'prompt'" on the screen, and wait for the speaker to speak this word. The word that the speaker actual says is then returned in the structure "word". Note that there is no guarantee that the returned WORD corresponds to an actual utterance of the word suggested by the prompt. The WORD structure returned by GetaWord is placed in "word", unless "word" is NULL, in which case a new WORD is created using malloc(). In either case a pointer to the WORD is returned by GetaWord. If NULL is returned then an error occurred.

```
HEARING *CompareWord(word, hearing)
    WORD *word;
    HEARING *hearing;
```

CompareWord uses the processing power of the Mara PC board to compare the WORD "word" with all words currently loaded. The resulting HEARING is processed according to the options set, and returned in hearing. If hearing is NULL then a new HEARING is created. If NULL is returned then an error occurred. The user should be cautious to set all the appropriate options before calling CompareWord. Also, CompareWord (and GetaWord) can only be used while the recognizer is off.

```
TpVerify()
```

TpVerify is useless to all people but those repairing Mara PC boards. It attempts to verify that the templates on the PC board have not been destroyed due to a synchronization bug in the hardware.

```
FlashString(string, x, y)
    char *string;
    int x, y;
```

FlashString will flash "string" on the screen associated with the Mara Daemon. X and y specify the coordinates where the string is to be flashed (screen coordinate system). It is suggested that the x and y from the HEARING that generated the event be used. This call can be used when the recognizer is either on or off. This call is included so that a user doesn't have to include the SunWindow library in order to flash messages.

```
FlashUname(uname, x, y)
    int uname;
    int x, y;
```

FlashUname is similar to FlashString except that it is passed a uname instead of a string. If the uname has a value less than zero than "REJECT" will be flashed on the screen. Otherwise the spelling of the canspell corresponding to the uname is flashed.

B.3.2.4. Command List

Mara Virtual Recognizer Commands		
Type	Meaning	Function
WORD * HEARING *		GetaWord(char *prompt; WORD *word) CompareWord(WORD *w; HEARING *hearing)
int int int int	uname uname uname uname	LoadaRawTp(WORD *word; int flags) UnloadaUname(int uname) LoadaSpelling(char *spelling; int flags) UnloadaSpelling(char *spelling)
int int int int int	param param param param param	MaraParameter(int parameter; int value_type; int value) SetMaraParameter(parameter; value) GetMaraParameter(parameter) SetDefaultMaraParameter(parameter) GetDefaultMaraParameter(parameter)
int int int int int	flags flags flags flags flags	MaraFlags(int flags; int value_type; int value) SetMaraFlags(flags, value) GetMaraFlags(flags) SetDefaultMaraFlags(flags) GetDefaultMaraFlags(flags)
int int int int	T/F T/F T/F T/F	ConnectRecognizer(int fatal) OnRecognizer(int grab_all) OffRecognizer() DisconnectRecognizer() AbortRecognizer()
int int	uname uname	AddMemberToClass(char *class, *member, *cdata) DeleteMemberFromClass(char *class, *member)
int	T/F	AssocWindow(int window)
		FlashString(char *string; int x, y) FlashUname(int uname; int x, y) TpVerify()

Mara Virtual Recognizer Responses		
Type	Meaning	Function
HEARING *		GetHearing(HEARING *hearing)

Explanation of Types and Meanings	
xxx *	Pointer to an xxx structure. Error in indicated by NULL.
uname	an integer between 0 and 999 (inclusive). A value less than 0 indicates an error condition.
T/F	TRUE or FALSE
param	The value of the parameter.
flags	The value of all flags specified.

B.3.3. Standard Libraries

In order to save time regenerating the same code for each speech application, a set of standard

library routines was written. These routines perform all the word model functions, word and frame distance computations, averaging algorithms, training algorithms, and a few miscellaneous functions.

B.3.3.1. Word Model

Before one can start to use any of the word model routines the dictionary must be set up. This is done by calling:

```
SetupDicitonary()
```

Once the dictionary is set up it need never be set up again. If entries are added to the dictionary file it is automatically reread and the data-base updated. The dictionary resides in the templates directory under ones main directory. The exact full path to the templates directory can be obtained by calling:

```
char *MaraDirectory()
```

This routine will normally return the expanded ~/templates, but if the environment variable MARA_TEMPLATES is set then ~/$MARA_TEMPLATES is returned.

The dictionary file is named "Dictionary" and it contains lines in the form:

<canspell> <template group> <other spelling> <other spelling> .... ;

Once an entry is made to the dictionary it should not be deleted.

```
CANSPELL *CanspellOfaSpelling(spelling)
    char *spelling;
```

CanspellOfaSpelling returns pointer to the CANSPELL structure for the given spelling. If the word is not in the dictionary then a new canspell is created. This new canspell will have only one spelling. The template group for this canspell is determined from the first letter in the spelling. The pointer returned is unique for each CANSPELL, and WILL NOT CHANGE even if the dictionary is changed. The canspell is intended to be the ultimate reference for all words. The spelling of this canspell can be obtained using the macro:

```
char *SpellingOfaCanspell(canspell)
    CANSPELL *canspell;
```

A spelling, as you remember, is a string suitable for printing.

The word model specifies that each WORD is uniquely indentified by three pieces of information: canspell, type and appendix. The type is one of:

WT_TEMPLATE	This WORD should be used as a template.
WT_UTTERANCE	This WORD is just an utterance.
WT_CONSIDERATION	This WORD may become a template soon.

Appendicies start from 1 and are assigned in increasing order. Words are stored in files on the disk. The filename for a given word can be obtained by calling

```
char *FilenameOfaWord(filename, canspell, type, appendix)
    char *filename;
    CANSPELL *canspell;
    char type;
    int appendix;
```

Filename is the place in memory that will get filled with the appropriate file name. The function returns a pointer to the filename.

```
char *DirectoryOfaCanspell(directory, canspell)
    char *directory;
    CANSPELL *canspell;
```

DirectoryOfACanspell fills "directory" with the string for the UNIX directory that canspell can be found in.

```
int ExpandWordFilename(fn, sp, splen, type, appendix)
    char *fn;
    char *sp;
    int *splen;
    char *type;
    int *appendix;
```

ExpandWordFilename gets passed a filename (fn) and breaks it into its three components sp (spelling), type, and appendix. Splen is the length of the spelling "sp". The function returns TRUE if "fn" is a valid filename. Note: all leading path parts of the filename are deleted so that the resulting spelling "sp" should be usable as an actual spelling.

To read a word from a file the following function should be used:

```
WORD *ReadaWord(filename, word)
    char *filename;
    WORD *word;
```

In this case, if "word" is NULL then a new word is created using malloc(), otherwise the WORD is placed in "word". ReadaWord return NULL if an error occurred (like the WORD doesn't exist), otherwise it returns a pointer to the WORD. A WORD can be written using:

```
int WriteaWord(filename, word)
    char *filename;
    WORD *word;
```

This function returns FALSE if an error occurred, otherwise it returns TRUE.

The functions FilenameOfAWord(), DirectoryOfACanspell(), ExpandWordFilename(), ReadaWord(), and WriteaWord() are low level routines and should not be used by users (although you can use them if you want). There are four higher level functions that users should use. They are:

```
WORD **GetWordFiles(cs, type, lower, upper, wordlist, appendflag)
    CANSPELL *cs;
    int type, lower, upper;
    WORD **wordlist;
    int appendflag;

int PutWordFiles(cs, type, lower, upper, wordlist, overwrite)
    CANSPELL *cs;
    int type, lower, upper;
    WORD **wordlist;
    int overwrite;

WORD **GetBlockWordFiles(cs, type, wordlist, appendflag)
    CANSPELL *cs;
    int type;
    WORD **wordlist;
    int appendflag;

int PutBlockWordFiles(cs, type, wordlist, overwrite)
    CANSPELL *cs;
    int type;
    WORD **wordlist;
    int overwrite;
```

GetWordFiles returns a list of new WORDs (created using malloc()) in wordlist. The WORDs put in the list are those that correspond to the canspell "cs", with type "type", and an appendix between "lower" and "upper". If appendflag is TRUE then wordlist is appended to (i.e. new words are added to the end of the

list); otherwise wordlist is assumed to be empty. Remember that wordlist is NULL terminated. That is, the last WORD in wordlist is actually a NULL pointer.

PutWordFiles does the opposite operation, writing WORDs to disk. Again "cs" is the canspell and "type" specifies the type. Words are written starting with appendix equal to "lower" and going up to "upper". If overwrite is TRUE then WORDs will be written over existing WORDs on disk. Normally overwrite should be FALSE.

Certain types are stored, by convention, in blocks. This means that the all appendicies are consecutive and start from one. It turns out that the GetWordFiles and PutWordFiles are too inefficient under certain conditions. Currently WT_TEMPLATE and WT_CONSIDERATION are stored as block word files. The command GetBlockWordFiles will get words starting with appendix equal to one until a word is not found. PutBlockWordFiles will write words starting either from one (overwrite is TRUE), or from the highest appendix for this word (overwrite is FALSE). After all words in the list are written, PutBlockWordFiles will remove all disk files that correspond to larger appendicies for that word.

B.3.3.2. Classes

In order to let users manipulate class information the following routines are provided:

```
IsClassName(class)
    char class;
```

IsClassName will return true if the argument passed to it is a valid class name.

```
FILE *OpenClassesFile(mode)
    char *mode;
```

OpenClassesFile will return a FILE * as the result of opening the "Classes" file with mode string "mode". The mode string is the same as is found in fopen(). If the file cannot be opened then NULL is returned.

B.3.3.3. Distances

At the heart of all speech recognition systems is the definition of the distance between two WORDs. This distance should have the property that WORDs that correspond to the same word should have small distances, and different words should have large distances. Mara uses the squared Euclidean

distance between 4 bit quantized spectral estimates. The FRAME to FRAME distance measure that can be computed using:

```
int Fr2Fr(frame1, frame2)
    FRAME *frame1, *frame2;
```

Fr2Fr returns a number between 0 and 255 indicating the spectral distance between frame1 and frame2. The WORD to WORD distance is much more complicated. The distance measure implemented in Mara is a dynamic time warp algorithm. The algorithm has no slope constraints, adjustments windows, or pruning. A 1-1-1 weighting is used, with a normalization factor equal to the maximum length of the two WORDs. The results of the WORD to WORD comparison is a 15 bit value, and for good matches will generally be less than 30:

```
int Word2Word(word1, word2)
    WORD *word1, *word2;
```

The algorithm is symmetric, so the order of word1 and word2 is unimportant. Another useful function is:

```
int Word2WordPath(word1, word2, path1, path2, pathlength)
    WORD *word1, *word2;
    int **path1, **path2;
    int *pathlength;
```

This function returns the same thing as Word2Word. In addition to computing the distance, Word2WordPath also computes the dynamic time warp path. The path is a set of pairs ((*path1)[i], (*path2)[i]) that correspond to the FRAME to FRAME comparison along the path with the smallest distance. The total length of the path is put into *pathlength. The paths are created using malloc() and must be freed after they are used. This is the recommended usage of Word2WordPath:

```
int *w1path, *w2path, pathlen;
WORD *word1, *word2;
int score;

...
score = Word2WordPath(word1, word2, &w1path, &w2path, &pathlen);
...
free(w1path);
free(w2path);
```

B.3.3.4. Averaging

In addition to computing distances, another common speech recognition function is WORD averaging.

```
WORD *AverageWords(wc, wordlist, wo, threshold, wordcount)
    WORD *wc;
    WORD **wordlist;
    WORD *wo;
    int threshold;
    int *wordcount;
```

AverageWords averages the WORDs in "wordlist", using center WORD "wc". The averaged WORD is placed in "wo." A new WORD is created using malloc() if "wo" is NULL. The only WORDs included in the average are those whose distance from the center WORD is less than or equal to threshold. If threshold is less than zero then all WORDs are used. AverageWords returns a pointer to the average WORD, and sets wordcount to the number of WORDs that were used to form the average. The average of a set of PROMPTED and UNPROMPTED WORDs results in an UNPROMPTED WORD.

AverageWords uses two routines to interpret the FRAME data structure. These are:

```
ExpandFrame(frame, intarray)
    FRAME *frame;
    int  *intarray;
```

and

```
CompressFrame(frame, intarray)
    FRAME *frame;
    int  *intarray;
```

ExpandFrame takes as input a FRAME frame, and expands it into an array of integers intarray. Intarray[0] is fr_ave, intarray[1] is fr_max, intarray[2] is the low nibble of fr_data[0] etc. CompressFrame performs the inverse operation.

In addition to a normal average, the library also contains a routine to compute a weighted average. The functions :

```
WordWeight(word)
    WORD *word;
```

and

```
WordWeightNoClip(word)
    WORD *word;
```

return the weightings for the given words. WordWeight will clip weights at some small constant. The weighted average of a set of words can be computed by calling:

```
WORD *WeightedAverageWords(wc, wordlist, wo, threshold, wordcount)
    WORD *wc;
    WORD **wordlist;
    WORD *wo;
    int threshold;
    int *wordcount;
```

The parameters for this routine are identical to those for AverageWords.

B.3.3.5. Trainer

A big part of any speech recognition system is the training algorithm. The Mara system currently uses a trainer called Rickie. Rickie takes as input a WORD list and uses a UWA clustering algorithm to find possible key WORDs. These WORDs are then averaged with their neighbors to form templates. Outliers are excluded. Rickie can be used by calling:

```
WORD **Rickie(wordlist, clminsize, clthresh, force, numtemplates)
    WORD **wordlist;
    int clminsize;
    int clthresh;
    int force;
    int *numtemplates;
```

Rickie takes the WORD list "wordlist", and attempts to form clusters of maximum spread "clthresh". These clusters are then averaged around their center to form templates. Rickie returns a WORD list numtemplates long (NULL terminated). Two other parameters are under user control: clminsize and force. Clminsize specifies the minimum size (in number of WORDs) of a cluster for it to be considered a valid template. Anything smaller is considered an outlier. If force is true then Rickie will always create at least one template. Its algorithm specifies that the threshold be incremented by one until at least one valid template is formed.

B.3.3.6. Miscellaneous

```
ListLength(list)
    int **list;
```

ListLength returns the length of a list (number of non-NULL elements) for a wordlist or any other list of pointers to things.

```
FreeListMembers(list)
    int **list;
```

FreeListMembers calls free on all the members of the specified list.

B.3.4. High Level Recognizer Commands

In order to supply a set of recognizer commands closer to the users needs, a set of higher level recognizer calls was developed. These calls can be used in most applications.

B.3.4.1. Basic Strategies

The purpose of the high level package is to provide support for different types of voice commands. All high level commands can be mixed with lower level commands. To initialize this package the routine:

```
mara_initialize();
```

should be called before any other Mara calls. The package supports different commands through the use of two structures:

```
UNAMEITEM {
    char ui_type;
    char ui_flags;
    char *ui_data;
};

MARAMENUITEM {
    char *mm_data;
    short mm_flags;
    UNAMEITEM mm_ui;
};
```

These structures are defined in <mara/marawindow.h>. There are one thousand (MAXUNAMES) UNAMEITEM structures in the system (one per uname). Each structure indicates how to respond when

a uname is spoken. Ui_flags is set to a vector of:

UI_FLASH	flash the spelling of this word when spoken
UI_ICONACTIVE	command should be recognized when tool is iconic

Ui_type is the type of operation to be performed if this word is recognized. Current value are:

UI_TTYSW	1
UI_PROCEDURE	2
UI_TOOLPROC	3
UI_OPTSW_BOOL	4
UI_OPTSW_COMMAND	5
UI_OPTSW_ENUM	6
UI_OPTSW_TEXT	7
UI_USENAME	8
UI_USEVOCABFILE	9

The largest allowable value is currently 40. New types can be created by calling:

```
mara_settype(type, procedure, free_procedure)
    int type;
    int (*procedure)();
    int (*free_procedure)();
```

Mara_settype informs the system to call "procedure" when a word of type "type" is spoken. "Procedure" is called with:

```
<procedure>(ui, hearing)
    UNAMEITEM *ui;
    HEARING *hearing;
```

"Free_procedure" is called with the just the UNAMEITEM parameter when a uname is redefined. This procedure should free the instance data associated with the given type, that is, data created using malloc(). The routine

```
mara_freedata(ui)
    UNAMEITEM *ui;
```

is the generic memory freeing routine. It calls free(ui->ui_data). If free_procedure is NULL, then no freeing routine is called.

The MARAMENUITEM structure contains spelling and flags fields (see LoadaSpelling above) and the UNAMEITEM to be associated with this spelling. A single MARAMENUITEM can be attached (loaded) by calling:

```
mara_attachsingle(maramenu)
        MARAMENUITEM *maramenu;
```

An entire menu of items can be attached by calling:

```
mara_attachmenu(maramenu)
        MARAMENUITEM *maramenu;
```

A menu is an array of MARAMENUITEMs that is terminated with a NULL in the mm_spelling field.

In addition to supplying a set of attach calls, the basic package supplies a standard routine to handle Mara input. When a process wishes to read a Mara event and process it, it should call:

```
mara_handleinput(iconic)
        int iconic;
```

Iconic is TRUE if the window is currently iconic (see UI_ICONACTIVE flag above). The mara_handleinput routine reads a HEARING from mara_fd and performs the appropriate actions (calls the procedure associated with the ui_type field). In addition, if the Mara Daemon dies then AbortRecognizer is called. After AbortRecognizer is called, a user routine can be called by using:

```
mara_setabort(procedure)
        int (*procedure)();
```

B.3.4.2. Tool Support

A library of calls was developed for use when one designs a new tool. These calls are adapted to be very similar to their corresponding SunWindow calls. There is currently one restriction placed on tools: only one tool is allowed per UNIX process. A tool that wishes to use a recognizer should call

```
struct tool *mara_tool_create(name, flags, normalrect, icon, mm)
        char *name;
        short flags;
        struct rect *normalrect;
        struct icon *icon;
        MARAMENUITEM *mm;
```

instead of tool_create. All the parameters are the same as the normal tool_create, except for the addition of the mm parameter. Mm should be the mara menu associated the the tool window (or NULL if no menu is wanted). The normal menu is:

```
MARAMENUITEM *mara_toolmenu[];
```

Mara_tool_create calls mara_initialize. After the tool is set up other menus can be attached and other Mara subwindows can be established. Explicit names may be given to tools on their command line by specifying the "-n <name>" flag. The call to extract these names from the command line is:

```
mara_tool_args(argc, argv)
    int *argc;
    char *argv[];
```

Here argc is a pointer to the real argc. The name and flag are removed from the command line and the resulting space is compacted. The automatic naming feature can be activated by calling:

```
mara_tool_activatenameclass()
```

This function loads the class "#names", a class of window names. When one of the names is spoken, and the mouse is in this users window, then that name is trained as the name of the window (after a mouse confirmation). The namestripe of the window is updated with the name of the window. The cdata associated with the #names class is the same as the spelling. When all menus have been set up, and all subwindows have be created one should call:

```
mara_tool_install(tool)
    struct tool *tool;
```

Mara_tool_install replaces the normal tool_install call. At this point the tool should be installed, and the recognizer should be on. Mara_tool_install changes the "selected" routine associated with the tool so that input is handled automatically. One need never call mara_handleinput.

The tool naming feature places the all of its names in the namestripe above the tool. Thus, in order to change the namestripe one should call:

```
mara_tool_newnamestripe(name, display)
    char *name;
    int display;
```

Display should be true if the windows are installed, otherwise it should be false. Name is the new namestripe.

B.3.4.3. Tty Subwindow Support

Support for the tty subwindow consists of supplying a method of attaching to each word a string that is "typed" when that word is spoken. In this case, ui_type should be UI_TTYSW, and the ui_data should be the string to be "typed". In order to let people make better use of the support one can use:

```
mara_ttysw_loadmenufile(filename)
        char *filename;
```

This routine will load the words and strings found in the specified file. The file contains lines of the form:

<spelling>:<flags>:<string>

Flags is a string that can be used to set various flags. Current flags are:

g - LDF_GLOBAL
r - LDF_RETAINED
f - UI_FLASH
i - UI_ICONACTIVE

String is encoded so that \n, \r, \e (ESCape) etc. can be used. Also \[A-Z] can be used as control characters. If the spelling is a class, then the cdata associated with the word will be inserted in the string when a "%s" is encountered. "%%" should be used as "%". One can also load files from the command line using:

```
mara_ttysw_args(argc, argv)
        int *argc;
        char *argv[];
```

Where argc is a pointer to the actual argc. Files that get loaded are those that occur after a "-v" flag. More than one vocabulary file can be specified.

The class "#vocabs" can be used to associate vocabulary files with windows. To activate the #vocabs class one should call:

```
mara_ttysw_activatevocabsclass(windowfd)
        int windowfd;
```

Windowfd should be the windowfd of the tool or the ttysw subwindow. The #vocabs class contains associations of words to vocabulary files (full path names). When one of these words is spoken, the corresponding vocabulary file is loaded (after mouse confirmation).

B.3.4.4. Option Subwindow Support

The include file for the option subwindow is <mara/optionsw.h>. Three options types are currently supported: boolean, command, and enumerated. The boolean option can be set up by calling

```
caddr_t mara_optsw_bool(optsw, label, init, notify, spelling, flags)
        caddr_t     optsw;
        struct      typed_pair *label;
        int         init;
        int         (*notify)();
        char        *spelling;
        int         flags;
```

instead of the normal optsw_bool() routine. This call is identical to optsw_bool except that it also attachs the word "spelling" to toggle the value. Flags specifies the ui_flags field associated with this word.

```
caddr_t mara_optsw_command(optsw, label, notify, spelling, flags)
        caddr_t     optsw;
        struct      typed_pair *label;
        int         (*notify)();
        char        *spelling;
        int         flags;
```

Mara_optsw_command performs the same function for command options. For this case flags should be set to UI_FLASH since I could not find a way to get the label to flash otherwise.

```
caddr_t mara_optsw_enum(optsw, label, choices, init, notify, spellings, flags)
        caddr_t     optsw;
        struct      typed_pair *label;
        struct      typed_pair *choices;
        int         init;
        int         (*notify)();
        char        *spellings[];
        int         flags;
```

Mara_optsw_enum performs the similar function for enumerated types. In this case spellings is a list of spellings.

```
caddr_t mara_optsw_text(optsw, label, default_value, flags, notify, class, clflags)
        caddr_t     optsw;
        struct      typed_pair *label;
        char        *default_value;
        int         flags;
        int         (*notify)();
        char        *class;
        int         clflags;
```

Mara_optsw_text is used for text items. In this case a class is used instead of a spelling. Thus, different text items can be associated with different classes (i.e. one for files, one for directories, one for commands). I am sure that this type of item will have to be expanded on in the future to include spellings and numbers.

B.3.5. Relevant Files

In order to use the libraries above one must include the following header files:

```
#include <mara/mara.h>          /* Low Level Commands */
#include <mara/marawindow.h>    /* Basic Level and Tool Commands */
#include <mara/optionsw.h>      /* Option Subwindow Commands */
```

The libraries themselves can be linked into your programs by including:

```
-lmarawindow -lmara
```

on the C compile line.

B.4. What the Computer User Sees

B.4.1. Programs

B.4.1.1. The Mara Daemon

The Mara Daemon is called "mara" and lives in /usr/local. The Daemon will download the program that runs on the PC Board. This program is in /usr/local/mara86.com. When the Daemon is started it will normally print all sorts of stuff on the console. This stuff can be suppressed by using the -e command line option. For example, the Daemon should normally be called with the line:

```
/usr/local/mara /usr/local/mara86.com -e
```

The daemon can run in a few different terminal environments (SunWindows, terminals with 25th lines and terminals without 25th lines). It will automatically determine the environment to use when started, but can be overridden with the -d <display> option. Here <display> is one of: sun, h19, h19-a, ansi, or vt100.

The daemon, when started, will read from the file ".mara_setup" in ones home directory and use it to set the default user flags and parameters. The file should have lines in the form:

<option> <value>

Current options are:

flashreject	off/on [normally on]
flashspelling	off/on [normally off]
trainonload	off/on [normally on]
rejectthreshold	[normally 27]
silentgap	Word to word gap time in 10ms units [normally 18]
downsamplethreshold	[normally 6]
highenergythreshold	[normally 80]
sentencegap	in 10ms units [normally 150]
adaptionradius	in percent of rejection threshold [normally 120]

The daemon can be killed by either selecting "Quit" in its "tool manager" menu, or by running the program:

/usr/local/killmara

If the daemon dies for some strange reason then the recognizer must be reset. This is done by the program resetmara.

/usr/local/resetmara

B.4.1.2. Suntools

In the directory /usr/local/maratool is a copy of the standard suntools programs that have been recompiled to include voice capabilities. To use these programs one should include in ones path variable the entry /usr/local/maratool BEFORE the entry /usr/suntool. To change the recognizer from its default position on the screen, one should set the environment variable WINDOW_MARALOCATION:

setenv WINDOW_MARALOCATION 980:156:164:168:1082:156:64:64:0

This line will start the daemon up on the right side of the screen (for SUN 120/170 displays), about two icon heights from the top of the screen.

The suntools programs sets up three phrases: "new shell", "new graphics", and "redisplay all". If one of these phrases is spoken the appropriate action is performed.

The shelltool and gfxtool programs are set up to read voice files for their tty subwindows. These tools activate the #names and #vocabs classes.

B.4.1.3. Other Programs

The program "maraparams" provides the user with a way to set the global recognition parameters (and print them). The program will print the values of all the parameters if no arguments are provided. Arguments are in to form:

-<flag> <value>

possible flags are:

r	Reject threshold
w	Word to word gap time (in 10ms units)
d	Down-sample threshold
h	High Energy (in word) threshold
s	Intersentence gap (in 10ms units)
a	Adaption Radius (in percent of rejection threshold)

Values should be numbers. There are 3 special values:

d	use default value
p	print current value
P	print out default value

The program "class" is used to add/delete/look at class relationships. If called with no arguments it will dump all the class associtates to the screen in a readable format (really is a pretty printer of the Classes file). If arguments are supplied then the first argument is the name of a class (with the "#" character). The rest of the arguments apply to this class. There are two different types of requests, delete and add. A delete is indicated by a "-" as the first character of an argument. The rest of the argument is taken as the spelling to be deleted from the class. The absence of a "-" character indicates an add command. In

this case the two arguments are needed, first the spelling, then the cdata string. One can put as many requests as one wants on a single class commmand.

Since the description above is pretty unclear I will give a few examples.

class #vocabs v_i_words ~/vocabs/vi.v -test_words

This request will add the spelling "v_i_words" to the class #vocabs and associate the string ~/vocabs/vi.v (the ~ should be expanded by the c-shell). It will also delete the spelling "test_words" from the class #vocabs.

class #names clyde clyde -v_i_window bonnie bonnie

This request will add the spellings clyde and bonnie to the #names class, and delete v_i_window from this same class. Note that in the case of the #names class the spelling and its string should be the same (or else all h___ will break loose).

B.4.2. Window Environment

This section deals with Mara in the SunWindows environment.

B.4.2.1. Getting Started

Getting started is probably the most difficult part of using Mara. First one must obtain a working PC Board with all the neccessary connectors, the pre-amp and a microphone. After the board is installed and all the cables are set up the hardest part is over. Your UNIX system must have the SPUDS board driver installed, and should have the device /dev/sp0. Assuming all this is set up beforehand, setting up Mara itself is relatively easy.

Run the program "setupmara <dictionary>" where <dictionary> is a Dictionary file. A reasonable start-up dictionary is supplied in /usr/local/mara.Dictionary.

setupmara /usr/local/mara.Dictionary

This shell script will create all the neccessary files (templates directory, Dictionary, sub-directories). Next, edit your .cshrc file to include the directory /usr/local/maratool BEFORE /usr/suntool in your path variable. You are now ready to use Mara.

At this point it is best to test to make sure everything is set up correctly. Start the Mara Daemon using the command:

/usr/local/mara /usr/local/mara86.com

You should see the message "Spuds started", then a new window should appear with the title "Mara". Turn on the pre-amp (also flip the switch to "filters on"), set the filters to 6K, and turn the gain to about 80% of full scale. Now say a word or two and look at the Mara window. You should see a bar go across the window. This is a VU-meter. If you don't see this then something is wrong. To kill the daemon move the mouse to the title line, press the rightmost mouse button, and select "Quit" on the menu.

Now it is time to use Mara for real. First kill the daemon, then exit the suntools environment and log out. Log back in, and start up the suntools environment. You should be prompted to say some words. Say them. Now point to a window and say one of the words you just trained. Isn't that interesting, it should do what you tell it.

B.4.2.2. What You See

Look at the Mara window and will notice two different things going on. First, at the top of window is a grey bar that goes from left to right whenever you speak a word. This is a VU-meter, and it is used to help set the gain on the pre-amplifier. The gain should be adjusted so that words typically cause the bar to go to 75% of full scale. You might have to adjust the position of the microphone too. If the bar is all the way to the right then the system will clip the signal, if the bar is below 50% of full scale then the word will be ignored. You might also notice a black bar at the left hand side of the window. This bar indicates the background noise level. Normally it should be either invisible or just barely visible. If the background noise increases then the bar will start creeping to the right. Notice also that the grey bar goes away after about a second and a half. This is triggered by an "end of sentence" detector. Above the gray portion of the VU-meter is a small black line. This line indicates the long term average of word peaks.

The rest of the Mara window contains words and scores. When Mara hears a word it tries to find the trained words that are most similar to the spoken word. The smaller the score the more similar the two words. For each spoken word, the top candidate words are put in the Mara window along with their

scores. The word with the smallest score is the one that was assumed to have been spoken, unless its score is above a "rejection threshold". If you want to see fewer/more candidate words one need only decrease/increase the height of the Mara window using the "Stretch" menu command. Only global words, and words attached to the window under the mouse are displayed in the window. You might also notice that if you move the mouse over the Mara window, the word under the mouse will go into reverse video. This indicates that the word is selected. Mara allows one to do strange and interesting things with selected words.

B.4.2.3. Adaptive Training

If a word is trained poorly then its scores will be high. In fact its score might be so high that it exceeds the rejection threshold most of the time. In other cases the score might be so high that other trained words are a better match to the spoken word. In either case the word can be retrained first saying that word, then selecting the word in the Mara window and clicking the left mouse button. When the adapation algorithm is done, the window will go blank and Mara is ready for action again. Usually it takes from 1 to 4 seconds to retrain a word.

If the word that you spoke does not appear in the window then you cannot select it. Unfortunately this will often occur when words are very poorly trained (i.e. when you train a word as a "door slam" by accident). All hope is not lost in this case. If you type characters on the keyboard while the mouse is in the Mara window then the words that appear in the window will be those words that match the typed characters so far. For example, if the word "clock" was trained as some strange noise that can not be retrieved, then one need only type "c", "l", ... until the word clock comes into the window, then select it normally (move the mouse over the word), and click the left button to adapt. The backspace character and kill (control-X) characters will do the right thing.

B.4.2.4. Assigning Window Names

Each window can have one or more names associated with it. These names allow one to redirect voice commands to windows other than the window directly under the mouse. A name is associated with

a window by first entering the name in the #names class.

class #names <window name> <window name>

Notice that the window name must be given TWICE. At this point you will be asked to say the name. Once the name is in the #names class it need not be entered again. Next one should move the mouse into the window that is to be given the name. Now say the name. The system will ask you if this name is OK. If all went right click the left button. (If you changed you mind then hit the right button.) The name should now be assigned to that window. Notice the namestripe now contains the name of the window.

Words can be redirected to this window by first speaking the name, then the words. A silence of one and a half seconds "clears" the name, sending words to their usual window. The command associated with each word will execute as soon as that word is spoken.

B.4.2.5. Creating and Modifying TTY Vocabulary Files

Tty vocabulary files form a first attempt at retrofitting most common programs to voice input. The idea of a tty vocabulary file is to associate an arbitrary string with each word. This string is then sent to a process as though it was typed on the keyboard. Tty vocabulary files can only work within "tty subwindows", (i.e. SunWindow graphics programs that use the keyboard cannot use tty-vocabulary files). In order to keep things consistent one should create a directory ~/vocabs for vocabulary files. Generally, a different vocabulary file should be used for each application or set of applications. For a given application the vocabulary file should be named ~/vocabs/<application>.v, For example ~/vocabs/unix.v might be general purpose unix commands, and ~/vocabs/edit.v might be editor commands.

There are three steps in creating a new vocabulary file. First one must create a file ~/vocabs/<application>.v in the format specified in section 3.4.3. Next one should add the vocabulary file to the class of all vocabulary files. This is done by typing the command:

class #vocabs <application>_words ~/vocabs/<application>.v

Mara will ask you to say "<application>_words", and you should do so. Finally, one should say "<application>_words" in the window that wants to load this application.

When changing a vocabulary file or applying it to a new window, one need only say "<application>_words" in the appropriate window and the new vocabulary file will be loaded.

B.4.2.6. Some Other Conventions to Make Life Simple

In order to make life simple for yourself (and me), I suggest the following convention: the spelling of a word or word phrase should be in the form:

<word>_<word>_ ....

For cases where the word is one letter, the letter should be spoken as one would normally speak that letter. For example, the unix command "ls" should be spelled as "l_s" and spoken as "ell ess".

B.4.2.7. Some Common Mistakes and How to Fix Them

Of course, the recognizer is not perfect, and many bug are yet to be found. If something catastrophic occurs one should just start from the beginning. The most common non-fatal error is not saying the word that you are asked to say. For example, when training a word for the first time the Mara Daemon put up a box on the screen asking you to say a particular word. Just then you cough, and the daemon says "Thank You". Well, you have now trained the word to sound like a cough. What you want to do is retrain the word so that it sounds like it should sound. The way to do this is to say the word, then move the mouse into the Mara window, then type the word in this window. As soon as you see the word to be retrained in the window, move the mouse over the word, it should go into reverse video. Now press the left mouse button. The word is now retrained (unless you coughed again). A common source of cough-like noise is the keyboard on the SUN (models 120 and 170).

References

1. Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," *Computer*, vol. 14, no. 4, pp. 25-34, April 1981.

2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.

3. V.M. Velichko and N.G. Zagoruyko, "Automatic Recognition of 200 Words," *International Journal of Man-Machine Studies*, vol. 2, p. 223, June 1970.

4. F. Itakura, "Minimum Prediction Residual Principle Applied to Speech Recognition," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-23, pp. 67-72, Feb. 1975.

5. George M. White and Richard B. Neely, "Speech Recognition Experiments with Linear Prediction, Bandpass Filtering, and Dynamic Programming," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-24, pp. 183-188, April 1976.

6. Hiroaki Sakoe and Seibi Chiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-26, pp. 43-49, Feb. 1978.

7. James K. Baker, "The DRAGON System -- an Overview," *Proc. IEEE Symposium on Speech Recognition*, pp. 22-26, 1974.

8. F. Jelinek, "Continuous Speech Recognition by Statictical Methods," *Proceedings of the IEEE*, vol. 64, no. 4, April 1976.

9. Stephen L. Moshier, "Talker Independent Speech Recognition in Commercial Environments," *Fiftieth Meeting of the Acoustic Society of America*, June 1979.

10. J.K. Baker, J.D. Baker, R. Roth, and P.G. Bamburg, "Cost-Effective Speech Processing," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 9.7.1-4, Spring 1984.

11. Hy Murveit and R.W. Brodersen, "An Integrated-Circuit-Based Speech Recognition System," *IEEE Transactions on Acoustics Speech and Signal Processing*, to be published.

12. B. S. Atal and S. Hanauer, "Speech Analysis and Synthesis by Linear Prediction of the Speech Wave," *Journal of the Acoustic Society of America*, vol. 50, pp. 637-655, Aug. 1971.

13. M. R. Sambur and L. R. Rabiner, "An Algorithm for Determining the Endpoints of Isolated Utterances," *The Bell System Technical Journal*, vol. 54, pp. 297-315, Feb. 1975.

14. Eric Davies, Endpoint Detection of Speech for Real Time Isolated-Word Recognition, University of California, Berkeley, June 1983. M.S. Thesis

15. Jean-Luc Gauvain and J. Mariani, "Evaluation of Time Compression for Connected Word Recognition," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, p. 35.10.1, Spring 1984.

16. L. R. Rabiner and J. G. Wilpon, "Considerations in Applying Clustering Techniques to Speaker-Independent Word Recognition," *Journal of the Acoustic Society of America*, vol. 66, pp. 663-673, Sept. 1979.

17. G. R. Doddington and T. B. Schalk, "Speech Recognition: Turning Theory to Practice," *IEEE Spectrum*, pp. 26-32, Sept. 1981.

18. S. Magar, E. Caudel, and A. Leigh, "A Microcomputer with Digital Signal Processing Capability," *ISSCC Technical Digest*, pp. 32-33, Feb 1982.

19. Chris Terman, *Distribution Tape for C Cross Compiler*, M.I.T., Cambridge, Mass., 1984.

20. Peter Ruetz, S.P. Pope, B. Solberg, and R.W. Brodersen, "Computer Generation of Digitial Filter Banks," *Proceedings of the IEEE International Solid-State Circuits Conference*, pp. 20-21, Feb. 1984.

21. Peter Ruetz, S. Pope, and R.W. Brodersen, "Computer Generation of Digital Filter Banks," *IEEE Transactions on CAD in Circuits and Systems*, April 1986.

22. *MOSIS User's Manual*, U.S.C. Information Sciences Institute, Marina Del Rey, California, 1984.

23. Yoshiaki Kuraishi, K. Nakayama, and K. Miyadera, "A Single-Chip 20-Channel Speech Spectrum Analyzer," *Proceedings of the IEEE International Solid-State Circuits Conference*, pp. 112-113, Feb. 1984.

24. Amnon Aliphas, W.F. Ganong, P. Stonestrom, and Dan Perkins, "High Resolution Digital Filter Chip," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 224-227, Spring 1985.

25. F. Itakura and S. Saito, "Analysis synthesis telephone based upon the maximum likelihood method," *Conference Record 6th International Congress on Acoustics*, Tokyo, Japan, 1968.

26. Motorola Inc., *Motorola Product Catalog*, 1983.

27. Lee Dusek, T.B. Schalk, and M. McMahan, "Voice Recognition Joins Speech on Programmable Board," *Electronics*, pp. 128-132, April 21, 1983.

28. Harvey F. Silverman, "Some Architectural Concepts for Speech Recognition," *Proceedings International Conference on Computer Design: VLSI in Computers*, pp. 741-744, Oct. 1985.

29. Masao Watari, H. Sakoe, S. Chiba, H. Ishizuka, Y. Kawakami, and T. Iwata, "A DP-Matching LSI for Speech Recognition," *NEC Research and Development*, no. 70, pp. 71-78, July 1983.

30. M.K. Brown, R. Thorkildsen, Y.H. Oh, and S.S. Ali, "The DTWP: An LPC Based Dynamic Time Warping Processor for Isolated Word Recognition," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 25B.5.1-4, Spring 1984.

31. Joel Feldman, S. Gaverick, F. M. Rhodes, and J. Mann, "A Wafer Scale Integration Systolic Processor for Connected Word Recognition," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 25B.4.1-4, Spring 1984.

32. Neil Weste, D.J. Burr, and B.D. Ackland, "Dynamic Time Warp Pattern Matching Using an Integrated Multiprocessing Array," *IEEE Transactions on Computers*, vol. C-32, no. 8, Aug. 1983.

33. David Ritchie, *The Computer Pioneers*, Simon and Schuster, New York, NY, 1986.

34. Bruce T. Lowerre, "Dynamic Speaker Adaptation in the Harpy Speech Recognition System," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 788-790, Spring 1977.

35. Bruce Lowerre, The Harpy Speech Recognition System, Department of Computer Science, Carnegie-Mellon University, April 1976. Ph.D. Thesis

36. John Shore and D. Burton, "Discrete Utterance Speech Recognition Without Time Normalization," *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 907-910, Spring 1982.

37. Richard Stern and Moshe Lasry, "Dynamic Speaker Adaptation for Isolated Letter Recognition Using MAP Estimation," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 734-737, Spring 1983.

38. Ken Keller and A. R. Newton, "KIC2: A Low-Cost Interactive Editor for Integrated Circuit Design," *Digest of Papers of IEEE CompCon 82*, pp. 302-304, Feb 1982.

39. John D. Gould, J. Conti, and T. Hovanyecz, "Composing Letters with a Simulated Listening Typewriter," *Communications of the ACM*, vol. 26, no. 4, pp. 295-308, April 1983.

40. Speech Recognition Group (Headed by F. Jelinek), "A Real-Time Isolated-Word Speech Recognition System for Dictation Transcription," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 858-861, Spring 1985.

41. L. R. Bahl, S. K. Das, P. V. de Souza, F. Jelinek, S. Katz, R. L. Mercer, and M. A. Picheny, "Some Experiments with Large-Vocabulary Isolated-Word Sentence Recognition," *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, pp. 26.5.1-26.5.2, Spring 1984.

42. *WordStar User's Manual*, MicroPro Inc., San Rafael, 1979.

43. Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, Sept. 1981.

44. Richard M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in *Interactive Programming Environments*, ed. David R. Barstow, H. Shrobe, E. Sandewall, pp. 300-325, McGraw-Hill, New York, NY, 1984.

45. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.