Speech Recognition 6

6.1 OVERVIEW

This chapter describes the basic framework for speech recognition using ADSP-2100 Family Digital Signal Processors. Although there are many techniques available for speech recognition, this chapter focuses on a single LPC-based technique. This technique takes advantage of the flexible architecture, computational power, and integration of these processors. It also takes full advantage of the family's development tools, which support a modular design that can be easily tested and quickly modified. The modular code design lets you customize the code efficiently for each individual application. For this reason, the programming examples in this chapter have not been optimized since the memory space and speed requirements are specific for each system.

Because of advances in speech recognition and processing, you can design systems that users control through speech. Today, systems and applications that have limited vocabularies are available with a recognition accuracy nearing 100%. The increasing speed and integration of digital signal processors make portable speech processing and recognition units possible. State of the art DSPs have serial ports, substantial memory, and analog interfaces on a single chip, letting you design single-chip solutions for many speech processing applications.

Speech recognition research and development has several goals. Simplifying the interface between user and machine is one major goal. Just as many users consider the mouse an improvement to the user interface on a personal computer, machine speech recognition and understanding has the potential to greatly simplify the way people work with machines. Examples of this emerging technology include dialing telephones and controlling consumer electronics through voice-activation. As voice input and output become further integrated into the everyday machines, many advances will be possible.

6 Speech Recognition

Analog Devices is at the forefront of this emerging technology. With the powerful ADSP-2100 family of DSPs, ADI has asserted its leadership and commitment to this field. For example, the ADSP-21msp50 is a complete system for speech processing. It contains an analog interface, two serial ports, one parallel communication port, expansive on-chip memory, and the superior signal processing architecture of Analog Devices Digital Signal Processors.

6.2 SPEECH RECOGNITION SYSTEMS

Speech recognition systems fall into two categories:

- *Speaker dependent systems* that are used (and often trained) by one person
- *Speaker independent systems* that can be used by anyone

Regardless of the type of system, the theory behind speech recognition is relatively simple. First, the DSP acquires an input word and compares it to a library of stored words. Then, the DSP selects the library word that most closely matches the unknown input word. The selected word is the recognition result. Systems that follow this model have two distinct phases: *training phase* and *recognition phase*.

To help you understand the processes used to develop the speech recognition system implemented in this chapter, this section also briefly describes the theory of voice production and modeling.

6.2.1 Voice Production & Modeling

You can separate human speech production into two distinct sections: sound production and sound shaping. *Sound production* is caused by air passing across the vocal chords (as in "a", "e", and "o") or from a constriction in the vocal tract (as in "sss", "p", or "sh"). Sound production using the vocal chords is called *voiced speech*; *unvoiced speech* is produced by the tongue, lips, teeth, and mouth. In signal processing terminology, sound production is called *excitation*.

Sound shaping is a combination of the vocal tract, the placement of the tongue, lips, teeth, and the nasal passages. For each fundamental sound, or phoneme, of English, the shape of the vocal tract is somewhat different, leading to a different sound. In signal processing terminology, sound shaping is called *filtering*.

Speech Recognition 6

An efficient method of modeling human speech is to separate the speech into its components: an excitation (the sound production) and a filter (the sound shaping). When you need to compress speech for transmission, each part can be efficiently coded and transmitted independently. The coded parameters can then be decoded and synthesized to reconstruct the original speech.

In most speech processing applications, the two parts of speech have an equal importance. For speech recognition, however, they do not. The excitation changes drastically from person to person, and it changes according to the speaker's gender, physical and emotional state. The sound shaping, or filtering, is less sensitive to these factors. For this reason, in a basic speech recognition system, you only need to consider the filter.

A robust and efficient method exists for estimating the sound shaping filter. Called linear predictive coding, or LPC, it estimates the filter characteristics, or the spectral envelope of the sound shaping. By using only the LPC generated coefficients, redundant and unnecessary information is removed from the speech signal, leaving just the essential information for speech recognition. For a more detailed explanation of LPC, refer to *Chapter 10 of Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1.*

Since many different sounds are strung together to form a single word, many sets of LPC filter coefficients are necessary to represent the word. A series of coefficient sets is stored to represent the sound-shaping filter at each particular place the word is sampled. This is possible because speech is a slowly-varying signal. If the speech is processed in short enough time slices, or frames, the sound-shaping filter is approximately constant for the duration of that frame. A series of LPC coefficient sets generated from a series of frames then represents the word, with each frame representing a time-slice of the speech.

A word is stored as a series of frames, with each time-slice of speech represented as a feature vector, using a set of LPC coefficients. In an isolated word system, the beginning and ending points of the word can be detected automatically, therefore only the word itself is captured and stored.

You can build a recognition library from these captured words. Each word to be recognized is stored in a library. For speaker-independent systems, multiple copies of each word may be stored to represent different ways of saying the same word. Once the library is built, the system training is complete, and the task of recognition can begin.

6.2.2 Training Phase

When you train a system to recognize words, you first create a library of stored words. The training phase changes depending on the type of speech recognition system. The system compares the input words against this library to find the closest match.

In a speaker dependent system, ideally the user and trainer are the same person. In this situation, these systems offer the best performance because the input words will be fairly consistent. Also, the recognition library can be relatively small because of the limited number of speech samples required to recognize the input words. Because of accents, dialects, and other variations in speech, the performance of speaker-dependent systems degrades when one person trains the system and another person uses it.

Speaker independent systems are usually trained with speech from many people. This process can be more involved than training speaker-dependent systems because you need more speech samples, (perhaps several hundred, or a thousand samples for each word) to train the system. Speaker independent systems typically require larger memories to hold the larger library.

Although the number of required samples may vary depending on the type of speech recognition system, fundamentally the training process remains the same. Figure 6.1 shows a functional block diagram of the training phase of the speech recognition system implemented in this chapter.

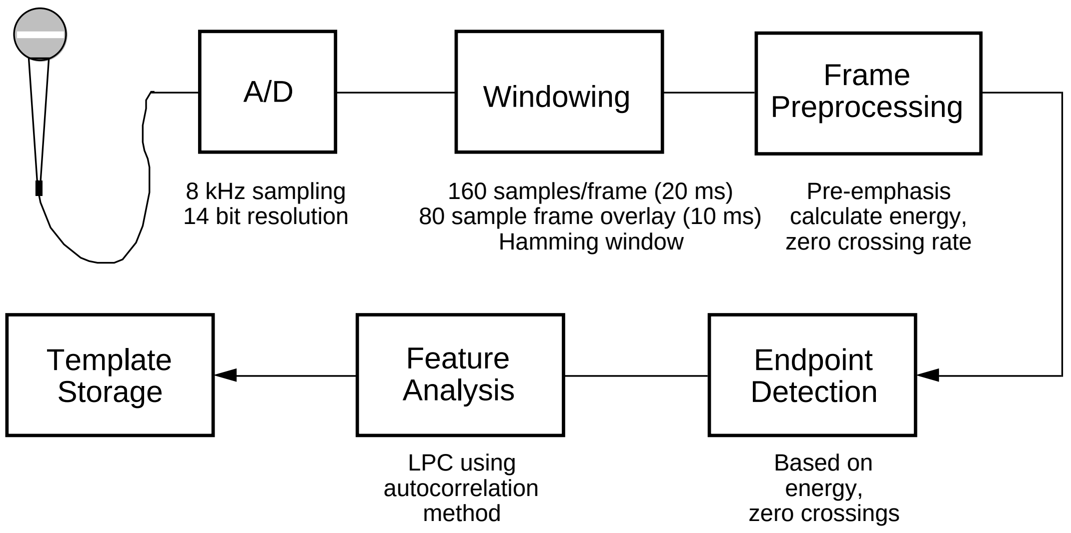

Figure 6.1 Speech Training System Block Diagram

6.2.3 Recognition Phase

Figure 6.2 shows a block diagram of the recognition phase. Notice that the two phases of the speech recognition system share the same word acquisition functions.

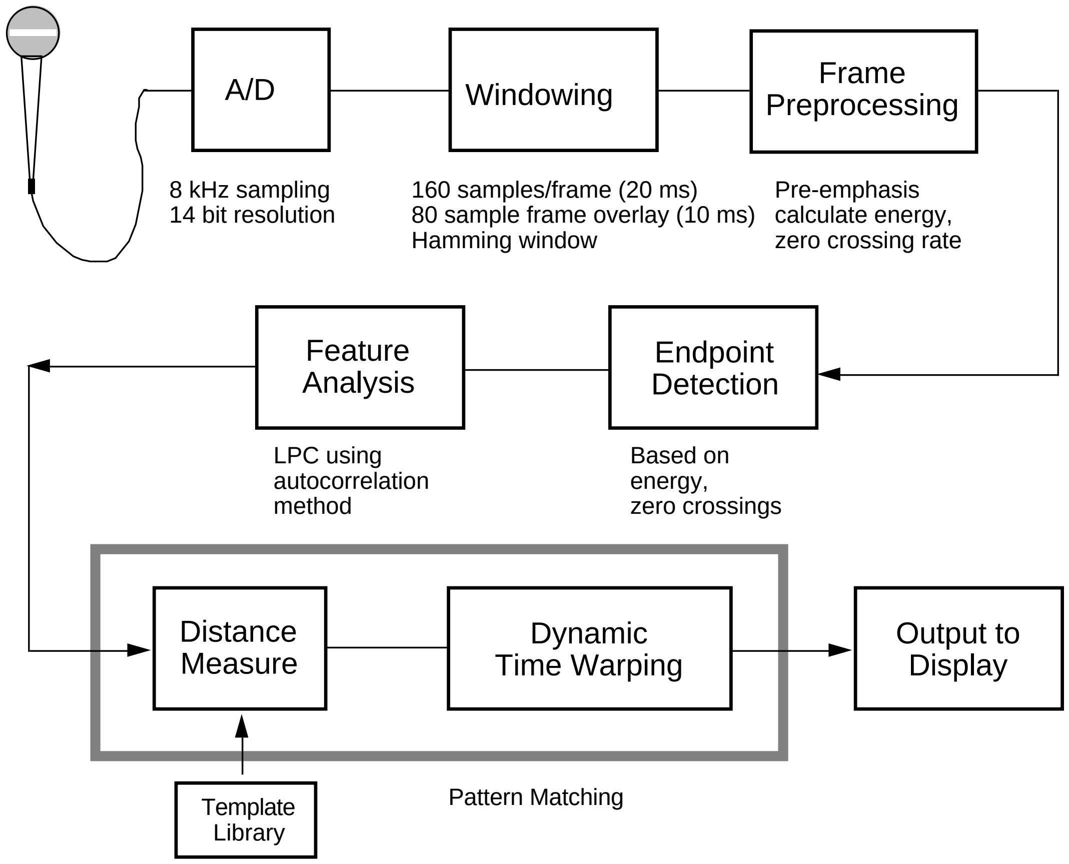

Figure 6.2 Speech Recognition System Block Diagram

During speech recognition, the DSP compares an unknown input word to a library of stored words. Then, for the recognition result, it selects the library word that is most similar to the unknown word. The method implemented in this chapter is a template-based, dynamic time warping (DTW) system. Each library word, stored as a series of feature vectors containing LPC-derived coefficients, is referred to as a template. Since the time you take to completely utter a word changes, dynamic time warping aligns the time axes of the unknown word to a library template. By lengthening or shortening sections of the unknown word, the system attains a “best fit” between words. With DTW, different durations for words have little effect on the recognition accuracy. Dynamic time warping is described in Section 6.3.2.3.

The unknown word and the library word are represented as a series of feature vectors. To compare the words, a measure of similarity between the words is necessary. At the most basic level, the system must measure the similarity between two feature vectors. This is referred to as a distance, or distortion, measure. Many distortion measures are proposed and evaluated in the published literature. Two of the most popular distortion measures are the *Itakura log-likelihood ratio* and the *bandpass cepstral distortion measure.*

During the recognition stage, the distortion measure is integrated into the dynamic time warping routine. The “best fit” between the unknown word and a library word is calculated; the system compares the unknown input word to each library word in turn. The system maintains a recognition word score for each library word. For a single template-per-word system (usually speaker-dependent systems), typically, the system chooses the lowest score as the recognition result. For speaker-independent systems where more than one template-per-word is stored, the lowest scores for each library word are averaged. This results in an average recognition word score for each library word. The system still selects the lowest score as the recognition result.

6.3 SOFTWARE IMPLEMENTATION

This section describes the software implementation of the speech recognition system; it is divided into the following three sections that correspond to the organization of the listing that accompany the text.

The software implementation is divided into the following sections that correspond to the organization of the program examples.

- Word Acquisition and Analysis
- Word Recognition
- Main Shell Routines

6.3.1 Word Acquisition & Analysis

This section describes the functions necessary for word acquisition. These functions are divided into a receive shell, frame analysis, word endpoint detection and coefficient conversion. Each of these functions is contained in a separate subroutine.

The input and output of data samples is interrupt-driven using serial port 0. The interrupt routine is in the main shell. The data is sampled at 8 kHz, and is sectioned into 20 ms frames (160 samples). Each frame overlaps the previous frame by 10 ms (80 samples). This leads to 100 frames per second. As currently configured, a word can not exceed one second.

6.3.1.1 Receive Shell

The subroutine *get_word* in the receive shell is called when a word is acquired. This routine returns a word with I0 pointing to the starting vector of the word, and AX0 holding a count of the number of vectors in the word.

After the software initializes the necessary variables, pointers, and buffers, the receive shell enables data acquisition. A circular buffer of 240 locations is used as the sample input buffer. Each frame of samples is 160 samples long, with consecutive frames overlapping by 80 samples. The code at `code_1_loop` counts the samples. The first time through this loop, the loop exits after a full frame of 160 samples is acquired. After the first iteration, it exits after 80 samples. The result is an 80 sample overlap for consecutive frames.

The 160 samples are copied into a separate frame buffer for processing, since the frame analysis destroys the input data. A frame pointer points to the beginning of each frame in the 240 location circular input buffer. Next, the software analyzes the frame and converts the coefficients. (Sections 6.3.1.2, *Frame Analysis* and 6.3.1.4, *Coefficient Conversion* describe these processes.) The resulting coefficients are written into the feature vector buffer at the current location.

Before word acquisition is complete, the system must detect the word’s endpoints (see Section 6.3.1.3, *Endpoint Detection*). Several options exist based on the results from this subroutine call. If the system detects the word’s start, or possible start, the vector pointer is advanced and the vector count is incremented. The system compares the length of the word to a maximum length, and, if this maximum is exceeded, forces the word to end. If the word does not exceed the maximum length, the system acquires additional frames and the repeats the process.

If the system detects the word’s end, it stops sampling data. It pads the beginning of the word with the five previous feature vectors to insert noise before the word. Then, the routine returns.

If the system fails to detect the beginning or end of a word, the vector pointer and count are reset. The feature vector is written into a circular start buffer, and can be used to pad the beginning of the next word. The code then jumps to the start, and acquires more frames.

At several places in the shell, code is included for conditional assembly. If you use this code, the first four features of the feature vector are the energy, change in energy, zero crossing rate (ZCR), and change in zero crossing rate. For most applications, this information is not necessary. If your application requires this information, the included code adds processing to the receive shell. When the end of the word is detected, the energy values are scaled based on the maximum value determined. Then, the change in energy and the change in ZCR values are determined for each feature vector.

6.3.1.2 Frame Analysis

The subroutine analyzes the frames with an LPC analysis that uses auto correlation and the Schur recursion. It requires pointers to the data frame and the output buffer as inputs. The routine returns eight reflection coefficients.

The subroutine calculates the energy of the frame after scaling and offset filtering the coefficients. A sum of the magnitude of the energy measure is used. Before summing, the magnitudes are scaled to prevent overflows. The subroutine also calculates the zero crossing rate. One zero crossing occurs each time the input data changes sign and passes beyond the noise threshold. Each crossing increases the zero crossing rate by 205. Using this value, a 4000 Hz input results in a zero crossing rate of 32595, taking advantage of the full data width.

Pre-emphasis takes place after these calculations, and uses a coefficient of -28180. Next, a Hamming window is multiplied by the frame of data. Finally, the auto correlation and the Schur recursion complete the frame analysis.

6.3.1.3 Endpoint Detection

The endpoint detector is a variation of an endpoint detector proposed by Rabiner (see the references for the source). It is based on the energy (sum of magnitudes) and the zero crossing rate (ZCR) of each frame of input data. The subroutine determines the word's endpoints by comparing these values to several thresholds. These thresholds adapt to steady background noise levels. Several flags are returned from this routine to indicate a definite word start, a possible word start, or the end of a word.

There are two types of thresholds for the energy and ZCR, possible thresholds and word start (WS) thresholds. *Possible thresholds* are set just above the background noise levels, and for this reason, they may be exceeded occasionally by spurious background noise. The *word start thresholds* are set relatively high so they are exceeded only when the system is sure a word is being spoken. Setting WS thresholds to high, however, causes the detector to miss some softly spoken words. It may be necessary to experiment with threshold levels to achieve the best results.

There are two additional thresholds. The *minimum word length threshold* is set to the minimum number of frames per word. This should be long enough to avoid isolating background noise spikes, but not too long. The *threshold time* is the length of silence that must be detected before a word end is determined. This is necessary to allow silence in the middle of words (especially preceding stops, like “t” or “p”).

When searching for the start of a word, the algorithm first compares the frame energy and zero crossing rate to the WS thresholds. If the frame energy or ZCR exceeds the threshold, the word start flag is asserted, and the system starts storing frames. If the threshold is not exceeded, the possible thresholds are compared. If the frame energy or ZCR exceeds the possible thresholds, the possible start flag is set and the system starts storing frames. For this to be considered the actual start of a word, however, the WS thresholds must be exceeded before the frame energy and ZCR fall below the possible thresholds.

Once a word is determined, the algorithm searches for the end of the word. The subroutine finds the end of the word when the energy and ZCR fall below the possible thresholds for longer than the threshold time. When this happens, the word end flag is set.

6.3.1.4 Coefficient Conversion

The LPC analysis performed on the incoming frames of data produces eight reflection coefficients for each 160 samples of input speech. While this data compression is outstanding, for recognition purposes, the reflection coefficients are not the best features to represent the speech. There are two widely used representations; the predictor coefficients of the LPC analysis and the cepstral coefficients. The *predictor coefficients* are the parameters of the all-pole filter that is being modeled. These predictor coefficients are often referred to as α_k. The *cepstral coefficients* are parameters of the impulse response of the log power spectrum of the input speech. In this case, the cepstral coefficients are solved for recursively from the predictor coefficients, and are referred to as c_k.

Coefficient conversion immediately follows the frame analysis, but happens before feature vector storage. The conversion module is separated into several subroutine calls, each with a specific function. The implementation is in floating-point, with a 16-bit mantissa and 16-bit exponent. This method lets you ignore scaling issues, speeding the code development. The floating-point routines are adapted from the routines in Chapter 3, *Floating-Point Arithmetic*, in *Digital Signal Processing Using the ADSP-2100 Family*, Volume 1, and are called throughout the module.

The first subroutine called from the conversion shell, `k_to_alpha`, converts the fixed-point reflection coefficients (k's) to the floating-point predictor coefficients (α_k). The conversion is accomplished using a the following recursion

$$a_i^{(i)} = k_i$$
$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)} \quad 1 \le j \le i-1$$

which is solved recursively for i = 1, 2, ..., p. The final results are found from

$$a_j = a_j^{(p)} \quad 1 \le j \le p$$

For the current system, p = 8.

Two buffers are used to store the temporary results of the recursion, one for even values of i and one for odd values. These buffers alternate as the input buffer and the result buffer at each stage of the recursion, until the final result is contained in the even buffer (since p = 8).

At the completion of this subroutine, all of the predictor coefficients have been calculated. Many of the popular distortion measures use these parameters in the recognition calculations, such as the Itakura log-likelihood ratio. If the predictor coefficients are the desired features, conversion from floating-point back to a fixed-point representation finishes the routine.

The present system uses a cepstral representation. The following recursion is also used to convert from the predictor coefficients to the cepstral coefficients.

$$c_1 = -a_1$$

$$c_k = -a_k - \sum_{i=1}^{k-1} a_i c_{k-i} \left(\frac{k-i}{k} \right) \quad 1 \le k \le p$$

$$c_k = -\sum_{i=1}^{p} a_i c_{k-i} \left(\frac{k-i}{k} \right) \quad p < k$$

The implementation of this algorithm in subroutine `alpha_to_cep` is straightforward, and is commented in the code. From the eight predictor coefficients, twelve cepstral coefficients are calculated. These twelve coefficients are also used in several well-known distortion measures, and can be used directly following conversion to a fixed-point representation.

You can obtain better performance by using a window in the cepstral domain to weight each coefficient. Several different weights are described and evaluated in the literature, including weighting by the inverse variance of the coefficients or weighting by a raised sine function. The weighting chosen for this implementation is shown below:

$$w(k) = 1 + 6\sin\left(\frac{\pi k}{12} \right) \quad 1 \le k \le 12$$

The subroutine `weight_cep`, used to weight the cepstral coefficients, is also straightforward. The weighting values are initialized in a separate buffer, making them easier to modify.

The next subroutine *normalize_cep*, normalizes the twelve cepstral coefficients to the length of the entire vector. Normalization is necessary for the cepstral projection distortion measure. The length of the vector is the square-root of the sum of each coefficient squared. To square each coefficient, multiply it by itself. These values are then accumulated in a temporary variable. The square-root subroutine calculates an approximate square-root of the mantissa. This subroutine is adapted from a subroutine in Chapter 4, *Function Approximation*, of *Digital Signal Processing Applications Using the ADSP-2100 Family*, Volume 1, and calculates an approximate square-root of the mantissa. If the exponent is even, it is divided by two, giving the correct floating-point result. If the exponent is odd, it is incremented and divided by two, and the mantissa is scaled by

$$\frac{1}{\sqrt{2}}$$

This results in the appropriate value. Each cepstral coefficient is then scaled by this calculated length, using a floating-point divide routine.

The final step is to convert the floating-point cepstral coefficients back to fixed-point using `cep_to_fixed` . The results are written over the original input buffer.

6.3.2 Isolated Word Recognition

Following the word acquisition, one of two things happens. If the system is in the training mode, the word is stored in the library, and a record is kept of its location and length. In the recognition mode, this unknown word is compared to each template in the library, and the recognition result value is returned.

6.3.2.1 Library Routines

The library routines store and catalog the acquired words. The words are stored in a template library that occupies the external program memory of the ADSP-2100 Family processor (as currently implemented). The most important function of these routines is to store a new word in the template library. The code uses several variables to organize the library. These include a variable for catalog size (tracks of the number of words in the library); a library catalog is built as words are added. Two values are stored for each library template. The first value represents a pointer to the starting location of the template. The second value represents the length of the template, and it is stored as the number of vectors in the word. A final variable records the location of the next available catalog entry.

While template storage and library catalog maintenance are the most important functions of the library routines, the code contains other optional routines including playback. If the features representing the library are reflection coefficients, the words are played back through the speaker. Several variations of playback are available. A single template may be played, all the templates in the library may be played in order, or the library templates may be played in any order. This final routine is useful to play the library templates after recognition, beginning with the most probable word and ending with the least.

6.3.2.2 Comparison

A comparison routine compares an unknown word to the full library. Several comparison routines exist, differing only in the distance measure used for the recognition. In the implemented system, four different people are used for training, with each person's speech stored in a different bank of program memory.

The unknown word is compared to each of the four template banks separately. Each bank has its own library catalog, storing the location and length of each entry in the bank. Using a bank's catalog, the comparison subroutine initializes values needed to compare the unknown word to each of the specified bank's templates, in order. The comparison includes dynamic time warping and the distortion measure (see Section 6.3.2.3, *Dynamic Time Warping*, for more information). The comparison subroutine must be called once for each bank used.

The result of the comparison between the unknown word and a template is the word distance score. A buffer must be specified to hold these double-precision results, msw followed by lsw. These word distance scores are stored in the same order as the words stored in the library. A different buffer is used for each bank. After an unknown word is compared to each template in all four banks, the results are stored in four separate distance buffers.

Since all banks contain the same vocabulary in the same order, four word distance scores exist for each template. A *K-Nearest Neighbor* routine averages the results for each word. The implemented algorithm finds the two lowest scores of the four scores for each vocabulary word. These two are then summed, resulting in the final word distance score. This final word distance score is found for each word of the vocabulary. Using the K-Nearest Neighbor decision algorithm, the speech recognition becomes speaker-independent.

6.3.2.3 Dynamic Time Warping

The speech recognition code contains a complete system to perform dynamic time warping, or DTW, between two words. DTW dynamically matches two patterns of different lengths. In this case, each pattern represents a word, and each pattern is represented by a time sequence of feature vectors taken from a moving window of speech. The DTW algorithm aligns the time axis of the library word with the time axis of the unknown word, leading to the lowest possible word distance score.

The constraints used in this implementation were suggested by Itakura. This example tries to match unknown word(x), of length N, to a library word(y), of length, M. The indices x and y refer to a particular time frame of speech data, represented by a feature vector. A distance matrix can be calculated to represent the distance between an x (unknown word) feature vector and all y (library word) feature vectors, evaluated for 0 <= x <= N. Each point of the distance matrix has a value that is the distance between a single x feature vector and a single y feature vector. The specific distance measure used between feature vectors is arbitrary. The distance matrix is the only thing DTW needs.

To warp the time axis of the library word to the time axis of the unknown word, several constraints must be set. The starting point of the warping is (0,0), and the ending point must always be (N,M). The minimum slope of the warp is 1/2, and the maximum slope is 2. Finally, two consecutive slopes of 0 are not allowed. Figure 6.3 shows a diagram of a distance matrix with these constraints.

As this diagram shows, most of the distance matrix is invalid when the slope and warping constraints are imposed. Significant execution time is saved if only valid warping paths are considered, and only vector distances within the warp boundaries are calculated.

To determine the boundaries of the warping, the points A and B (or x_A and x_B), shown in the diagram, must be calculated. The following equations represent these two points:

$$x_A = \frac{1}{3}(2M - N)$$

$$x_B = \frac{2}{3}(2N - M)$$

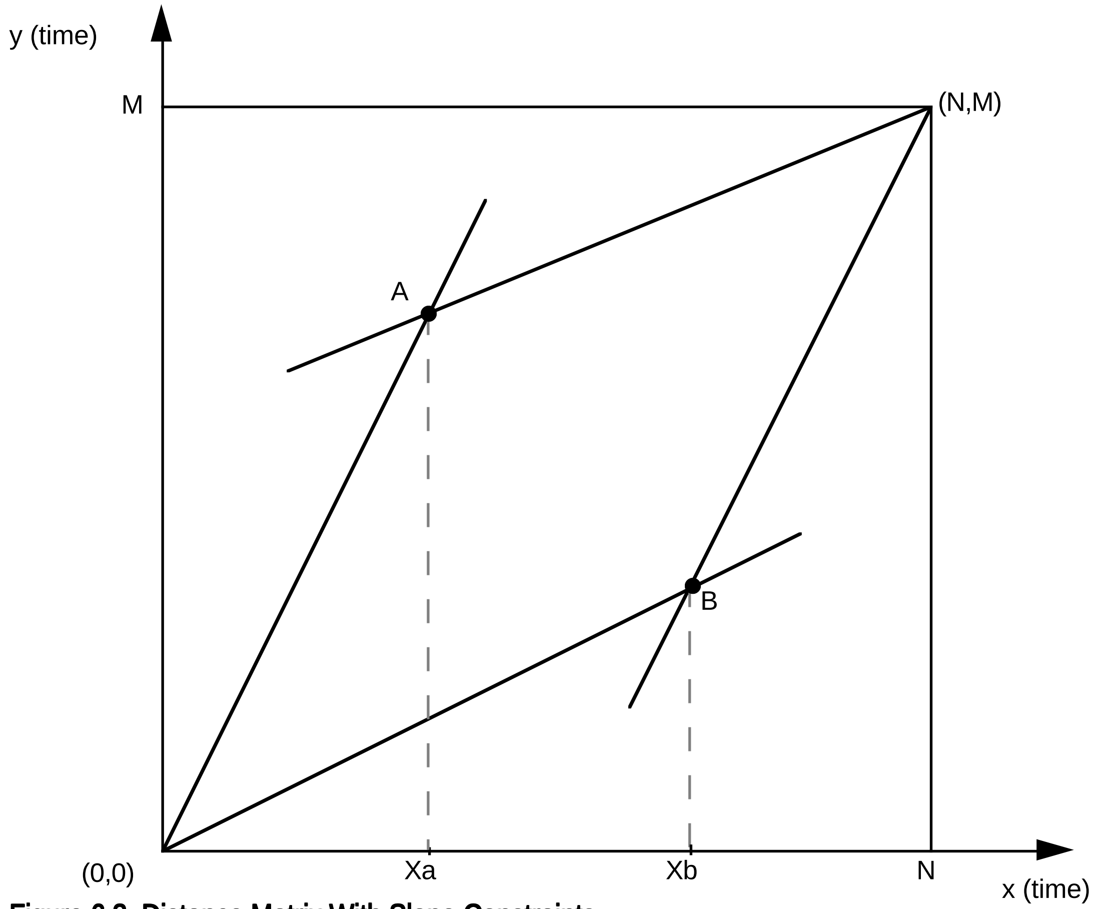

Figure 6.3 Distance Matrix With Slope Constraints

Since the actual processing is performed only at points where x and y are integers, the values of x_A and x_B are rounded down to the nearest integer in all cases, without loss of accuracy.

The values of x_A and x_B must be in the range of $0 < x_A < N$ and $0 < x_B < N$. This imposes a constraint on the lengths of the unknown word and the library word. The equations for this requirement are:

$$2M - N \geq 3$$

$$2N - M \geq 2$$

If this relation is not met, the two words cannot be warped together in this implementation.

Finally, the minimum and maximum y values must be determined for each x value. The equations for this are:

y minimum

$$= \frac{1}{2}x \quad 0 \le x \le x_B$$
$$= 2x + (M - 2N) \quad x_B < x \le N$$

y maximum

$$= 2x \quad 0 \le x \le x_A$$
$$= \frac{1}{2}x + \left(M - \frac{1}{2}N\right) \quad x_A < x \le N$$

The warping can be broken into two or three sections, based on the relationship of $x_A$and x_B. $x_A$can be less than, greater than, or equal to x_B. Each of these cases has different boundaries for each section, as summarized below in Table 6.1.

Section	$x_A < x_B$	$x_B < x_A$	$x_A = x_B$
1	$0 <= x <= x_A$	$0 <= x <= x_B$	$0 <= x <= x_A, x_B$
2	$x_A < x <= x_B$	$x_B < x <= x_A$	$x_A, x_B < x <= N$
3	$x_B < x <= N$	$x_A < x <= N$	none

Table 6.1 Time Warping Boundaries

For each case, the boundaries of y are different, but the warping is the same. The DTW finds the path of minimum word distance through the distance matrix, while considering the given constraints. This is done sequentially, beginning at x = 0 and ending at x = N. The following recursion shows the path through the matrix that is subject to warping constraints.

$$D(x,y) = d(x,y) + \min\left[D(x-1,y), D(x-1,y-1), D(x-1,y-2)\right] \quad 0 \le x \le N$$

D(x,y) represents the (intermediate) word distance score at (x,y), and d(x,y) is the value (vector distance) at point (x,y).

Since the recursion only involves the values in columns (x-1) and x, the complete distance matrix does not need to be calculated before the recursion begins. Instead, two buffers are set up. The intermediate sum buffer contains the values of D(x-1,y) for all y, organized as msw, lsw, warp value. The vector distance buffer contains the values of d(x,y) for all allowable y, organized as msw, lsw, empty location. The warp value in the intermediate sum buffer is the previous warp value (from column (x-2) to (x-1)), and is required to determine the allowable warp from column x-1 to x.

When the time warping commences from column x-1 to column x, the values in the intermediate sum buffer are examined to determine the minimum intermediate sum present in the allowed warping path. This minimum is then added to the value of the vector distance and placed in the vector distance buffer, along with the slope of the warp used. Figure 6. 4 shows the allowable paths. After the warping is complete for all values of y (y minimum–y maximum) the vector distance buffer contains the current intermediate sums. Before the next column is processed, these values must be copied into the intermediate sum buffer.

The recursion continues until x=N, when the vector distance buffer contains the final word distance score.

A single exception exists to the constraints on the warping path. It specifies that a warp of 0 is not allowed for two consecutive warps. However, since only integer indices are considered for (x,y), a case exists

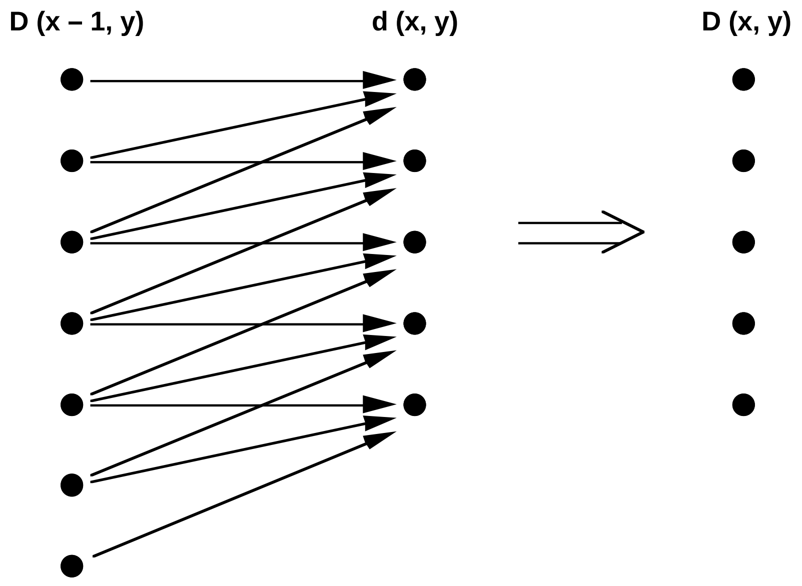

Figure 6.4 Time Warping Paths Between Intermediate Sums & Vector Distances

where the best option is to allow two consecutive 0 warps. You can saturate the intermediate distance calculated at the exception point, but this option yields an unknown effect on the recognition.

The warping path is constrained by saturating the intermediate sum values that, if selected, result in illegal warps. This insures that these paths are selected.

6.3.2.4 Ranking

After the unknown input word is compared to the complete template library, the routine returns a single buffer containing a word distance score for each vocabulary word. The ranking routine then compares the scores for each template. The routine finds the smallest distance contained in the buffer and places a pointer to the corresponding library catalog entry in the candidate order buffer. The buffer is filled, storing the most probable recognition candidate first, the second next, and so on. The least word distance score is considered the most likely recognition result. The location of each word's entry in the catalog library is the stored value. A separate buffer contains the candidate's number in the library (first, second, tenth, etc.), stored in the same order. The returned buffer for candidate order contains pointers to library catalog entries of template words, in order from the least word distance score to the greatest word distance score. The second buffer contains each candidate's number in the library, stored in the same order.

You will probably need to make minor modifications to the code so it will be compatible with your particular application.

6.3.3 Main Shell Routines

This speech recognition system has two different main shell routines. The *executive shell* is used for the initial training of the system, and can be used for testing the resulting templates. The *demonstration shell* is used after the system is fully trained, and it is designed to demonstrate hands-free dialing for a telephone application, although the recognition system can be used for any application.

Only one of these shells is used at a time. Since the code is written in a modular fashion, and includes many subroutine calls, this scheme is possible. Both shells contain an interrupt table, initialization functions, and an interrupt routine used to process samples. The interrupt sample routine has an output flag to select whether data is being input or output, since both are not done at the same time. The additional features of each shell are described in more detail in the following sections.

6.3.3.1 Executive Shell

The executive shell (EXECSHEL.DSP), shown in Listing 6.1, calls the functions necessary for speech recognition: getting an input word, adding a word to the library, and recognizing a word. Figure 6.5 shows the link file menu tree used by EXECSHEL.DSP. The interface is the minimum necessary to accomplish the tasks. If the LPC reflection coefficients are used as features, this shell can call routines to playback a single word or the entire library. If another representation is used, the recognized word can be output to a display.

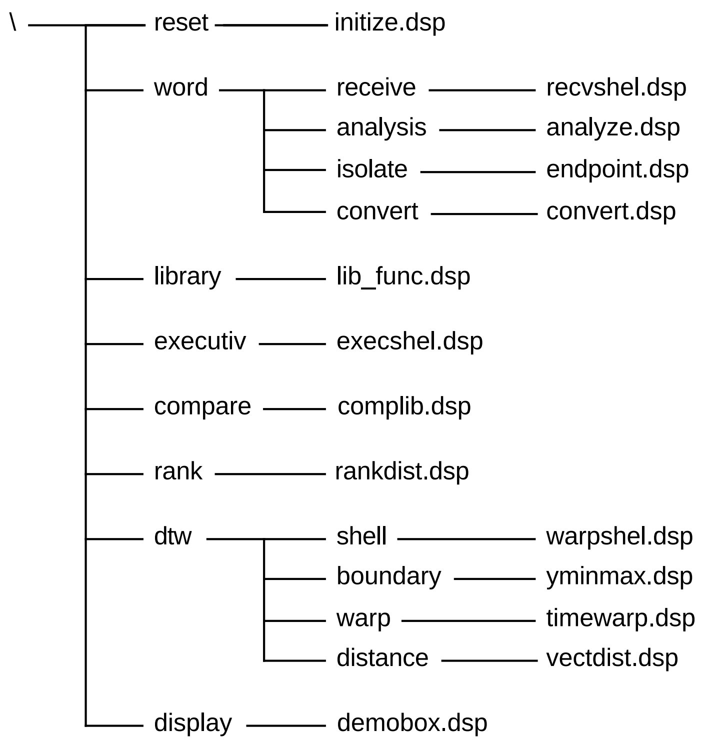

Figure 6.5 EXECSHEL.DSP Link File Menu Tree

The code is organized into one main loop. On reset, the system gets an input word. This word is either added to the recognition library or compared for recognition. An interrupt must be asserted before the word is spoken if the word will become a library template. Note that the interrupt is only enabled during the word acquisition routine.

6.3.3.2 Demonstration Shell

The demonstration shell (DEMOSHEL.DSP), shown in Listing 6.2, calls some of the functions necessary for speech recognition: getting an input word and recognizing a word. It also calls many display routines. Figure 6.6 shows the link file menu tree used by DEMOSHEL.DSP. The interface is designed for a demonstration of hands-free dialing. The specific display routines can be changed to communicate with any desired display.

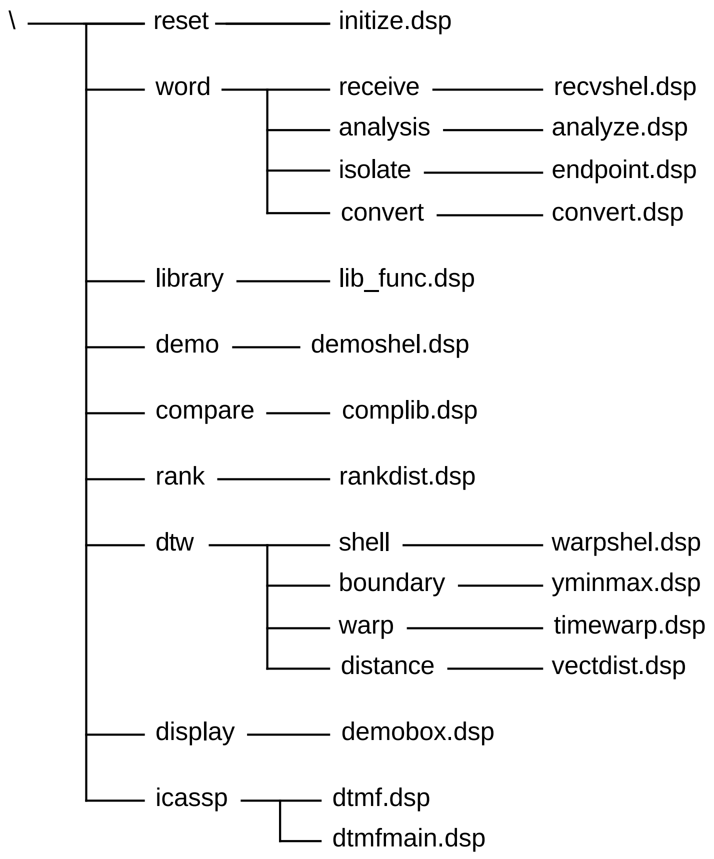

Figure 6.6 DEMOSHEL.DSP Link File Menu Tree

The code is organized to reflect the different stages of dialing a telephone. Using a fifteen word vocabulary (the letter "o", zero, one, two, three, four, five, six, seven, eight, nine, dial, delete, phonecall, scratch), the demonstration accepts a spoken phone number as isolated words, then dials. After reset, the demonstration continuously gets input words for recognition. The command, "phonecall," alerts the processor that a phone number is about to be spoken. Once "phonecall" is recognized, the demonstration moves to the next stage.

The demonstration then accepts the first digit of the number, and moves into the final state of the system. In the last state, the processor adds each digit to the phone number, as it is spoken. When the command “dial” is recognized, the software boots the next boot page, which consists of a dialing routine.

If a mistake is made during recognition, the command “delete” removes the preceding digit from the phone number. Repeatedly speaking “delete” continues to erase digits. To reset the demonstration use the command “scratch” to return the demonstration to its initial state, where it waits for the command “phonecall.”

This demonstration shell performs a basic calling routine. It could serve as a framework for an actual implementation. Functions that might be added include: local or long distance dialing, memory dialing, and so on.

6.4 HARDWARE IMPLEMENTATION

The speech recognition uses a hardware platform designed specifically for this application. This expansion board is connected to the ADSP-2101 EZ-LAB® Demonstration Board through the EZ-LAB connector. Figure 6.7 is the schematic diagram for this circuit board.

6.5 LISTINGS

This section contains the listings for this chapter.

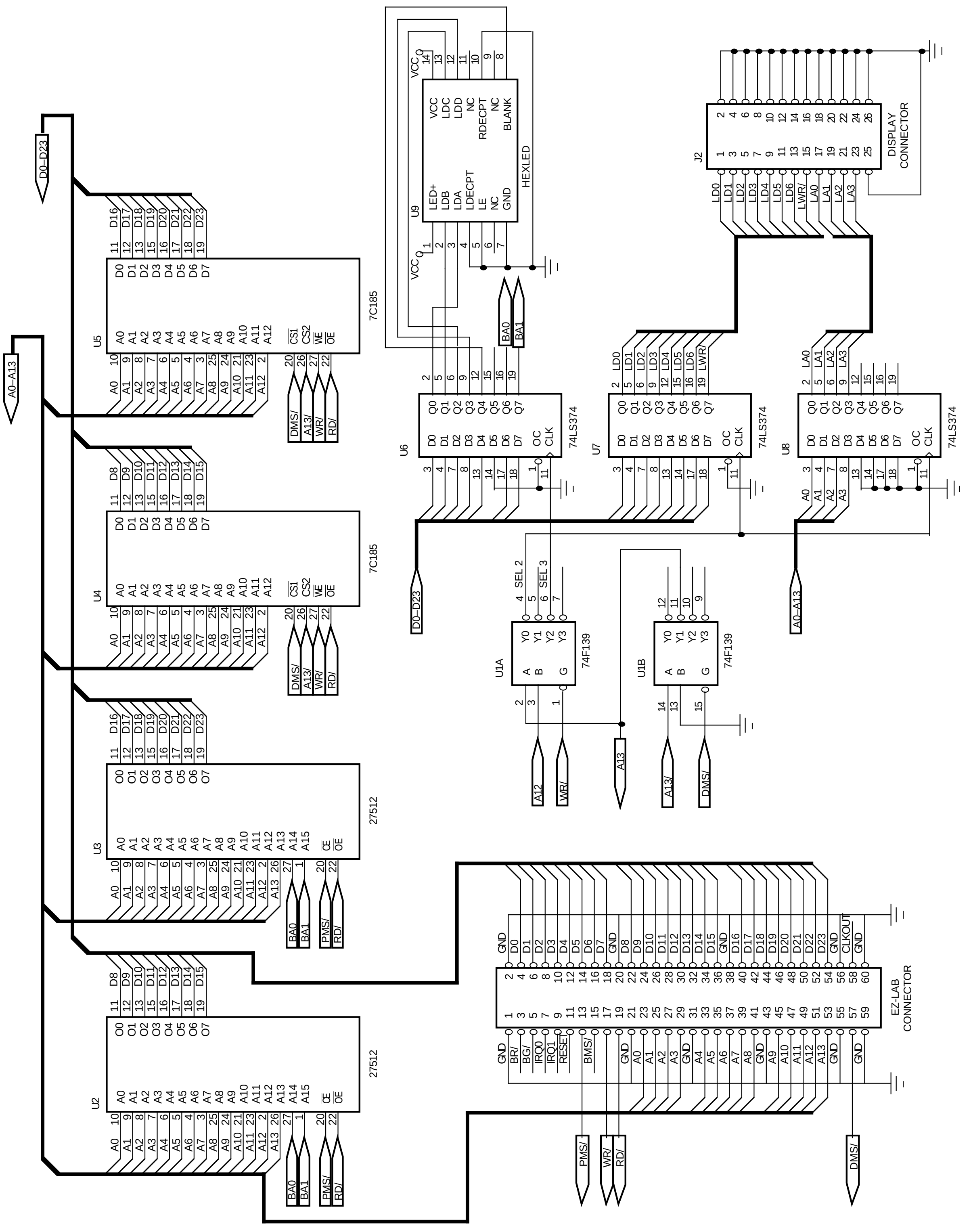

Figure 6.7 Speech Recognition System Circuit Board Schematic Diagram

```
.MODULE/ABS=0/RAM/BOOT=0      executive_shell;

.VAR/DM/RAM       flag;
.VAR/DM/RAM       output_flag;
.VAR/DM/RAM       unknown_feature_dimension;
.VAR/DM/RAM       library_feature_dimension;

.GLOBAL           output_flag;
.GLOBAL           unknown_feature_dimension;
.GLOBAL           library_feature_dimension;
.GLOBAL           flag;

.EXTERNAL         get_word;
.EXTERNAL         put_in_library;
{.EXTERNAL        play_library;}
{.EXTERNAL        play_single;}
{.EXTERNAL        coarse_compare;}
{.EXTERNAL        fine_compare;}
{.EXTERNAL        full_compare;}
{.EXTERNAL        shuffle_play;}
.EXTERNAL         rank_candidates;
.EXTERNAL         reset_recog;
.EXTERNAL         display_digit;
{.EXTERNAL        add_a_digit;}
{.EXTERNAL        display_number;}
.EXTERNAL         set_bank_select;
.EXTERNAL         show_bank;
.EXTERNAL         cepstral_compare;
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{_______________________main shell for speech recognition_____________________}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

reset_vector:     JUMP start; NOP; NOP; NOP;
irq2:             JUMP toggle_flag; NOP; NOP; NOP;
trans0:           NOP; NOP; NOP; NOP;
recv0:            JUMP sample; NOP; NOP; NOP;
trans1:           NOP; NOP; NOP; NOP;
recv1:            NOP; NOP; NOP; NOP;
timer_int:        NOP; NOP; NOP; NOP;

start:            IMASK=0;
                  ICNTL=B#00100;
                  L0=0;   L1=0;   L2=0;   L3=0;
                  L4=0;   L5=0;   L6=0;   L7=0;
                  M0=0;   M1=1;   M2=-1;  M3=2;
                  M4=0;   M5=1;   M6=-1;  M7=2;
```

(listing continues on next page)

```
reg_setup:  AX0 = 0;
            DM(0X3FFE) = AX0;       { DM wait states }
            AX0 = 2;
            DM(0X3FF5) = AX0;       { sclkdiv0 with 12.288 MHz input }
            AX0 = 255;
            DM(0X3FF4) = AX0;       { rfsdiv0 }
            AX0 = 0X6927;
            DM(0X3FF6) = AX0;       { control reg0 }
            AX0 = 0X1004;
            DM(0X3FFF) = AX0;       { system control reg }

            CALL reset_recog;
            AR = 4;
            CALL set_bank_select;
            CALL show_bank;

recognition:
            IMASK = 0x20;
            CALL get_word;
            AY0 = DM(flag);
            AF  = PASS AY0;
            IF NE JUMP build_library;

            CALL cepstral_compare;
            CALL rank_candidates;   { buffer pointer returned in AY0 }

            CALL display_digit;

            AX0 = 1;                { play the top three candidates}
{           CALL shuffle_play;}
            JUMP recognition;

build_library:
            CALL put_in_library;
{           CALL play_library;}
            AX0 = 0;
            DM(flag) = AX0;
            JUMP recognition;

{______________________toggle record/recognize flag__________________________}
toggle_flag:
            ENA SEC_REG;
            MR0 = DM(flag);
            AR  = NOT MR0;
            DM(flag) = AR;
            RTI;
```

```
{______________________process sample________________________________________}
sample:  ENA SEC_REG;

         AR  = DM(output_flag);
         AR  = PASS AR;
         IF EQ JUMP get_input;

send_output:
         SR1 = DM(I7,M5);
         SR  = ASHIFT SR1 BY -2 (HI);
         TX0  = SR1;
         JUMP inc_count;

get_input:
         AR=RX0;
         TX0 = AR;
         DM(I7,M5)=AR;        {Save sample}

inc_count:
         AY0=MX0;
         AR=AY0+1;
         MX0=AR;
         RTI;

.ENDMOD;
```

Listing 6.1 Executive Shell Subroutine (EXECSHEL.DSP)

```
.MODULE/ABS=0/RAM/BOOT=0        demonstration_shell;

.VAR/DM/RAM flag;
.VAR/DM/RAM output_flag;
.VAR/DM/RAM unknown_feature_dimension;
.VAR/DM/RAM library_feature_dimension;

.GLOBAL     flag;
.GLOBAL     output_flag;
.GLOBAL     unknown_feature_dimension;
.GLOBAL     library_feature_dimension;

.EXTERNAL   get_word;
.EXTERNAL   put_in_library;
{.EXTERNAL  play_library;}
{.EXTERNAL  play_single;}
{.EXTERNAL  coarse_compare;}
{.EXTERNAL  fine_compare;}
{.EXTERNAL  full_compare;}
{.EXTERNAL  shuffle_play;}
.EXTERNAL   rank_candidates;
.EXTERNAL   reset_recog;
.EXTERNAL   init_catalog;
.EXTERNAL   catalog_size;
.EXTERNAL   inc_bank_select;
.EXTERNAL   show_bank;
.EXTERNAL   set_local_call;
.EXTERNAL   set_long_distance;
.EXTERNAL   digit_count;
.EXTERNAL   display_number;
.EXTERNAL   display_digit;
.EXTERNAL   add_a_digit;
.EXTERNAL   display_numpls;
.EXTERNAL   display_dial;
.EXTERNAL   reset_display;
.EXTERNAL   timed_display;
.EXTERNAL   reset_timed;
.EXTERNAL   cepstral_compare;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{_______________main shell for speech recognition demonstration______________}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
```

```
reset_vector:
            JUMP start; NOP; NOP; NOP;
irq2:       JUMP next_bank; NOP; NOP; NOP;
trans0:     RTI; NOP; NOP; NOP;
recv0:      JUMP sample; NOP; NOP; NOP;
trans1:     NOP; NOP; NOP; NOP;
recv1:      NOP; NOP; NOP; NOP;
timer_int:  JUMP timed_display; NOP; NOP; NOP;

start:   IMASK=0;
         ICNTL=B#00100;
         L0=0;   L1=0;   L2=0;   L3=0;
         L4=0;   L5=0;   L6=0;   L7=0;
         M0=0;   M1=1;   M2=-1;  M3=2;
         M4=0;   M5=1;   M6=-1;  M7=2;

reg_setup:
         AX0 = 0;
         DM(0X3FFE) = AX0;               { DM wait states }
         AX0 = 2;
         DM(0X3FF5) = AX0;               { sclkdiv0 with 12.288 MHz input }
         AX0 = 255;
         DM(0X3FF4) = AX0;               { rfsdiv0 }
         AX0 = 0X6927;
         DM(0X3FF6) = AX0;               { control reg0 }
         AX0 = 0X1003;
         DM(0X3FFF) = AX0;               { system control reg }

         CALL reset_recog;
         CALL reset_display;
         {CALL play_library;}

{_______________wait for (phonecall) while displaying intro__________________}

phone_call:
         IMASK = 0x21;
         ENA TIMER;
         CALL get_word;
         DIS TIMER;
         CALL cepstral_compare;
         CALL rank_candidates;           { buffer pointer returned in AY0 }

         { failsafe feature }
         IF NOT FLAG_IN JUMP its_a_call;

         { is it (phonecall)? }
         AX0 = 14;
         AF  = AX0 - AY1;
         IF NE JUMP phone_call;
```

(listing continues on next page)

```
         { decrement catalog_size to remove (phonecall) }
its_a_call:
         AY0 = DM(catalog_size);
         AR  = AY0 - 1;
         DM(catalog_size) = AR;

{_______________wait for digit while displaying (number please?)_____________}

first_digit:
         CALL display_numpls;       { display }
         AX0 = 0;
         DM(digit_count) = AX0;
         CALL set_local_call;

         CALL get_word;
         CALL cepstral_compare;
         CALL rank_candidates;      { buffer index returned in AY1 }

         { is it (question_mark)?}
         AX0 = 14;
         AF  = AX0 - AY1;
         IF GT JUMP chk_scratch1;
         AY1 = 15;
         CALL display_digit;
         JUMP first_digit;

         { is it (scratch)?}
chk_scratch1:
         AX0 = 12;
         AF  = AX0 - AY1;
         IF EQ JUMP catsiz_reset;

         { is it (dial) or (delete)? }
         AX0 = 11;
         AF  = AX0 - AY1;
         IF EQ JUMP first_digit;
         AX0 = 13;
         AF  = AX0 - AY1;
         IF EQ JUMP first_digit;

         { is it (one)? }
         AX0 = 1;
         AF  = AX0 - AY1;
         IF EQ CALL set_long_distance;
         CALL add_a_digit;          { increment digit_count }
         CALL display_digit;        { display digit }
```

```
{______________collect and display remaining digits, wait for (dial)_________}

more_digits:
         CALL display_number;       { display number }
         CALL get_word;
         CALL cepstral_compare;
         CALL rank_candidates;      { buffer pointer returned in AY0 }

         { failsafe feature }
         IF NOT FLAG_IN JUMP dial_number;

         { is it (question_mark)?}
         AX0 = 14;
         AF  = AX0 - AY1;
         IF GT JUMP chk_scratch2;
         AY1 = 15;
         CALL display_digit;
         JUMP more_digits;

         { is it (scratch)?}
chk_scratch2:
         AX0 = 12;
         AF  = AX0 - AY1;
         IF EQ JUMP catsiz_reset;

         { is it (dial)? }
         AX0 = 11;
         AF  = AX0 - AY1;
         IF EQ JUMP dial_number;

         { is it (delete)? }
         AX0 = 13;
         AF  = AX0 - AY1;
         IF NE JUMP its_a_digit;
         AY0 = DM(digit_count);
         AR  = AY0 - 1;
         IF EQ JUMP first_digit;
         DM(digit_count) = AR;
         JUMP more_digits;

its_a_digit:
         CALL add_a_digit;          { increment digit_count }
         CALL display_digit;        { display digit }

         JUMP more_digits;
```

(listing continues on next page)

```
{______________________reset catalog_size____________________________________}

catsiz_reset:
         AY0 = DM(catalog_size);
         AR  = AY0 + 1;
         DM(catalog_size) = AR;
         CALL reset_timed;
         JUMP phone_call;

{______________________boot code to dial the number__________________________}

dial_number:
         CALL display_dial;
         AR  = 0X025B;
         DM(0X3FFF) = AR;     { boot page 1 }

{______________________enable new template library___________________________}

next_bank:
         ENA SEC_REG;
         CALL inc_bank_select;
         CALL show_bank;
         CALL init_catalog;
         RTI;

{______________________process sample________________________________________}

sample:  ENA SEC_REG;
         AR  = DM(output_flag);
         AR  = PASS AR;
         IF EQ JUMP get_input;

send_output:
         SR1 = DM(I7,M5);

         SR  = ASHIFT SR1 BY -2 (HI);
         TX0  = SR1;
         JUMP inc_count;

get_input:
         AR=RX0;
         DM(I7,M5)=AR;        {Save sample}
inc_count:
         AY0=MX0;
         AR=AY0+1;
         MX0=AR;
         RTI;

.ENDMOD;
```

Listing 6.2 Demonstration Shell Subroutine (DEMOSHEL.DSP)

```
{_____________________________________________________________________________

         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
______________________________________________________________________________

INITIZE.DSP

This routine performs necessary initialization of data memory variables for
the speech recognition system.  There are several assembly switches switches
available.  The -Dplayback switch is used when reflection coefficients are
stored as features and templates are to be output to a speaker. The -Dinit_lib
switch is used to initialize a rom copy of the library catalogs.  One of the
two remaining switches MUST be used.  The -Drecord switch is used with
external program ram to record new templates.  The -Ddemo switch is used in a
rom-based system for the demonstration and recognition accuracy testing.

The conditional assembly options and a description of each follows.  At a
minimum, assembly must include:

   asm21 INITIZE -cp    -Drecord          used when recording new templates

OR

   asm21 INITIZE -cp    -Ddemo            used when demonstrating system,
                                          templates and catalog already stored in
                                          rom or initialized with switch

The other options are:  -Dplayback        allows playback of library templates
                                          if reflection coefficients are the
                                          stored features
                        -Dinit_lib        used to initialize the template library
                                          catalog with data contained in the file
                                          catalog.dat
_____________________________________________________________________________}

.MODULE/RAM/BOOT=0      initialize;

.VAR/PM/RoM/SEG=EXT_PM  catalog_init[32];

.EXTERNAL   threshold_time;
.EXTERNAL   min_word_length;
.EXTERNAL   ws_energy_thresh;
.EXTERNAL   ws_zcr_thresh;
.EXTERNAL   ps_energy_thresh;
.EXTERNAL   ps_zcr_thresh;
```

(listing continues on next page)

```
.EXTERNAL   flag;
.EXTERNAL   unknown_feature_dimension;
.EXTERNAL   library_feature_dimension;

.EXTERNAL   catalog_size;
.EXTERNAL   library_catalog;
.EXTERNAL   next_catalog_entry;
.EXTERNAL   template_library;
.EXTERNAL   catalog_init;
{..............................................................................}
{ conditional assembly use -Dplayback }
#ifdef playback
.EXTERNAL   voiced_energy_thresh;
.EXTERNAL   spseed_lsw;
.EXTERNAL   spseed_msw;
.EXTERNAL   synth_train;
#endif
{..............................................................................}

.ENTRY      reset_recog;
{..............................................................................}
{ conditional assembly use -Ddemo }
#ifdef demo
.ENTRY      init_catalog;     { necessary when multiple template banks used  }
#endif
{..............................................................................}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________initialize data memory variables_____________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

reset_recog:
            AX0 = 15;                     { variables from endpoint detection }
            DM(threshold_time) = AX0;
            AX0 = 30;
            DM(min_word_length) = AX0;
            AX0 = 1000;
            DM(ws_energy_thresh) = AX0;
            AX0 = 11000;
            DM(ws_zcr_thresh) = AX0;
            AX0 = 250;
            DM(ps_energy_thresh) = AX0;
            AX0 = 5500;
            DM(ps_zcr_thresh) = AX0;

            AX0 = 0;                      { shell variables }
            DM(flag) = AX0;
            AX0 = 12;
            DM(unknown_feature_dimension) = AX0;
            AX0 = 12;
            DM(library_feature_dimension) = AX0;
```

```
{..............................................................................}
{ conditional assembly use -Dplayback }
#ifdef playback

         AX0 = 3072;                     { synthesis variables }
         DM(voiced_energy_thresh) = AX0;
         AX0 = 15381;
         DM(spseed_lsw) = AX0;
         AX0 = 7349;
         DM(spseed_msw) = AX0;
#endif {.......................................................................}

{_______________________EITHER this section__________________________________}

{..............................................................................}
{ conditional assembly use -Drecord }
#ifdef record
         AX0 = 0;                        { for recording new templates }
         DM(catalog_size) = AX0;
         AX0 = ^template_library;
         DM(library_catalog) = AX0;
         AX0 = ^library_catalog;
         DM(next_catalog_entry) = AX0;
#endif
{..............................................................................}

{_______________________OR this section______________________________________}

{..............................................................................}
{ conditional assembly use -Dinit_lib }
#ifdef init_lib
.INIT    catalog_init : <catalog.dat>;    { for initializing external pm }
#endif {.......................................................................}
{..............................................................................}
{ conditional assembly use -Ddemo }
#ifdef demo
init_catalog:     I0  = ^catalog_size;    { necessary when multiple template }
                  I4  = ^catalog_init;    { banks are used }
                  CNTR = 32;
                  DO copy_catalog UNTIL CE;
                        AX0 = PM(I4,M5);
copy_catalog:           DM(I0,M1) = AX0;
#endif
{..............................................................................}

{____________________________________________________________________________}

            RTS;

.ENDMOD;
```

Listing 6.3 Data Variable Initialization Routine (INITIZE.DSP)

```
.MODULE/RAM/BOOT=0      receive_shell;

.CONST            vector_buffer_length = 1200;  {enough for one second}

.VAR/DM/RAM       LPC_coeff_buffer[12];
.VAR/DM/RAM/CIRC  input_buffer[240];
.VAR/DM/RAM       frame_buffer[160];
.VAR/DM/RAM       frame_pntr;
.VAR/DM/RAM       vector_pntr;
.VAR/DM/RAM       feature_vector_buffer[vector_buffer_length];
.VAR/DM/RAM       vector_count;
.VAR/DM/RAM/CIRC  start_buffer[60];
.VAR/DM/RAM       start_buffer_pntr;

.ENTRY      get_word;

.EXTERNAL   analyze_frame;
.EXTERNAL   frame_energy, frame_zcr;
.EXTERNAL   word_start_flag, poss_start_flag;
.EXTERNAL   word_end_flag, find_endpoints;
.EXTERNAL   unknown_feature_dimension;
.EXTERNAL   output_flag;
.EXTERNAL   convert_coeffs;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________receive input word___________________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________initialize buffers and analysis variables____________}

get_word:AX0 = 0;

         DM(vector_count) = AX0;
         DM(output_flag) = AX0;
         DM(word_end_flag) = AX0;
         DM(word_start_flag) = AX0;
         DM(poss_start_flag) = AX0;

         I0  = ^frame_buffer;
         CNTR = 160;
{        DO dmloop1 UNTIL CE;}
dmloop1:    DM(I0,M1) = AX0;
            IF NOT CE JUMP dmloop1;

         I0  = ^input_buffer;
         CNTR = 240;
{        DO dmloop2 UNTIL CE;}
```

```
dmloop2:    DM(I0,M1) = AX0;
            IF NOT CE JUMP dmloop2;

         I0  = ^start_buffer;
         CNTR = 60;
{        DO dmloop3 UNTIL CE;}
dmloop3:    DM(I0,M1) = AX0;
            IF NOT CE JUMP dmloop3;

         AX0 = ^input_buffer;
         DM(frame_pntr) = AX0;
         AX0 = ^feature_vector_buffer + 60;
         DM(vector_pntr) = AX0;
         AX0 = ^start_buffer;
         DM(start_buffer_pntr) = AX0;

         RESET FLAG_OUT;

         I7  = ^input_buffer;       {I7 is speech buffer pointer}
         L7  = 240;

         ENA SEC_REG;
         MX0=0;                     {sample recvd counter}
         AX1=160;

         AX0 = IMASK;
         AY0 = 0X08;
         AR  = AX0 OR AY0;
         IMASK = AR;                {0X28;}

{________________________acquire data frame___________________________________}

code_1_loop:
         AY1=MX0;
         AR=AX1-AY1;
         IF NE JUMP code_1_loop;
         AX1 = 80;
         MX0 = 0;
         DIS SEC_REG;

{________________________copy data to frame buffer____________________________}

         I0 = DM(frame_pntr);
         I1 = ^frame_buffer;
         CNTR = 160;
         L0 = 240;
{        DO copy_frame UNTIL CE;}
RSHELL1:    AX0 = DM(I0,M1);
copy_frame: DM(I1,M1) = AX0;
            IF NOT CE JUMP RSHELL1;
         {L0 = 0;}
```

(listing continues on next page)

```
{_______________________update frame pointer_________________________________}

         I0 = DM(frame_pntr);
         M3 = 80;
         {L0 = 240;}
         MODIFY (I0,M3);
         M3 = 2;
         L0 = 0;
         DM(frame_pntr) = I0;

{_______________________frame analysis_______________________________________}

do_dmr_1:
         I0=^frame_buffer;
         I1=^LPC_coeff_buffer;
         CALL analyze_frame;

{_______________________feature conversion___________________________________}

         I4  = ^LPC_coeff_buffer;
         CALL convert_coeffs;

{_______________________store feature vector_________________________________}

         I0  = DM(vector_pntr);
         I1  = ^LPC_coeff_buffer;

{............................................................................}
   { conditional assembly }
   #ifdef eight_features
         AX0 = DM(frame_energy);
         AX1 = DM(frame_zcr);
         DM(I0,M3) = AX0;                 {M3 = 2}
         DM(I0,M3) = AX1;

         AY0 = DM(unknown_feature_dimension);
         AR  = 4;                         {E, delta_E, zcr, delta_zcr}
         AR  = AY0 - AR;
         CNTR = AR;
   #else
         CNTR = DM(unknown_feature_dimension);
   #endif
{............................................................................}

{        DO write_feature UNTIL CE;}
RSHELL2:          AY0 = DM(I1,M1);
write_feature:    DM(I0,M1) = AY0;
                  IF NOT CE JUMP RSHELL2;
```

```
{________________________speech test___________________________________________}

         CALL find_endpoints;

{________________________update vector pointer, count__________________________}

test_sp_flags:
         AX0 = DM(word_start_flag);
         AY0 = DM(poss_start_flag);
         AR  = AX0 OR AY0;
         IF EQ JUMP test_word_end;
         SET FLAG_OUT;

update_vp:
         I0  = DM(vector_pntr);
         M3  = DM(unknown_feature_dimension);
         MODIFY(I0,M3);
         DM(vector_pntr) = I0;
         M3  = 2;

update_vc:
         AY0 = DM(vector_count);
         AY1 = 100;
         AR  = AY0 + 1;
         AF  = AY1 - AR;
         IF LE JUMP word_end;
         DM(vector_count) = AR;

         JUMP more_frames;

{________________________word test_____________________________________________}

test_word_end:
         RESET FLAG_OUT;
         AX0 = DM(word_end_flag);
         AF  = PASS AX0;
         IF EQ JUMP reset_vpvc;

word_end:
         IMASK = 0;

{____________________copy start_buffer to feature_vector_buffer_______________}

         I0  = ^feature_vector_buffer;
         I1  = DM(start_buffer_pntr);
         L1  = 60;
         CNTR = 60;
         DO copy_start UNTIL CE;
                  AX0 = DM(I1,M1);
copy_start:       DM(I0,M1) = AX0;

         L1  = 0;
```

(listing continues on next page)

```
{.............................................................................}
   { conditional assembly }
   #ifdef eight_features

{_______________________scale energy in word template________________________}

         I0  = ^feature_vector_buffer;
         CNTR = DM(vector_count);
         M3  = DM(unknown_feature_dimension);
         AF  = PASS 0, AX0 = DM(I0,M3);
         DO find_max_energy UNTIL CE;
            AR  = AF - AX0, AX0 = DM(I0,M3);
find_max_energy:
         xIF LT AF = PASS AX0;
         AR  = AF + 1;

         I0  = ^feature_vector_buffer;
         CNTR = DM(vector_count);
         AX0 = AR;
         DO scale_energy UNTIL CE;
            AY1 = DM(I0,M0);
            AY0 = 0;
            DIVS AY1,AX0;
            CNTR = 15;
            DO scale_divloop UNTIL CE;
scale_divloop:
         DIVQ AX0;
scale_energy:
         DM(I0,M3) = AY0;
         M3  = 2;

{_______________calculate delta energy,zcr in word template__________________}

         I0  = ^feature_vector_buffer;
         M2  = DM(unknown_feature_dimension);
         AY1 = DM(vector_count);
         AR  = AY1 - 1;

         AY0 = 0;
         I1  = I0;
         AX0 = DM(I1,M1);                 { read energy }
         DM(I1,M1) = AY0;                 { store delta energy }
         AX1 = DM(I1,M1);                 { read zero-crossings }
         DM(I1,M1) = AY0;                 { store delta zero-crossings }
         CNTR = AR;
         DO compute_deltas UNTIL CE;
            MODIFY(I0,M2);
            I1  = I0;
            AY0 = DM(I1,M1);              { read energy }
            AR  = AY0 - AX0, AX0 = AY0;
            DM(I1,M1) = AR;               { store delta energy }
            AY1 = DM(I1,M1);              { read zero-crossings }
            AR  = AY1 - AX1, AX1 = AY1;
compute_deltas:
         DM(I1,M0) = AR;                  { store delta zero-crossings }
         M2  = -1;

   #endif
{..............................................................................}

{_______________________word template complete - return______________________}

         L7  = 0;
         I0  = ^feature_vector_buffer;
         AX0 = DM(vector_count);

         RTS;

{_______________________reset vector pointer and count_______________________}

reset_vpvc:
         AX0 = ^feature_vector_buffer + 60;
         DM(vector_pntr) = AX0;
         AX0 = 0;
         DM(vector_count) = AX0;

{_______________________store vector in start buffer_________________________}

         I0  = DM(start_buffer_pntr);
         L0  = 60;
         I1  = ^LPC_coeff_buffer;

{..............................................................................}

   { conditional assembly }
   #ifdef eight_features
         AX0 = DM(frame_energy);
         AX1 = DM(frame_zcr);
         DM(I0,M3) = AX0;                 {M3 = 2}
         DM(I0,M3) = AX1;

         AY0 = DM(unknown_feature_dimension);
         AR  = 4;                         {E, delta_E, zcr, delta_zcr}
         AR  = AY0 - AR;
         CNTR = AR;
```

(listing continues on next page)

```
   #else
         CNTR = DM(unknown_feature_dimension);
   #endif
{..............................................................................}

{        DO write_start UNTIL CE;}
RSHELL9: AY0 = DM(I1,M1);
write_start:
         DM(I0,M1) = AY0;
         IF NOT CE JUMP RSHELL9;

         DM(start_buffer_pntr) = I0;
         L0  = 0;

{_______________________jump to get more frames______________________________}

more_frames:
         ENA SEC_REG;
         JUMP code_1_loop;

{==============================================================================}

.ENDMOD;
```

Listing 6.4 Receive Word Routine (RECVSHEL.DSP)

```
.MODULE/RAM/BOOT=0      LPC_analysis;

.ENTRY   analyze_frame;

.CONST   analysis_window_length = 160;
.CONST   input_scaler = 1;                 { 1 for u-law, 2 for A-law}
                                           { assumes right justified input}
.CONST   zcr_noise_threshold = 15;

.VAR/PM/RoM/SEG=EXT_PM
                  hamming_dat[analysis_window_length];
.VAR/DM/RAM       frame_energy;
.VAR/DM/RAM       frame_zcr;
.VAR/DM/RAM       spL_ACF[18];             {9 long (32-bit) words}
.VAR/DM/RAM       r[8];
.VAR/DM/RAM       k[8];
.VAR/DM/RAM       acf[9];
.VAR/DM/RAM       p[9];
{.VAR/DM/RAM      z1, L_z2_h, L_z2_l, mp;}
.VAR/DM/RAM       spscaleauto;             {Used in pre-emphasis save}
.VAR/DM/RAM       speech_in, xmit_buffer;

.INIT             hamming_dat : <hammdat.dat>;

.GLOBAL           spL_ACF, spscaleauto;
{.GLOBAL          mp, L_z2_l, L_z2_h, z1;}
.GLOBAL           frame_energy, frame_zcr;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{____________________________Analysis Subroutine_____________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

analyze_frame:
         ENA AR_SAT;                {Enable ALU saturation}
         DM(speech_in)=I0;          {Save pointer to input window}
         DM(xmit_buffer)=I1;        {Save pointer to coeff window}
         MX1=H#4000;                {This multiply will place the}
         MY1=H#100;                 {vale of H#80 in MF that will}
         MF=MX1*MY1 (SS);           {be used for unbiased rounding}
```

(listing continues on next page)

```
{__________________downscaling and offset compensation_______________________}

         I0=DM(speech_in);           {Get pointer to input data}
         I1=I0;                      {Set pointer for output data}
         SE=-15;                     {Commonly used shift value}
         MX1=H#80;                   {Used for unbaised rounding}
         AX1=16384;                  {Used to round result}
         MY0=32735;                  {Coefficient value}
         AY1=H#7FFF;                 {Used to mask lower L_z2}

         MY1 = 0;
         MR  = 0;                    {Since frames now overlapped}
{        MY1=DM(z1);}
{        MR0=DM(L_z2_l);}
{        MR1=DM(L_z2_h);}
         DIS AR_SAT;                       {Cannot do saturation}
         AR=MR0 AND AY1, SI=DM(I1,M1);    {Fill the pipeline}
         CNTR=analysis_window_length;

{        DO offset_comp UNTIL CE;}
AN1:        SR=LSHIFT SI BY input_scaler (HI);{assumes right justified}
            AX0=SR1, SR=ASHIFT MR1 (HI);  {Get upper part of L_z2 (msp)}
            SR=SR OR LSHIFT MR0 (LO);     {Get LSB of L_z2 (lsp)}
            MR=MX1*MF (SS), MX0=SR0;      {Prepare MR, MX0=msp}
            MR=MR+AR*MY0 (SS), AY0=MY1;   {Compute temp}
            AR=AX0-AY0, AY0=MR1;          {Compute new s1}
            SR=ASHIFT AR BY 15 (LO);      {Compute new L_s2}
            AR=SR0+AY0, MY1=AX0;          {MY1 holds z1, L_s2+temp is in}
            AF=SR1+C, AY0=AR;             {SR in double precision}
            MR=MX0*MY0 (SS);              {Compute msp*32735}
            SR=ASHIFT MR1 BY -1 (HI);     {Downshift by one bit }
            SR=SR OR LSHIFT MR0 BY -1 (LO);{before adding to L_s2}
            AR=SR0+AY0, AY0=AX1;          {Compute new L_z2 in }
            MR0=AR, AR=SR1+AF+C;          {double precision MR0=L_z2}
            MR1=AR, AR=MR0+AY0;           {MR1=L_z2, round result }
            SR=LSHIFT AR (LO);            {and downshift for output}
            AR=MR1+C, SI=DM(I1,M1);       {Get next input sample}
            SR=SR OR ASHIFT AR (HI);
offset_comp:
            DM(I0,M1)=SR0, AR=MR0 AND AY1;{Store result, get next lsp}
            IF NOT CE JUMP AN1;
{        DM(L_z2_l)=MR0;}                  {Save values for next call}
{        DM(L_z2_h)=MR1;}
{        DM(z1)=MY1;}
         ENA AR_SAT;                       {Re-enable ALU saturation}
```

```
{_______________________energy calculation___________________________________}

         I0  = DM(speech_in);
         AF  = PASS 0, AX0 = DM(I0,M1);
         CNTR = analysis_window_length;
{        DO calc_energy UNTIL CE;}
AN2:        AR  = ABS AX0;
            SR  = ASHIFT AR BY -7 (HI);
calc_energy:AF  = SR1 + AF, AX0 = DM(I0,M1);
            IF NOT CE JUMP AN2;
         AR  = PASS AF;
         DM(frame_energy) = AR;

{_______________________zero crossing calculation____________________________}

         I0  = DM(speech_in);
         AF  = PASS 0, AX0 = DM(I0,M1);
         AR  = ABS AX0;             {set either POS or NEG}
         AX1 = 205;                 {temporary - improves scaling}
         AY0 = zcr_noise_threshold;
         CNTR = analysis_window_length;
{        DO calc_zcr UNTIL CE;}
AN3:        IF POS JUMP last_was_pos;
last_was_neg:
            AR  = AX0 - AY0, AX0 = DM(I0,M1);
            IF GE AF = AX1 + AF;
            JUMP calc_zcr;
last_was_pos:
            AR  = AX0 + AY0, AX0 = DM(I0,M1);
            IF LT AF = AX1 + AF;
calc_zcr:
            AR  = ABS AR;
            IF NOT CE JUMP AN3;
         AR = PASS AF;
         DM(frame_zcr) = AR;

{_______________________pre-emphasis filter__________________________________}

         MX0 = 0;
{        MX0=DM(mp);}               {Get saved value for mp}
         MY0=-28180;                {MY0 holds coefficient value}
         MX1=H#80;                  {These are used for biased}
         MR=MX1*MF (SS);            {rounding}
         SB=-4;                     {Maximum scale value}
         I0=DM(speech_in);          {In-place computation}
         CNTR=analysis_window_length;
{        DO pre_emp UNTIL CE;}
```

(listing continues on next page)

```
AN4:        MR=MR+MX0*MY0 (SS), AY0=DM(I0,M0);
            AR=MR1+AY0, MX0=AY0;
            SB=EXPADJ AR;                 {Check for maximum value}
pre_emp:    DM(I0,M1)=AR, MR=MX1*MF (SS); {Save filtered data}
            IF NOT CE JUMP AN4;
{        DM(mp)=MX0;}
         AY0=SB;                          {Get exponent of max value}
         AX0=4;                           {Add 4 to get scale value}
         AR=AX0+AY0;
         DM(spscaleauto)=AR;              {Save scale for later}

{_________________________hamming windowing_________________________________}

         I0 = DM(speech_in);
         I1 = I0; {output}
         I5 = ^hamming_dat;
         MX0 = DM(I0,M1);
         MY0 = PM(I5,M5);
         MX1 = H#80;
         MR  = MX1 * MF (SS);
         CNTR = analysis_window_length;
{        DO window_frame UNTIL CE;}
AN5:        MR  = MR + MX0 * MY0 (SS), MX0 = DM(I0,M1);
            MY0 = PM(I5,M5);
window_frame:
            DM(I1,M1) = MR1, MR = MX1 * MF (SS);
            IF NOT CE JUMP AN5;

{_______________________dynamic scaling_____________________________________}

         IF LE JUMP auto_corr;            {If 0 scale, only copy data}
         AF=PASS 1;
         AR=AF-AR;
         SI=16384;
         SE=AR;
         I0=DM(speech_in);
         I1=I0;                           {Output writes over the input}
         SR=ASHIFT SI (HI);
         AF=PASS AR, AR=SR1;              {SR1 holds temp for multiply}
         MX1=H#80;                        {Used for unbiased rounding}
         MR=MX1*MF (SS), MY0=DM(I0,M1);   {Fetch first value}
         CNTR=analysis_window_length;
{        DO scale UNTIL CE;}
AN6:        MR=MR+SR1*MY0 (SS), MY0=DM(I0,M1);  {Compute scaled data}
scale:      DM(I1,M1)=MR1, MR=MX1*MF (SS);      {Save scaled data}
            IF NOT CE JUMP AN6;
```

```
{________________________autocorrelation_______________________________________}

auto_corr:
         I1=DM(speech_in);          {This section of code computes}
         I5=I1;                     {the autocorr section for LPC}
         I2=analysis_window_length; {I2 used as down counter}
         I6=^spL_ACF;               {Set pointer to output array}
         CNTR=9;                    {Compute nine terms}
{        DO corr_loop UNTIL CE;}
AN7:        I0=I1;                  {Reset pointers for mac loop}
            I4=I5;
            MR=0, MX0=DM(I0,M1);    {Get first sample}
            CNTR=I2;                {I2 decrements once each loop}
{           DO data_loop UNTIL CE;}
AN8:        MY0=DM(I4,M5);
data_loop:        MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1);
                  IF NOT CE JUMP AN8;
            MODIFY(I2,M2);          {Decrement I2, Increment I5}
            MY0=DM(I5,M5);
            DM(I6,M5)=MR1;          {Save double precision result}
corr_loop:  DM(I6,M5)=MR0;          {MSW first}
            IF NOT CE JUMP AN7;

         I0=DM(speech_in);          {This section of code rescales}
         SE=DM(spscaleauto);        {the input data}
         I1=I0;                     {Output writes over input}
         SI=DM(I0,M1);
         CNTR=analysis_window_length;
{        DO rescale UNTIL CE;}
AN9:        SR=ASHIFT SI (HI), SI=DM(I0,M1);
rescale:    DM(I1,M1)=SR1;
            IF NOT CE JUMP AN9;

{________________________schur recursion_______________________________________}

set_up_schur:
         AY1 = ^spL_ACF; {in DM}
         MY1 = ^acf;
         M0  = ^r;
         CALL schur_routine;

{________________________output reflection coefficients_______________________}

transmit_lar:
         I1 = ^r;             {The quantized LAR values}
         CNTR=8;              {can now be sent}
         CALL xmit_data;      {Copy to the output buffer}
```

(listing continues on next page)

```
   {All the coded variables have been sent to xmit_buffer}

finish:  DIS AR_SAT;
         RTS;                       {Return to caller}

xmit_data:
         I0=DM(xmit_buffer);        {Copy coeffs to the output}
{        DO xmit UNTIL CE;}         {buffer}
AN10:       AX0=DM(I1,M1);
xmit:       DM(I0,M1)=AX0;
            IF NOT CE JUMP AN10;
         DM(xmit_buffer)=I0;
         RTS;                       {Return from Encoder}

{______________Encoder and Voice Activity Detector Subroutines_______________}

{  This section of code computes the reflection coefficients using the
   schur recursion }

schur_routine:
         I6=AY1;                          {This section of code prepares}
         AR=DM(I6,M5);                    {for the schur recursion}
         SE=EXP AR (HI), SI=DM(I6,M5);    {Normalize the autocorrelation}
         SE=EXP SI (LO);                  {sequence based on spL_ACF[0]}
         SR=NORM AR (HI);
         SR=SR OR NORM SI (LO);
         AR=PASS SR1;               {If spL_ACF[0] = 0, set r to 0}
         IF EQ JUMP zero_reflec;
         I6=AY1;
         I5=MY1;
         AR=DM(I6,M5);
         CNTR=9;                    {Normalize all terms}
{        DO set_acf UNTIL CE;}
AN11:       SR=NORM AR (HI), AR=DM(I6,M5);
            SR=SR OR NORM AR (LO), AR=DM(I6,M5);
set_acf:    DM(I5,M5)=SR1;
            IF NOT CE JUMP AN11;

         I5=MY1;                    {This section of code creates}
         I4=^k+7;                   {the k-values and p-values}
         I0=^p;
         AR=DM(I5,M5);              {Set P[0]=acf[0]}
         DM(I0,M1)=AR;
         CNTR=7;
{        DO create_k UNTIL CE;}     {Fill the k and p arrays}
```

```
AN12:        AR=DM(I5,M5);
             DM(I0,M1)=AR;
create_k:    DM(I4,M6)=AR;
             IF NOT CE JUMP AN12;
         AR=DM(I5,M5);
         DM(I0,M1)=AR;              {Set P[8]=acf[8]}

         I5=M0;                     {Compute r-values}
         I6=7;                      {I6 used as downcounter}
         SR0=0;
         SR1=H#80;                  {Used in unbiased rounding}
         CNTR=7;                    {Loop through first 7 r-values}
{        DO compute_reflec UNTIL CE;}
AN13:        I2=^p;                       {Reset pointers}
             I4=^k+7;
             AX0=DM(I2,M1);               {Fetch P[0]}
             AX1=DM(I2,M2);               {Fetch P[1]}
             MX0=AX1, AF=ABS AX1;         {AF=abs(P[1])}
             AR=AF-AX0;
             IF LE JUMP do_division;      {If P[0]<abs(P[1]), r = 0}
             DM(I5,M5)=SR0;               {Final r =0}
             JUMP compute_reflec;
do_division:
             CALL divide_routine;         {Compute r[n]=abs(P[1])/P[0]}
             AR=AY0, AF=ABS AX1;
             AY0=32767;
             AF=AF-AX0;                   {Check for abs(P[1])=P[0]}
             IF EQ AR=PASS AY0;           {Saturate if they are equal}
             IF POS AR=-AR;               {Generate sign of r[n]}
             DM(I5,M5)=AR;                {Store r[n]}
             MY0=AR, MR=SR1*MF (SS);
             MR=MR+MX0*MY0 (SS), AY0=AX0;  {Compute new P[0]}
             AR=MR1+AY0;
             DM(I2,M3)=AR;                {Store new P[0]}
             CNTR=I6;                     {One less loop each iteration}
{            DO schur_recur UNTIL CE;}
AN14:              MR=SR1*MF (SS), MX0=DM(I4,M4);
                   MR=MR+MX0*MY0 (SS), AY0=DM(I2,M2);
                   AR=MR1+AY0, MX1=AY0;    {AR=new P[m]}
                   MR=SR1*MF (SS);
                   MR=MR+MX1*MY0 (SS), AY0=MX0;
                   DM(I2,M3)=AR, AR=MR1+AY0; {Store P[m], AR=new K[9-m]}
schur_recur:       DM(I4,M6)=AR;           {Store new K[9-m]}
                   IF NOT CE JUMP AN14;
```

(listing continues on next page)

```
compute_reflec:
            MODIFY(I6,M6);          {Decrement loop counter (I6)}
            IF NOT CE JUMP AN13;
         I2=^p;                     {Compute r[8] outside of loop}
         AX0=DM(I2,M1);             {Using same procedure as above}
         AX1=DM(I2,M2);
         AF=ABS AX1;
         CALL divide_routine;
         AR=AY0, AF=ABS AX1;
         AY0=32767;
         AF=AF-AX0;
         IF EQ AR=PASS AY0;
         AF=ABS AX1;
         AF=AF-AX0;                 {The test for valid r is here}
         IF GT AR=PASS 0;           {r[8]=0 if P[0]<abs(P[1])}
         IF POS AR=-AR;
         DM(I5,M5)=AR;
         JUMP schur_done;

zero_reflec:
         AX0=0;                     {The r-values must be set to}
         I5=M0;                     {0 according to the recursion}
         CNTR=8;
{        DO zero_rs UNTIL CE;}
zero_rs:    DM(I5,M5)=AX0;
            IF NOT CE JUMP ZERO_RS;

schur_done:
         M0 = 0;
         RTS;

{__________________________Divide Subroutine_________________________________}

divide_routine:
         AY0=0;
         DIVS AF,AX0;
         CNTR=15;
{        DO div_loop UNTIL CE;}
div_loop:   DIVQ AX0;
            IF NOT CE JUMP DIV_LOOP;
         RTS;

.ENDMOD;
```

Listing 6.5 Frame Analysis Routine (ANALYZE.DSP)

```
.MODULE/RAM/BOOT=0      detect_endpoints;

.VAR/RAM/DM       word_start_flag;
.VAR/RAM/DM       ws_energy_thresh;
.VAR/RAM/DM       ws_zcr_thresh;
.VAR/RAM/DM       silence_time;

.VAR/RAM/DM       poss_start_flag;
.VAR/RAM/DM       ps_energy_thresh;
.VAR/RAM/DM       ps_zcr_thresh;

.VAR/RAM/DM       min_word_length;
.VAR/RAM/DM       threshold_time;
.VAR/RAM/DM       word_end_flag;
.VAR/RAM/DM       speech_count;

.GLOBAL           word_end_flag, word_start_flag, poss_start_flag;
.GLOBAL           threshold_time;
.GLOBAL           min_word_length;
.GLOBAL           ws_energy_thresh;
.GLOBAL           ws_zcr_thresh;
.GLOBAL           ps_energy_thresh;
.GLOBAL           ps_zcr_thresh;

.EXTERNAL         frame_energy, frame_zcr;

.ENTRY            find_endpoints;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{________________________Endpoint Detection Subroutine________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

find_endpoints:
            MX0 = 0;
            MX1 = 1;
            AX0 = DM(word_start_flag);
            AR  = PASS AX0;
            IF EQ JUMP find_word_start;

{===================== find end of word ====================================}

find_word_end:
            AX0 = DM(ps_energy_thresh);
            AX1 = DM(ps_zcr_thresh);
            CALL comp_energy_and_zcr;
            IF NE JUMP set_word_start;   {still speech, return to shell}
```

(listing continues on next page)

```
{_______________________no longer speech_____________________________________}

            AY0 = DM(silence_time);
            AR  = AY0 + 1;              { increment silence time }
            DM(silence_time) = AR;
            AY0 = DM(threshold_time);   { if threshold time exceeded,}
            AR  = AR - AY0;             { assume end of word }

{_______________________silence inside word__________________________________}

            IF LT JUMP inc_sp_count;    { returns to shell }

{_______________________end of word reached__________________________________}

end_of_word:
            AX0 = DM(speech_count);
            AY0 = DM(min_word_length);
            AR  = AY0 - AX0;

word_too_short:
            IF GT JUMP reset_vars;      { returns to shell }

word_length_ok:
            DM(word_start_flag) = MX0;
            DM(word_end_flag) = MX1;
            RTS;

{===================== find start of word ==================================}

find_word_start:
            AX0 = DM(ws_energy_thresh);
            AX1 = DM(ws_zcr_thresh);
            CALL comp_energy_and_zcr;
            IF NE JUMP set_word_start;  { returns to shell }

{_______________check for possible starting point____________________________}

not_word_start:
            AX0 = DM(ps_energy_thresh);
            AX1 = DM(ps_zcr_thresh);
            CALL comp_energy_and_zcr;
            IF EQ JUMP reset_vars;      { returns to shell }

{_______________possible starting point found________________________________}

poss_word_start:
            DM(poss_start_flag) = MX1;
            JUMP inc_sp_count;          { returns to shell }
```

```
{==============================================================================}

{============== set variables for word start and increment speech count =====}

{==============================================================================}

set_word_start:
            DM(word_start_flag) = MX1;
            DM(poss_start_flag) = MX0;
            DM(silence_time) = MX0;

inc_sp_count:
            AY0 = DM(speech_count);
            AR = AY0 + 1;
            DM(speech_count) = AR;
            RTS;

{==============================================================================}

{============== reset variables to find new starting endpoint ===============}

{==============================================================================}

reset_vars:
            DM(poss_start_flag) = MX0;
D           M(word_start_flag) = MX0;
            DM(word_end_flag) = MX0;
            DM(silence_time) = MX0;
            DM(speech_count) = MX0;
            RTS;

{==============================================================================}

{============== compare frame energy and zcr with thresholds ================}

{==============================================================================}

comp_energy_and_zcr:                     { inputs: AX0 = energy threshold }
            AY0 = DM(frame_energy);      {         AX1 = zcr threshold    }
            AY1 = DM(frame_zcr);
            AF  = PASS 0;
            AR  = AY0 - AX0;
            IF GT AF = PASS 1;           { test frame_energy }
            AR  = AY1 - AX1;
            IF GT AF = PASS 1;           { test frame_zcr }
            AR  = PASS AF;               { AR will be NE for (poss_) speech }
            RTS;

{==============================================================================}

.ENDMOD;
```

Listing 6.6 Endpoint Detection Routine (ENDPOINT.DSP)

6 Speech Recognition

```
{_____________________________________________________________________________

         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
______________________________________________________________________________

CONVERT.DSP

The purpose of the routines in this module is to derive different feature
representations from the LPC reflection coefficients.  Separate routines exist
to convert from reflection (k) to predictor (alpha) coefficients, from
predictor to cepstral (c) coefficients, and to weight and normalize the
cepstral vector.

For several of these conversion recursions, scaling to prevent overflows would
reduce the significance of results.  A pseudo-floating-point number format is
used to alleviate scaling concerns during processing.  The inputs are in 1.15
fixed-point format, floating-point is used in processing, and the results are
returned in 1.15 fixed-point format.  The pseudo-floating-point format has a
one word (16 bit) mantissa and a one word (16 bit) exponent, stored as
mantissa followed by exponent.

The floating-point routines are adapted from Applications Handbook, volume 1.
All of the routines have been optimized for this particular application.

A more detailed description of the algorithms implemented can be found in the
Application Note.
_____________________________________________________________________________}

.MODULE/RAM/BOOT=0      coefficient_conversion;
.VAR/DM/RAM i_odd_a[16];      { temporary buffers for storage of intermediate}
.VAR/DM/RAM i_even_a[16];     {    predictor coefficient values }
.VAR/DM/RAM cepstral_coeff[24];     { temporary storage of cepstral coeffs }
.VAR/DM/RAM temp_mant;              { temporary mantissa storage }
.VAR/DM/RAM temp_exp;               { temporary exponent storage }
.VAR/PM/RAM sqrt_coeff[5];          { used in square root approximation }

.INIT    sqrt_coeff : H#5D1D00, H#A9ED00, H#46D600, H#DDAA00, H#072D00;

.VAR/PM/RAM weighting_coeff[12];    { used to weight (window) cepstral }
                                    {    coeff for improved recognition }

{ weighting of ( 1 + 6*sin(pi*k/12) ) }
.INIT       weighting_coeff : 0X2EAE00, 0X492400, 0X5FDD00, 0X714D00,
                              0X7C4200, 0X7FFF00, 0X7C4200, 0X714D00,
                              0X5FDD00, 0X492400, 0X2EAE00, 0X124900;
```

```
{ weighting of ( 1 + 6.5*sin(pi*k/16) ) }
{.INIT   weighting_coeff :    0X000000, 0X3B8400, 0X4EB200, 0X5F8200,
                              0X6D4D00, 0X778E00, 0X7DDE00, 0X7FFF00,
                              0X7DDE00, 0X778E00, 0X6D4D00, 0X5F8200;
}

.CONST   cepstral_order = 12;

.ENTRY   convert_coeffs;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________Conversion Shell_____________________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{       required inputs:        I4  -> ^reflection_coefficients              }

convert_coeffs:
         I6  = I4;            { I6 points to output buffer }
         CALL k_to_alpha;     { convert from reflection to predictor }
         CALL alpha_to_cep;   {             predictor  to cepstral }
         CALL weight_cep;     { weight cepstral coefficients }
         CALL normalize_cep;  { normal coeffs to length of vector }
         CALL cep_to_fixed;   { convert back to fixed-point }
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________Convert from reflection to predictor coefficients____________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{       required inputs:        I4  -> ^reflection_coefficients              }

{____________________________________________________________________________}
{       floating point implementation                                        }

{ for each stage of this recursion:
                              I1 -> pointer to result buffer
                              I0 -> points to last data in previous
                                    results buffer
                              I2 -> points to start of previous
                                    results buffer                 }
```

(listing continues on next page)

```
k_to_alpha:
         M3  = -3;

         I1  = ^i_odd_a;                 { i = 1 }
         MY0 = DM(I4,M5);                { read k1 }
         DM(I1,M1) = MY0;                { a1(1) mantissa = k1 (M1 = 1) }
         DM(I1,M2) = 0;                  { a1(1) exponent = 0 (M2 = -1) }

         I0  = I1;
         I1  = ^i_even_a;                { i = 2 }
         MY0 = DM(I4,M5);                { read k2 }
         AX0 = 0;                        { k2 exponent = 0 }
         AX1 = MY0;                      { k2 mantissa }
         AY1 = DM(I0,M1);                { a1(1) mantissa }
         AY0 = DM(I0,M1);                { a1(1) exponent }
         CALL fpm;                       { floating-point multiply }
         AX0 = AR, AR = pass SR1;
         AX1 = AR;
         CALL fpa;                       { floating-point add }
         DM(I1,M1) = SR1;                { a1(2) = a1(1) + k2*a1(1) mantissa}
         DM(I1,M1) = AR;                 {                         exponent }
         DM(I1,M1) = MY0;                { a2(2) mantissa = k2 }
         DM(I1,M2) = 0;                  { a2(2) exponent = 0 }
            I0  = I1;

         I3  = 2;                        { used as stage counter }
         CNTR = 3;     { two recursions done. two per loop. 2+2*3=8 total }
{        DO conver_recur UNTIL CE;}
loopa:      I1  = ^i_odd_a;              { output pointer }
            I2  = ^i_even_a;             { input pointer }
            CNTR = I3;
            CALL recursion;
            MODIFY(I3,M1);               { increment stage counter }

            I1  = ^i_even_a;             { output pointer }
            I2  = ^i_odd_a;              { input pointer }
            CNTR = I3;
            CALL recursion;
conver_recur:
            MODIFY(I3,M1);               { increment stage counter }
            IF NOT CE JUMP loopa;

         M3 = 2;
         RTS;
{==============================================================================}
```

```
{________________________Conversion Recursion_________________________________}

{============================================================================}

{  floating point implementation   }

recursion:
         MY0 = DM(I4,M5);                 { read k }
{        DO recur_routine UNTIL CE;}
loopb:      AX0 = 0;                      { k exponent }
            AX1 = MY0;                    { k mantissa }
            AY1 = DM(I0,M1);              { a mantissa }
            AY0 = DM(I0,M3);              { a exponent }
            CALL fpm;                     { k * a }
            AX0 = AR, AR = pass SR1;
            AX1 = AR;
            AY1 = DM(I2,M1);
            AY0 = DM(I2,M1);
            CALL fpa;                     { a + k*a }
            DM(I1,M1) = SR1;              { write new result mantissa }
recur_routine:
            DM(I1,M1) = AR;               {                  exponent }
            IF NOT CE JUMP loopb;

         DM(I1,M1) = MY0;                 { ai(i) = ki mantissa }
         DM(I1,M2) = 0;                   {            exponent }
         I0 = I1;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________convert from predictor to cepstral coefficients_____________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

alpha_to_cep:M7  = -3;

   { c1 = -a1 }
         I5  = ^i_even_a;                 { holds predictor coefficients }
         I1  = ^cepstral_coeff;
         AX1 = DM(I5,M5);
         AR  = -AX1;                      { negate mantissa only }
         DM(I1,M1) = AR;                  { store c1 mantissa }
         AY0 = DM(I5,M5);
         DM(I1,M1) = AY0;                 { store c1 exponent }
```

(listing continues on next page)

```
{ c2 = - (a2 + 1/2*c1*a1) }
      I5  = ^i_even_a + 2;
      AX1 = DM(I5,M5);             { read a2 }
      AX0 = DM(I5,M7);
      DM(temp_mant) = AX1;         { preload with a2 mantissa }
      DM(temp_exp) = AX0;          { preload with a2 exponent }

      I1  = ^cepstral_coeff;
      AX1 = 0x4000;                { 1/2 in 1.15 format }
      CALL scale_n_sum;      {accumulates sum of products in temp_mant,exp}

      AY0 = AR, AR = -SR1;         { negate mantissa }
      DM(I1,M1) = AR;              { store c2 mantissa }
      DM(I1,M1) = AY0;             { store c2 exponent }

{ c3 = - (a3 + 2/3*c2*a1 + 1/3*c1*a2) }
      I5  = ^i_even_a + 4;
      I1  = ^cepstral_coeff;
      AX1 = DM(I5,M5);             { read a3 }
      AX0 = DM(I5,M7);
      DM(temp_mant) = AX1;         { preload with a3 mantissa }
      DM(temp_exp) = AX0;          { preload with a3 exponent }
      AX1 = 0x2AAA;                { 1/3 in 1.15 format }
      CALL scale_n_sum;

      AX1 = 0x5555;                { 2/3 in 1.15 format }
      CALL scale_n_sum;

      AY0 = AR, AR = -SR1;         { negate mantissa }
      DM(I1,M1) = AR;              { store c3 mantissa }
      DM(I1,M1) = AY0;             { store c3 exponent }

{ c4 = - (a4 + 3/4*c3*a1 + 1/2*c2*a2 + 1/4*c1*a3) }
      I5  = ^i_even_a + 6;
      I1  = ^cepstral_coeff;
      AX1 = DM(I5,M5);             { read a4 }
      AX0 = DM(I5,M7);
      DM(temp_mant) = AX1;
      DM(temp_exp) = AX0;
      AX1 = 0x2000;                { 1/4 in 1.15 format }
      CALL scale_n_sum;

      AX1 = 0x4000;                { 1/2 in 1.15 format }
      CALL scale_n_sum;

      AX1 = 0x6000;                { 3/4 in 1.15 format }
      CALL scale_n_sum;

      AY0 = AR, AR = -SR1;
      DM(I1,M1) = AR;              { store c4 mantissa }
      DM(I1,M1) = AY0;             { store c4 exponent }
```

```
{ c5 = - (a5 + 4/5*c4*a1 + 3/5*c3*a2 + 2/5*c2*a3 + 1/5*c1*a4) }
      I5  = ^i_even_a + 8;
      I1  = ^cepstral_coeff;
      AX1 = DM(I5,M5);                 { read a5 }
      AX0 = DM(I5,M7);
      DM(temp_mant) = AX1;
      DM(temp_exp) = AX0;
      AX1 = 0x1999;
      CALL scale_n_sum;

      AX1 = 0x3333;
      CALL scale_n_sum;

      AX1 = 0x4CCC;
      CALL scale_n_sum;

      AX1 = 0x6666;
      CALL scale_n_sum;

      AY0 = AR, AR = -SR1;
      DM(I1,M1) = AR;
      DM(I1,M1) = AY0;

{ c6 = - (a6 + 5/6*c5*a1+2/3*c4*a2+1/2*c3*a3+1/3*c2*a4+1/6*c1*a5) }
      I5  = ^i_even_a + 10;
      I1  = ^cepstral_coeff;
      AX1 = DM(I5,M5);                 { read a6 }
      AX0 = DM(I5,M7);
      DM(temp_mant) = AX1;
      DM(temp_exp) = AX0;
      AX1 = 0x1555;
      CALL scale_n_sum;

      AX1 = 0x2AAA;
      CALL scale_n_sum;

      AX1 = 0x4000;
      CALL scale_n_sum;

      AX1 = 0x5555;
      CALL scale_n_sum;

      AX1 = 0x6AAA;
      CALL scale_n_sum;

      AY0 = AR, AR = -SR1;
      DM(I1,M1) = AR;
      DM(I1,M1) = AY0;
```

(listing continues on next page)

```
{ c7 = -(a7 + 6/7*c6*a1+5/7*c5*a2+4/7*c4*a3+3/7*c3*a4+2/7*c2*a5+1/7*c1*a6) }
         I5  = ^i_even_a + 12;
         I1  = ^cepstral_coeff;
         AX1 = DM(I5,M5);                  { read a7 }
         AX0 = DM(I5,M7);
         DM(temp_mant) = AX1;
         DM(temp_exp) = AX0;

         AX1 = 0x1249;
         CALL scale_n_sum;

         AX1 = 0x2492;
         CALL scale_n_sum;

         AX1 = 0x36DB;
         CALL scale_n_sum;

         AX1 = 0x4924;
         CALL scale_n_sum;

         AX1 = 0x5B6D;
         CALL scale_n_sum;

         AX1 = 0x6DB6;
         CALL scale_n_sum;

         AY0 = AR, AR = -SR1;
         DM(I1,M1) = AR;
         DM(I1,M1) = AY0;

{c8=-(a8+7/8*c7*a1+3/4*c6*a2+5/8*c5*a3+1/2*c4*a4+3/8*c3*a5+1/4*c2*a6+1/8*c1*a7)}
         I5  = ^i_even_a + 14;
         I1  = ^cepstral_coeff;
         AX1 = DM(I5,M5);                  { read a8 }
         AX0 = DM(I5,M7);
         DM(temp_mant) = AX1;
         DM(temp_exp) = AX0;

         AX1 = 0x1000;
         CALL scale_n_sum;

         AX1 = 0x2000;
         CALL scale_n_sum;

         AX1 = 0x3000;
         CALL scale_n_sum;

         AX1 = 0x4000;
         CALL scale_n_sum;
         AX1 = 0x5000;
         CALL scale_n_sum;

         AX1 = 0x6000;
         CALL scale_n_sum;

         AX1 = 0x7000;
         CALL scale_n_sum;

         AY0 = AR, AR = -SR1;
         DM(I1,M1) = AR;
         DM(I1,M1) = AY0;

{c9=-(8/9*c8*a1+7/9*c7*a2+2/3*c6*a3
                  +5/9*c5*a4+4/9*c4*a5+1/3*c3*a6+2/9*c2*a7+1/9*c1*a8)}
         I5  = ^i_even_a + 14;
         I1  = ^cepstral_coeff;
         AX0 = 0;
         DM(temp_mant) = AX0;
         DM(temp_exp) = AX0;

         AX1 = 0x0E38;
         CALL scale_n_sum;

         AX1 = 0x1C71;
         CALL scale_n_sum;

         AX1 = 0x2AAA;
         CALL scale_n_sum;

         AX1 = 0x38E3;
         CALL scale_n_sum;

         AX1 = 0x471C;
         CALL scale_n_sum;

         AX1 = 0x5555;
         CALL scale_n_sum;

         AX1 = 0x638E;
         CALL scale_n_sum;

         AX1 = 0x71C7;
         CALL scale_n_sum;

         AY0 = AR, AR = -SR1;
         DM(I1,M1) = AR;
         DM(I1,M1) = AY0;
```

(listing continues on next page)

```
{c10=-(9/10*c9*a1+4/5*c8*a2+7/10*c7*a3
                  +3/5*c6*a4+1/2*c5*a5+2/5*c4*a6+3/10*c3*a7+2/10*c2*a8)}
         I5  = ^i_even_a + 14;
         I1  = ^cepstral_coeff + 2;
         AX0 = 0;
         DM(temp_mant) = AX0;
         DM(temp_exp) = AX0;

         AX1 = 0x1999;
         CALL scale_n_sum;

         AX1 = 0x2666;
         CALL scale_n_sum;

         AX1 = 0x3333;
         CALL scale_n_sum;

         AX1 = 0x4000;
         CALL scale_n_sum;

         AX1 = 0x4CCC;
         CALL scale_n_sum;

         AX1 = 0x5999;
         CALL scale_n_sum;

         AX1 = 0x6666;
         CALL scale_n_sum;

         AX1 = 0x7333;
         CALL scale_n_sum;

         AY0 = AR, AR = -SR1;
         DM(I1,M1) = AR;
         DM(I1,M1) = AY0;

{c11=-(10/11*c10*a1+9/11*c9*a2+8/11*c8*a3
            +7/11*c7*a4+6/11*c6*a5+5/11*c5*a6+4/11*c4*a7+3/11*c3*a8)}
         I5  = ^i_even_a + 14;
         I1  = ^cepstral_coeff + 4;
         AX0 = 0;
         DM(temp_mant) = AX0;
         DM(temp_exp) = AX0;

         AX1 = 0x22EB;
         CALL scale_n_sum;

         AX1 = 0x2E8B;
         CALL scale_n_sum;
```

```
        AX1 = 0x3A2E;
        CALL scale_n_sum;

        AX1 = 0x45D1;
        CALL scale_n_sum;

        AX1 = 0x5174;
        CALL scale_n_sum;

        AX1 = 0x5D17;
        CALL scale_n_sum;

        AX1 = 0x68BA;
        CALL scale_n_sum;

        AX1 = 0x745D;
        CALL scale_n_sum;

        AY0 = AR, AR = -SR1;
        DM(I1,M1) = AR;
        DM(I1,M1) = AY0;

{c12=-(11/12*c11*a1+5/6*c10*a2+3/4*c9*a3
                  +2/3*c8*a4+7/12*c7*a5+1/2*c6*a6+5/12*c5*a7+1/3*c4*a8)}
        I5  = ^i_even_a + 14;
        I1  = ^cepstral_coeff + 6;
        AX0 = 0;
        DM(temp_mant) = AX0;
        DM(temp_exp) = AX0;

        AX1 = 0x2AAA;
        CALL scale_n_sum;

        AX1 = 0x3555;
        CALL scale_n_sum;

        AX1 = 0x4000;
        CALL scale_n_sum;

        AX1 = 0x4AAA;
        CALL scale_n_sum;

        AX1 = 0x5555;
        CALL scale_n_sum;

        AX1 = 0x6000;
        CALL scale_n_sum;

        AX1 = 0x6AAA;
        CALL scale_n_sum;
```

(listing continues on next page)

```
         AX1 = 0x7555;
         CALL scale_n_sum;

         AY0 = AR, AR = -SR1;
         DM(I1,M1) = AR;
         DM(I1,M1) = AY0;

         M7  = 2;
         RTS;

{============================================================================}

{_______________scale the product and add to running sum_____________________}

{============================================================================}

{  required inputs:        AX1 -> scale value                               }

scale_n_sum:
         AX0 = 0;                   { scale factor has exponent = 0 }
         AY1 = DM(I1,M1);           { read cepstral coefficient mant, exp }
         AY0 = DM(I1,M1);           { now points to next coefficient }
         CALL fpm;                  { scale factor * cepstral coeff }
         AX0 = AR;
         AX1 = SR1;

         AY1 = DM(I5,M5);           { read predictor coefficient mant, exp }
         AY0 = DM(I5,M7);           { now points to previous coefficient }
         CALL fpm;                  { predictor coeff * (scal * cepstral) }
         AX0 = AR;
         AX1 = SR1;
         AY1 = DM(temp_mant);
         AY0 = DM(temp_exp);
         CALL fpa;                  { accumulate product with prev results }
         DM(temp_mant) = SR1;       { store new results }
         DM(temp_exp) = AR;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________weight cepstral coefficients_________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
```

```
weight_cep: I0  = ^cepstral_coeff;
            I5  = ^weighting_coeff;

            CNTR = cepstral_order;
{           DO weighting UNTIL CE;}
top_weight: AX1 = PM(I5,M5);
            AX0 = 0;                { weighting coeff exponent = 0 }
            AY1 = DM(I0,M1);
            AY0 = DM(I0,M2);
            CALL fpm;
            DM(I0,M1) = SR1;        { store results }
weighting:  DM(I0,M1) = AR;
            IF NOT CE JUMP top_weight;

            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________convert cepstral coefficients to 1.15 fixed point____________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

cep_to_fixed:I0  = ^cepstral_coeff;
            AY0 = 0;                { exponent bias value }
            CNTR = cepstral_order;
   {        DO scale_cep UNTIL CE;}
scale_loop: SI  = DM(I0,M1);
            AX0 = DM(I0,M1);
            CALL fixone;            { converts to 1.15 format }
scale_cep:  DM(I6,M5) = SR1;        { I6 points to result buffer }
            IF NOT CE JUMP scale_loop;
            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________normalize cepstral coefficients to length of vector__________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
```

(listing continues on next page)

```
normalize_cep:
            I0  = ^cepstral_coeff;
            AX0 = 0;
            DM(temp_mant) = AX0;
            DM(temp_exp) = AX0;
            CNTR = cepstral_order;
{           DO mag_sqd UNTIL CE;}
top_mag_sqd:
            AX1 = DM(I0,M1);
            AX0 = DM(I0,M1);
            AY1 = AX1;
            AY0 = AX0;
            CALL fpm;               { square cepstral coefficient }
            AX0 = AR;
            AX1 = SR1;
            AY1 = DM(temp_mant);
            AY0 = DM(temp_exp);
            CALL fpa;               { accumulate squared values }
            DM(temp_mant) = SR1;
mag_sqd:    DM(temp_exp) = AR;
            IF NOT CE JUMP top_mag_sqd;

            CALL sqrt;              {find square root of mag sqrd mantissa }
            AY0 = DM(temp_exp);     {find square root of mag sqrd exponent }
            AX0 = 0X1;
            AF  = AX0 AND AY0;
            IF EQ JUMP exp_sqrt;    { is exponent even or odd ? }

            AR  = AY0 + 1;          {exponent odd, must add one and scale }
            AY0 = AR;               { mantissa by 1/(SQRT(2)) }
            MY0 = 0X5A82;
            MR  = SR1 * MY0 (SS);
            SR1 = MR1;              { scaled mantissa }

exp_sqrt:   DM(temp_mant) = SR1;
            SI  = AY0;
            SR  = ASHIFT SI BY -1 (HI);   { divide exponent by two }
            DM(temp_exp) = SR1;

            I0  = ^cepstral_coeff;  { normalize all cepstral coefficients }
            CNTR = cepstral_order;  {    to length of cepstral vector }
{           DO normalization UNTIL CE;}
top_norm:   AX1 = DM(I0,M1);
            AX0 = DM(I0,M2);
            AY1 = DM(temp_mant);
            AY0 = DM(temp_exp);
            CALL fpd;               { floating-point divide (coeff/length) }
            DM(I0,M1) = SR1;
```

```
normalization:    DM(I0,M1) = AR;
                  IF NOT CE JUMP top_norm;
                  RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________floating point multiply______________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{
   Floating-Point Multiply
         Z = X * Y

   Calling Parameters
         AX0 = Exponent of X
         AX1 = Fraction of X
         AY0 = Exponent of Y
         AY1 = Fraction of Y
         MX0 = Excess Code

   Return Values
         AR = Exponent of Z
         SR1 = Fraction of Z
}

fpm:     MX0 = 0;                        { set exponent bias = 0 }
         AF=AX0+AY0, MX1=AX1;            {Add exponents}
         MY1=AY1;
         AX0=MX0, MR=MX1*MY1 (SS);       {Multiply mantissas}
         IF MV SAT MR;                   {Check for overflow}
         SE=EXP MR1 (HI);
         AF=AF-AX0, AX0=SE;              {Subtract bias}
         AR=AX0+AF;                      {Compute exponent}
         SR=NORM MR1 (HI);               {Normalize}
         SR=SR OR NORM MR0 (LO);
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________floating point addition______________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
```

(listing continues on next page)

```
{
   Floating-Point Addition
         z = x + y

   Calling Parameters
         AX0 = Exponent of x
         AX1 = Fraction of x
         AY0 = Exponent of y
         AY1 = Fraction of y

   Return Values
         AR = Exponent of z
         SR1 = Fraction of z
}

fpa:     AF=AX0-AY0;                      {Is Ex > Ey?}
         IF GT JUMP shifty;               {Yes, shift y}
         SI=AX1, AR=PASS AF;              {No, shift x}
         SE=AR;
         SR=ASHIFT SI (HI);
         JUMP add;
shifty:  SI=AY1, AR=-AF;
         SE=AR;
         SR=ASHIFT SI (HI), AY1=AX1;
         AY0=AX0;
add:     AR=SR1+AY1;                      {Add fractional parts}
         IF AV JUMP work_around;
         SE=EXP AR (HI);
         AX0=SE, SR=NORM AR (HI);         {Normalize}
         AR= AX0+AY0;                     {Compute exponent}
         RTS;
work_around:
         AX1 = 0X08;                      { work around for HIX }
         AY1 = ASTAT;
         AX0 = AR, AR = AX1 AND AY1;
         SR  = LSHIFT AR BY 12 (HI);
         SE  = 1;
         AR  = AX0;
         AX0 = SE, SR = SR OR NORM AR (HI);
         AR  = AX0 + AY0;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________floating point conversion____________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
```

```
{
   Convert two-word floating-point to 1.15 fixed-point

   Calling Parameters
         AX0 = exponent             [16.0 signed twos complement]
         AY0 = exponent bias        [16.0 signed twos complement]
         SI = mantissa              [1.15 signed twos complement]

   Return Values
         SR1 = fixed-point number   [1.15 signed twos complement] }

.ENTRY   fixone;
fixone:  AR=AX0-AY0;                {Compute unbiased exponent}
         IF GT JUMP overshift;      { positive exponent would
                                     overflow so saturate }
         SE=AR;
         SR=ASHIFT SI (HI);         {Shift fractional part}
         RTS;
overshift:  AR  = SI;
            AF  = PASS AR;
            ENA AR_SAT;
            AR  = AR + AF;          { saturate positive or negative }
            DIS AR_SAT;
            SR1 = AR;
            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________square root routine__________________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{
   Square Root
         y = sqrt(x)

   Calling Parameters
         SR1 = x in 1.15 format
         M5 = 1
         L5 = 0

   Return Values
         SR1 = y in 1.15 format
}

{ most of the error checking has been removed from this routine }
```

(listing continues on next page)

```
.CONST   base=H#0D49;

sqrt:    I5=^sqrt_coeff;                       {Pointer to coeff. buffer}
         MY0=SR1, AR=PASS SR1;
         IF EQ RTS;                            { if x=0 then y=0 }
         MR=0;
         MR1=base;                             {Load constant value}
         MF=AR*MY0 (RND), MX0=PM(I5,M5);       {MF = x**2}
         MR=MR+MX0*MY0 (SS), MX0=PM(I5,M5);    {MR = base + C1*x}
         CNTR=4;
         DO approx UNTIL CE;
            MR=MR+MX0*MF (SS), MX0=PM(I5,M5);
approx:     MF=AR*MF (RND);
         SR=ASHIFT MR1 BY 1 (HI);
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________floating point divide________________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{
   Floating-Point Divide
         z = x / y

   Calling Parameters
         AX0 = Exponent of x
         AX1 = Fraction of x
         AY0 = Exponent of y
         AY1 = Fraction of y

   Return Values
         AR = Exponent of z
         SR1 = Fraction of z
}

fpd:     MX0 = 0;
         SR0=AY1, AR=ABS AX1;
         SR1=AR, AF=ABS SR0;
         SI=AX1, AR=SR1-AF;                    {Is Fx > Fy?}
         IF LT JUMP divide;                    {Yes, go divide}
         SR=ASHIFT SI BY -1 (LO);              {No, shift Fx right}
         AF=PASS AX0;
         AR=AF+1, AX1=SR0;                     {Increase exponent}
         AX0=AR;
```

```
divide:  AF=AX0-AY0, AX0=MX0;
         MR=0;
         AR=AX0+AF, AY0=MR1;
         AF=PASS AX1, AX1=AY1;                  {Add bias}
         DIVS AF, AX1;                          {Divide fractions}
         DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
         DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
         DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
         MR0=AY0, AF=PASS AR;
         SI=AY0, SE=EXP MR0 (HI);
         AX0=SE, SR=NORM SI (HI);               {Normalize}
         AR=AX0+AF;                             {Compute exponent}
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

.ENDMOD;
```

Listing 6.7 Coefficient Conversion Routine (CONVERT.DSP)

```
{_____________________________________________________________________________

         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
______________________________________________________________________________

LIB_FUNC.DSP

The routines in this module perform two different operations on templates. A
routine is present that adds a currently stored unknown word into the template
library, updating the library catalog and catalog size in the process.

The other operation allows a library template to be played back through an
external speaker. Since this is (currently) only possible when the stored
features are the reflection coefficients, conditional assembly is used for
these routines - they can easily be removed.

The conditional assembly options and a description of each follows. At a
minimum, assembly must include:

         asm21 LIB_FUNC -cp

The other options are:  -Dplayback  allows playback of library templates
                                    if reflection coefficients are the
                                    stored features
                        -Dinit_lib  used to initialize the template library
                                    with data contained in the file
                                    library.dat
_____________________________________________________________________________}

.MODULE/RAM/BOOT=0      library_functions;

.ENTRY   put_in_library;

.VAR/DM/RAM catalog_size, library_catalog[30], next_catalog_entry; {32 total}
.VAR/PM/RoM template_library[14144];  { 14K-(32 from above)-(160 hamm coeff)}

.GLOBAL     catalog_size, library_catalog, template_library;
.GLOBAL     next_catalog_entry;

.EXTERNAL   library_feature_dimension;
.EXTERNAL   unknown_feature_dimension;
```

```
{............................................................................}
{ conditional assembly use -Dplayback }
#ifdef playback
.ENTRY      shuffle_play, play_library, play_single;

.VAR/DM/RAM current_selection;
.VAR/DM/RAM shuffle_pntr;

.EXTERNAL   output_template;
#endif {.....................................................................}
{............................................................................}
{ conditional assembly use -Dinit_lib }
#ifdef init_lib
.INIT       template_library : <library.dat>;
#endif {.....................................................................}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________Store Template in Library____________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     I0  - location of word to be stored
                              AX0 - length of word to be stored }

put_in_library:
            I1  = DM(next_catalog_entry);
            AY0 = DM(I1,M1);
            I5  = AY0;        { start of next library location }
            DM(I1,M1) = AX0;  { store template length (# of vectors) }
            DM(next_catalog_entry) = I1;

            M3  = DM(unknown_feature_dimension);
            CNTR = AX0;                   { AX0 holds length }
            DO store_vectors UNTIL CE;
                  I2  = I0;               { I0 points to word }
                  CNTR = DM(library_feature_dimension);
                  DO store_features UNTIL CE;
                        MX0 = DM(I2,M1);
store_features:         PM(I5,M5) = MX0;
store_vectors:    MODIFY(I0,M3);
            M3  = 2;
            AX0 = I5;
            DM(I1,M0) = AX0;        { set start of next template }

            AY0 = DM(catalog_size);
            AR  = AY0 + 1;          { increment catalog size }
            DM(catalog_size) = AR;

            RTS;
```

(listing continues on next page)

```
{............................................................................}
{ conditional assembly use -Dplayback }
#ifdef playback

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________Play All Library Templates___________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

play_library:
            AX0 = DM(catalog_size);
            AR  = PASS AX0;
            IF LE RTS;

            AR  = ^library_catalog;
            DM(current_selection) = AR; CNTR = AX0;
            DO playit UNTIL CE;
                  I3  = DM(current_selection);
                  CALL play_single;

playit:           NOP;
            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________Play Single Library Template_________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     I3  - location of library_catalog entry }

play_single:MX0 = DM(I3,M1);
            I5  = MX0;                    { start of library word }
            AX0 = DM(I3,M1);              { length }
            AX1 = DM(library_feature_dimension);
            DM(current_selection) = I3;

            CALL output_template;
            IMASK = 0;
            RTS;
```

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_____________________Play Ranked Library Templates in order_________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     AX0 - # of templates to play, in order
                              AY0 - shuffled order of templates }

shuffle_play:
            MR0 = DM(catalog_size);
            AR  = PASS MR0;
            IF LE RTS;

            DM(shuffle_pntr) = AY0;
            CNTR = AX0;
            DO play_shuffled UNTIL CE;
                  I1  = DM(shuffle_pntr);
                  AX0 = DM(I1,M1);
                  I3  = AX0;
                  DM(shuffle_pntr) = I1;

                  CALL play_single;

play_shuffled:    NOP;

            RTS;

#endif
{.............................................................................}

.ENDMOD;
```

Listing 6.8 Library Functions Routine (LIB_FUNC.DSP)

```
.MODULE/RAM/BOOT=0      coarse_distance;

.VAR/DM/RAM       unknown_addr, unknown_length;
.VAR/DM/RAM       candidate_distance1[30];
.VAR/DM/RAM       candidate_distance2[30];
.VAR/DM/RAM       candidate_distance3[30];
.VAR/DM/RAM       candidate_distance4[30];
.VAR/DM/RAM       current_compare, result_buffer;
.VAR/DM/RAM       distance_routine;

.EXTERNAL         catalog_size, library_catalog;
{.EXTERNAL        coarse_euclidean, fine_euclidean;}
.EXTERNAL         warp_words;
{.EXTERNAL        full_euclidean;}
.EXTERNAL         cepstral_projection;
.EXTERNAL         set_bank_select;
.EXTERNAL         inc_bank_select;
.EXTERNAL         show_bank;
.EXTERNAL         blank_hex_led;
.EXTERNAL         init_catalog;

.GLOBAL           candidate_distance1;
.GLOBAL           candidate_distance2;
.GLOBAL           candidate_distance3;
.GLOBAL           candidate_distance4;

{.ENTRY           coarse_compare, fine_compare, full_compare;}
.ENTRY            cepstral_compare;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                                        }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{coarse_compare: AX1 = ^coarse_euclidean;
         JUMP compare;
}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                                        }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{fine_compare: AX1 = ^fine_euclidean;
         JUMP compare;
}
```

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                            }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{full_compare: AX1 = ^full_euclidean;
         JUMP compare;
}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                            }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{        required inputs:     I0  -> location of unknown word
                              AX0 -> length of unknown word
}
cepstral_compare:
         DM(unknown_addr) = I0;
         DM(unknown_length) = AX0;

         AR  = 0;
         CALL set_bank_select;      { set memory bank }
         CALL show_bank;            { display memory bank }
         CALL init_catalog;         { initialize library catalog }

         { initialize results pointer, distance measure }
         I1  = ^candidate_distance1;
         AX1 = ^cepstral_projection;
         CALL compare;

         CALL inc_bank_select;      { set memory bank }
         CALL show_bank;            { display memory bank }
         CALL init_catalog;         { initialize library catalog }

         { initialize results pointer, distance measure }
         I1  = ^candidate_distance2;
         AX1 = ^cepstral_projection;
         CALL compare;

         CALL inc_bank_select;      { set memory bank }
         CALL show_bank;            { display memory bank }
         CALL init_catalog;         { initialize library catalog }

         { initialize results pointer, distance measure }
         I1  = ^candidate_distance3;
         AX1 = ^cepstral_projection;
         CALL compare;
```

(listing continues on next page)

```
         CALL inc_bank_select;            { set memory bank }
         CALL show_bank;                  { display memory bank }
         CALL init_catalog;               { initialize library catalog }

         { initialize results pointer, distance measure }
         I1  = ^candidate_distance4;
         AX1 = ^cepstral_projection;

         CALL compare;

         CALL knn_routine;

         CALL blank_hex_led;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                            }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{       required inputs:      I1  -> location of results buffer
                              AX1 -> pointer to distance measure
}

compare: AY0 = DM(catalog_size);
         AR  = PASS AY0;
         IF LE RTS;

         CNTR = AY0;
         DM(distance_routine) = AX1;
         AR  = ^library_catalog;
         DM(current_compare) = AR;
         DM(result_buffer) = I1;          { stored as msw, lsw for each }

         DO calc_dist UNTIL CE;
            I0  = DM(unknown_addr);
            AX0 = DM(unknown_length);
            I3  = DM(current_compare);
            I6  = DM(distance_routine);
            I5  = DM(result_buffer);

            CALL warp_words;

            I1  = DM(result_buffer);
            MODIFY(I1,M3);
            DM(result_buffer) = I1;

            I3  = DM(current_compare);
            MODIFY(I3,M3);
calc_dist:  DM(current_compare) = I3;

         RTS;
```

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{                                                                              }
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
knn_routine:
         AY0 = DM(catalog_size);
         AR  = PASS AY0;
         IF LE RTS;

         CNTR = AY0;
         I0  = ^candidate_distance1;
         I1  = ^candidate_distance2;
         I2  = ^candidate_distance3;
         I3  = ^candidate_distance4;
         I4  = ^candidate_distance1;
         DO compute_knn UNTIL CE;
            I5  = I0;
comp12:     AX1 = DM(I0,M1);
            AX0 = DM(I0,M2);
            AY1 = DM(I1,M1);
            AY0 = DM(I1,M2);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP comp23;
            I5  = I1;
            AX1 = AY1;
            AX0 = AY0;

comp23:     AY1 = DM(I2,M1);
            AY0 = DM(I2,M2);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP comp34;
            I5  = I2;
            AX1 = AY1;
            AX0 = AY0;

comp34:     AY1 = DM(I3,M1);
            AY0 = DM(I3,M2);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP store_best;
            I5  = I3;
            AX1 = AY1;
            AX0 = AY0;
```

(listing continues on next page)

```
store_best: SR1 = AX1;
            SR0 = AX0;
            AX1 = H#7FFF;
            DM(I5,M5) = AX1;
            AX0 = H#FFFF;
            DM(I5,M5) = AX0;

comp12_2:   AX1 = DM(I0,M1);
            AX0 = DM(I0,M1);
            AY1 = DM(I1,M1);
            AY0 = DM(I1,M1);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP comp23_2;
            AX1 = AY1;
            AX0 = AY0;

comp23_2:   AY1 = DM(I2,M1);
            AY0 = DM(I2,M1);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP comp34_2;
            AX1 = AY1;
            AX0 = AY0;

comp34_2:   AY1 = DM(I3,M1);
            AY0 = DM(I3,M1);
            AR  = AY0 - AX0;
            AR  = AY1 - AX1 + C -1;
            IF GE JUMP sum_top_2;
            AX1 = AY1;
            AX0 = AY0;

sum_top_2:  AY1 = SR1;
            AY0 = SR0;
            AR  = AX0 + AY0;
            ENA AR_SAT;
            AX0 = AR, AR  = AX1 + AY1 + C;
            DIS AR_SAT;
            DM(I4,M5) = AR;
compute_knn:DM(I4,M5) = AX0;

         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

.ENDMOD;
```

Listing 6.9 Word Comparison Routine (COMPLIB.DSP)

```
{___________________________________________________________________________

        Analog Devices Inc., DSP Division
        One Technology Way, Norwood, MA  02062
        DSP Applications Assistance:  (617) 461-3672
____________________________________________________________________________

RANKDIST.DSP

The routine in this module will rank the recognition results based upon
distances contained in the candidate distance buffer. All the words of the
catalog are ranked (up to 15).  The results are stored in two forms.
Candidate_order contains a pointer to the catalog entry for each word, and
order_number contains the number (zero to fourteen) each word has in the
library.  Both these buffers have the best recognition candidate first, then
the second best, and so on.

___________________________________________________________________________}

.MODULE/RAM/BOOT=0      rank_candidates_distances;

.VAR/DM/RAM       candidate_order[15], order_number[15];

.EXTERNAL         candidate_distance1;
.EXTERNAL         catalog_size;
.EXTERNAL         library_catalog;

.ENTRY            rank_candidates;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________Rank Candidates by Distance_________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

rank_candidates:
        MR0 = DM(catalog_size);
        AR  = PASS MR0;
        IF LE RTS;

        I1  = ^order_number;
        I2  = ^candidate_order;
        AX1 = H#7FFF;
        AX0 = H#FFFF;
        CNTR = MR0;                     { MR0 holds catalog_size }
        DO sort_candidates UNTIL CE;
           I0  = ^candidate_distance1;
           I3  = ^library_catalog;
           AF  = PASS 0;                { initialize library entry # }
           CNTR = MR0;
           DO least_dist UNTIL CE;
```

(listing continues on next page)

```
                    AY1 = DM(I0,M1);
                    AY0 = DM(I0,M1);
                    AR  = AY0 - AX0;
                    AR  = AY1 - AX1 + C - 1;
                    IF GE JUMP inc_word_count;

                    MX0 = I3;
                    DM(I2,M0) = MX0;  { store library_catalog pointer}

                    AR  = PASS AF;
                    DM(I1,M0) = AR;   { store library entry/order # }
                    AX1 = AY1;        { update threshold }
                    AX0 = AY0;
                    MR1 = I0;         { save till loop complete }

inc_word_count:     AF  = AF + 1;
least_dist:         MODIFY(I3,M3);

            MODIFY(I2,M1);            { pointer to candidate_order }
            MODIFY(I1,M1);            { pointer to order_number }
            I0  = MR1;
            MODIFY(I0,M2);
            MODIFY(I0,M2);
            AX1 = H#7FFF;
            AX0 = H#FFFF;             { effectively removes the least}
            DM(I0,M1) = AX1;          { distance found from the }
sort_candidates:
            DM(I0,M1) = AX0;          { candidate_distance buffer }

         AY0 = ^candidate_order;
         AY1 = DM(order_number);
         RTS;

.ENDMOD;
```

Listing 6.10 Word Ranking Routine (RANKDIST.DSP)

```
{_____________________________________________________________________________

         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
______________________________________________________________________________

WARPSHELL.DSP

The routine contained in this module will calculate the distance between a
library template and an unknown word. It accomplishes this using dynamic time
warping and a selected distance measure. The resulting distance is a double
precision value, stored as msw, lsw.

The routine first calculates several necessary values. As it begins the
warping, it branches into one of three code regions, depending on the
relationship between Xa and Xb.

Each of these regions is divided into two or three different warping sections.
Each section has a different set of warping constraints, so y_min and y_max
must be calculated differently. Pointers to the correct min/max routines are
initialized in each section.

Following this, the warp_section routine performs the actual Dynamic Time
Warping for the current section. Y values are calculated each time through the
loop. The x_coordinate is incremented each time through the loop.

Update_sums copies previous results, stored in the vector_distance_buffer,
into the intermediate sums buffer. A new column of distance is then calculated
and stored in the vector distance buffer. Finally, the time warping occurs,
and the x value is incremented for the next loop.

_____________________________________________________________________________}

.MODULE/RAM/BOOT=0      dtw_shell;

.VAR/DM/RAM       Xa, Xb;     { points where warping constraint changes }
.VAR/DM/RAM       M, N;       { # of vectors in library template,unknown word}
.VAR/DM/RAM       x_coordinate;     { current x value }
.VAR/DM/RAM       y_min_routine, y_max_routine;
.VAR/DM/RAM       library_word_start, x_vector_pntr;
.VAR/DM/RAM       old_y_min;
.VAR/DM/RAM       vector_distance_buffer[180];
                                   {filled with distance matrix col.}
.VAR/DM/RAM       intermediate_sum_buffer[192]; {minimum sum of possible
                                                 warping paths, stored as msw,
                                                 lsw, previous warping slope }
.VAR/DM/RAM       result_pntr, distance_measure;
```

(listing continues on next page)

```
.EXTERNAL   pre_Xa_y_max, post_Xa_y_max;
.EXTERNAL   pre_Xb_y_min, post_Xb_y_min;
.EXTERNAL   y_min, y_max, y_range;
.EXTERNAL   calc_y_range;
.EXTERNAL   build_vd_buff;
.EXTERNAL   compute_warp, update_sums;
.EXTERNAL   unknown_feature_dimension;

.GLOBAL     M, N;
.GLOBAL     x_vector_pntr;
.GLOBAL     old_y_min;
.GLOBAL     vector_distance_buffer, intermediate_sum_buffer;

.ENTRY      warp_words;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________use dynamic time warping to compare two words________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:
                        I3  -> location of word in template catalog
                        I0  -> location of unknown word
                        AX0 -> length (# of vectors) of unknown word
                        I6  -> pointer to distance routine
                        I5  -> location of results (msw, lsw)                }

warp_words:
         DM(x_vector_pntr) = I0;
         AY0 = AX0;
         AR  = AY0 - 1;            { axis starts at 0, not 1 }
         DM(N) = AR;

         MX0 = DM(I3,M1);          { read lib template location }
         DM(library_word_start) = MX0;
         AY0 = DM(I3,M2);          { read lib template length }
         AX0 = AR, AR  = AY0 - 1;  { y axis starts at 0, not 1 }
         DM(M) = AR;

         DM(distance_measure) = I6;
         DM(result_pntr) = I5;
```

```
{________________________check if ( 2*M-N < 3 )_______________________________}

         AY0 = AR;
         AR  = AR + AY0, AY1 = AX0;
         AR  = AR - AY1;
         MX0 = AR;                      { MX0 = (2 * M) - N }
         AY1 = 3;
         AR  = AR - AY1;
         IF LT JUMP cannot_warp;

{________________________check if ( 2*N-M < 2 )_______________________________}

         AF  = PASS AX0;
         AR  = AX0 + AF;
         AR  = AR - AY0;                { AR = (2 * N) - M }
         AY1 = 2;
         AF  = AR - AY1;
         IF LT JUMP cannot_warp;
         AY1 = AR;
         AR  = AR + AY1;
         MX1 = AR;                      { MX1 = 2*( (2*N) - M ) }

{________________________compute Xa, Xb_______________________________________}

         MY0 = H#2AAB;                  { MY0 = (1/3) that always rounds down }
         MR  = MX0 * MY0 (UU);
         DM(Xa) = MR1;                  { Xa = (1/3) * ( (2*M) - N) }
         AY1 = MR1, MR  = MX1 * MY0 (UU);
         DM(Xb) = MR1;                  { Xb = (2/3) * ( (2*N) - M) }

{________________________setup for warping____________________________________}

         AX0 = 0;
         DM(y_min) = AX0;
         AX0 = 1;
         DM(y_range) = AX0;
         DM(x_coordinate) = AX0;

{_______________calculate distance between first vectors (x,y = 0)___________}

         I0  = DM(x_vector_pntr);
         I4  = DM(library_word_start);
         I6  = DM(distance_measure);
         CALL (I6);
         I3  = ^vector_distance_buffer;
         DM(I3,M1) = SR1;
         DM(I3,M3) = SR0;
```

(listing continues on next page)

```
         I0  = DM(x_vector_pntr);
         M3  = DM(unknown_feature_dimension);
         MODIFY(I0,M3);                   { I0 points to next x vector }
         M3  = 2;
         DM(x_vector_pntr) = I0;

{_______________________find relationship of Xa, Xb__________________________}

         AX0 = DM(Xa);
         AY0 = DM(Xb);
         AR  = AX0 - AY0;                 { AR = Xa - Xb }
         IF GT JUMP Xa_gt_Xb;
         IF LT JUMP Xb_gt_Xa;

{====================== match words - Xa equal to Xb ========================}

Xa_eq_Xb:
         CNTR = DM(Xa);       {# of x vectors in first section is Xa }

         AX0 = ^pre_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^pre_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;
         AX0 = DM(Xb);
         AY0 = DM(N);
         AR  = AY0 - AX0;
         CNTR = AR;           { # of x vectors in final section is (N-Xb) }
         AX0 = ^post_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^post_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         JUMP write_result;

{====================== match words - Xb greater than Xa ====================}

Xb_gt_Xa:
         CNTR = DM(Xa);       {# of x vectors in first section is Xa }

         AX0 = ^pre_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^pre_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         AX0 = DM(Xa);
         AY0 = DM(Xb);
         AR  = AY0 - AX0;
         CNTR = AR;           {# of x vectors in middle section is (Xb-Xa) }
         AX0 = ^post_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^pre_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         AX0 = DM(Xb);
         AY0 = DM(N);
         AR  = AY0 - AX0;
         CNTR = AR;          {# of x vectors in final section is (N-Xb) }
         AX0 = ^post_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^post_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         JUMP write_result;

{=================== match words - Xa greater than Xb =====================}

Xa_gt_Xb:CNTR = DM(Xb);      {# of x vectors in first section is Xb }

         AX0 = ^pre_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^pre_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         AX0 = DM(Xb);
         AY0 = DM(Xa);
         AR  = AY0 - AX0;
         CNTR = AR;          {# of x vectors in middle section is (Xa-Xb) }
         AX0 = ^pre_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^post_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         AX0 = DM(Xa);
         AY0 = DM(N);
         AR  = AY0 - AX0;
         CNTR = AR;          {# of x vectors in final section is (N-Xa) }
         AX0 = ^post_Xa_y_max;
         DM(y_max_routine) = AX0;
         AX0 = ^post_Xb_y_min;
         DM(y_min_routine) = AX0;
         CALL warp_section;

         JUMP write_result;
```

(listing continues on next page)

```
{==================== cannot warp due to M, N =============================}

cannot_warp:
         I3  = ^vector_distance_buffer;
         AX0 = H#7FFF;              { sets word distance score to }
         DM(I3,M1) = AX0;           {   maximum double-precision }
         AX0 = H#FFFF;              {   value. }
         DM(I3,M1) = AX0;

{=================== write word distance result and return ================}

write_result:
         I3  = ^vector_distance_buffer;
         I0  = DM(result_pntr);
         AX0 = DM(I3,M1);
         DM(I0,M1) = AX0;
         AX0 = DM(I3,M1);
         DM(I0,M1) = AX0;

         RTS;
{==========================================================================}
{_______________________warp an entire section_______________________________}
{==========================================================================}

warp_section:
            DO section UNTIL CE;
            SR0 = DM(y_min);
            DM(old_y_min) = SR0;

            SR0 = DM(x_coordinate);       { current x value }
            I6  = DM(y_max_routine);
            CALL (I6);                    { calculate maximum y value }
            I6  = DM(y_min_routine);
            CALL (I6);                    { calculate minimum y value }

            CALL update_sums;             { copy previous results to sums buffer }

            CALL calc_y_range;            { calculate range of y for warp}

            I6  = DM(distance_measure);
            I5  = DM(library_word_start);

            CALL build_vd_buff;           {compute distance matrix column}

            CALL compute_warp;            {warp sums buffer into current column}
            I0  = DM(x_vector_pntr);
            M3  = DM(unknown_feature_dimension);
            MODIFY(I0,M3);                {next unknown(x) feature vector}
            M3  = 2;
            DM(x_vector_pntr) = I0;
            AY0 = DM(x_coordinate);
            AR  = AY0 + 1;                {increment x value counter}
section:    DM(x_coordinate) = AR;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

.ENDMOD;
```

Listing 6.11 Library Template/Word Distance Routine (WARPSHEL.DSP)

6 Speech Recognition

```
{_____________________________________________________________________________
         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
______________________________________________________________________________

YMINMAX.DSP
The routines in this module calculate the minimum, maximum, and range of the
ycoordinate for dynamic time warping, using Itakura warping constraints. The
points Xa and Xb are the x-coordinate locations where the constraining slope
changes for the upper and lower boundaries, respectively.
_____________________________________________________________________________}

.MODULE/RAM/BOOT=0      find_y_bounds;

.VAR/DM/RAM y_min, y_max, y_range;

.GLOBAL     y_min, y_max, y_range;

.EXTERNAL   N, M;

.ENTRY      pre_Xb_y_min, post_Xb_y_min;
.ENTRY      pre_Xa_y_max, post_Xa_y_max;
.ENTRY      calc_y_range;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{_______________________routines to find y_min, y_max________________________}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     SR0 -> current x_coordinate
}

{______________________________ y_min = .5 * x ______________________________}

pre_Xb_y_min:
            MR  = 0, MX0 = SR0;
            MY0 = H#4000;           { .5 in fixed-point format }
            MR0 = H#8000;
            MR  = MR + MX0 * MY0 (UU);
            DM(y_min) = MR1;
            RTS;

{______________________________ y_min = 2(x-N) + M __________________________}

post_Xb_y_min:
            AY0 = DM(N);
            AR  = SR0 - AY0;        { AR = x_coordinate - N }
            AY0 = AR;
            AR  = AR + AY0;         { AR = 2 * (x_coordinate - N) }
            AY1 = DM(M);
            AR  = AR + AY1;         { AR = 2 * (x_coordinate - N) + M }
            DM(y_min) = AR;
            RTS;

{______________________________ y_max = 2 * x ______________________________}

pre_Xa_y_max:
            AY0 = SR0;
            AR  = SR0 + AY0;
            DM(y_max) = AR;

            RTS;
{______________________________ y_max = .5*(x-N) + M ________________________}

post_Xa_y_max:
            AY0 = DM(N);
            AR  = SR0 - AY0;
            MR  = 0;
            MR1 = DM(M);
            MY0 = H#4000;           { .5 in fixed-point format }
            MR  = MR + AR * MY0 (SS);
            DM(y_max) = MR1;
            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________routine to find y_range______________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{______________________ y_range = y_max - y_min + 1 _________________________}

calc_y_range:
            AX0 = DM(y_max);
            AY0 = DM(y_min);
            AF  = AX0 - AY0;
            AR  = AF + 1;
            DM(y_range) = AR;
            RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

.ENDMOD;
```

Listing 6.12 Y Coordinate Range Routine (YMINMAX.DSP)

6 Speech Recognition

```
{___________________________________________________________________________

         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
____________________________________________________________________________

TIMEWARP.DSP

Two distinct routines are contained in this module, both dealing with dynamic
time warping. The first update_sums copies the current vector_distance_buffer
contents into the intermediate_sum_buffer. The future illegal warping paths
are removed by saturating the boundary distances before and after copying,
using the y_offset value. y_offset measures the difference between the minimum
y value of two adjacent columns.

The second routine, compute_warp, will perform the dynamic time warping
between two columns. One column is the intermediate_sum_buffer, which contains
the results of all previous warps. The other column is the
vector_distance_buffer, which contains a column of the distance matrix. A
column consists of the distances between a single unknown and many library
template feature vectors.

The sum buffer is warped into the distance buffer. The previous warping path
is examined in each case to prevent an illegal warp, and the accumulated sums
are stored into the distance buffer.

The y_offset is the difference (in the y direction) between the y_min of the
(x)th column and y_min (stored as old_y_min) of the (x-1)th column.
___________________________________________________________________________}

.MODULE/RAM/BOOT=0      dynamic_time_warping;

.EXTERNAL         intermediate_sum_buffer, vector_distance_buffer;
.EXTERNAL         y_min, old_y_min, y_range;

.ENTRY            compute_warp;
.ENTRY            update_sums;

{==========================================================================}
{==========================================================================}

{_______________________move vector_distance to intermediate_sum____________}

{==========================================================================}
{==========================================================================}
```

```
update_sums:
         I1  = ^intermediate_sum_buffer;
         AX1 = H#7FFF;
         AX0 = H#FFFF;
         DM(I1,M1) = AX1;           { initialize D(x-1,y-2) }
         DM(I1,M3) = AX0;
         DM(I1,M1) = AX1;           { initialize D(x-1,y-1) }
         DM(I1,M3) = AX0;           { leaves I1 pointing to D(x-1,y) }

         AR  = DM(y_min);
         AY0 = DM(old_y_min);
         AR  = AR - AY0;            { AR is y_offset }

         AY1 = -3;
         AY0 = -6;
         AF  = PASS 1, MR0 = AR;    { MR0 is y_offset }
         AR  = PASS 0;
         AF  = AF - MR0;
         IF LT AR = PASS AY0;
         IF EQ AR = PASS AY1;       { now AR holds real offset value }

         M3  = AR;
         MODIFY(I1,M3);             { now I1 points to }
         M3  = 2;                   { ( D(x-1,y) + real offset value ) }

         I5  = ^vector_distance_buffer;
         CNTR = DM(y_range);
         DO copy_sum UNTIL CE;
            CNTR = 3;
            DO copy_parts UNTIL CE;
                        AY0 = DM(I5,M5);
copy_parts:             DM(I1,M1) = AY0;
copy_sum:   NOP;

         DM(I1,M1) = AX1;           { initialize D(x-1,y_max+1) }
         DM(I1,M3) = AX0;
         DM(I1,M1) = AX1;           { initialize D(x-1,y_max+2) }
         DM(I1,M3) = AX0;

         RTS;
```

(listing continues on next page)

```
{==============================================================================}
{==============================================================================}

{_______________________warp one column in x dimension_______________________}

{==============================================================================}
{==============================================================================}

{        required inputs:           y_range = (y_max - y_min + 1)           }

{_______________________setup________________________________________________}

compute_warp:
         M0  = -5;
         M1  = -4;
         M2  =  1;
         M3  =  8;
         M7  =  2;
         I5  = ^vector_distance_buffer;
         I1  = ^intermediate_sum_buffer + 6;    { points to D(x-1,y) }

         CNTR = DM(y_range);
         DO warp_buffer UNTIL CE;

{_______________________compare D(x-1,y-1), D(x-1,y)_________________________}

compare_warp_1:
            AF  = PASS 0, AX1 = DM(I1,M2);      {init warp_value in AF }

            AX0 = DM(I1,M2);                    { read D(x-1,y) }

            AR  = DM(I1,M0);                    { read old_warp_value }

            AY1 = DM(I1,M2);
            AR  = PASS AR, AY0 = DM(I1,M1);     { read D(x-1,y-1) }
            IF NE JUMP do_comparison;           {jump if old_warp_value NE 0}

            AR  = DM(y_range);                  { if old_warp_value = 0, only  }

            AF  = PASS AR;                      { allow consecutive warps of 0 }

            AR  = CNTR;                         { when it's the first warp of  }

            AR  = AR - AF;                      { the column                   }

            IF NE JUMP set_warp_1;
            AF  = PASS 0;                       { reset state }
```

```
do_comparison:
            AR  = AX0 - AY0;
            AR  = AX1 - AY1 + C - 1;
            IF LT JUMP compare_warp_2;

set_warp_1:
            AF  = PASS 1, AX1 = AY1;           { change warp_value }
            AX0 = AY0;                         { move D(x-1,y-1) to AXn }

{_______________compare D(x-1,y-2), minimum[D(x-1,y),D(x-1,y-1)]____________}

compare_warp_2:
            AY1 = DM(I1,M2);
            AY0 = DM(I1,M3);                   { read D(x-1,y-2) }
            AR  = AX0 - AY0;
            AR  = AX1 - AY1 + C - 1;
            IF LT JUMP compute_sum;

set_warp_2: AX1 = AY1;
            AY1 = 2;                           { change warp_value }
            AF  = PASS AY1, AX0 = AY0;         { move D(x-1,y-2) to AXn }

{_______compute sum of d(x,y), minimum[D(x-1,y),D(x-1,y-1),D(x-1,y-2)]_______}

compute_sum:AY1 = DM(I5,M5);
            AY0 = DM(I5,M4);                   { read d(x,y) }
            AR  = AX0 + AY0;

            ena ar_sat;

            DM(I5,M6) = AR, AR  = AX1 + AY1 + C; { write D(x,y) lsw }

            dis ar_sat;

{           if av trap;}
            DM(I5,M7) = AR, AR  = PASS AF;     { write D(x,y) msw }
warp_buffer:DM(I5,M5) = AR;                    { write D(x,y) warp_value }

{_______________________restore state________________________________________}

         M0  = 0;
         M1  = 1;
         M2  = -1;
         M3  = 2;
         M7  = 2;
         RTS;

{===========================================================================}

.ENDMOD;
```

Listing 6.13 Dynamic Time Warping Routine (TIMEWARP.DSP)

```
{___________________________________________________________________________
         Analog Devices Inc., DSP Division
         One Technology Way, Norwood, MA  02062
         DSP Applications Assistance:  (617) 461-3672
____________________________________________________________________________

VECTDIST.DSP

This routine will calculate the distances necessary to fill one column of the
distance matrix. It uses a single vector of the unknown word (x dimension) and
the correct range of the library template vectors (y dimension). The resulting
distances are stored in the vector_distance_buffer.

There are two implemented distortion measures, the full euclidean and the
cepstral projection. A pointer to the selected measure is passed to this
module in I6.  Additional distortion measures can easily be added, following
the structure of the current measures.
___________________________________________________________________________}

.MODULE/RAM/BOOT=0      build_vector_distance;

.EXTERNAL         y_min, y_range;
.EXTERNAL         x_vector_pntr, vector_distance_buffer;
.EXTERNAL         library_feature_dimension;

.ENTRY            build_vd_buff;
.ENTRY            full_euclidean;
.ENTRY            cepstral_projection;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________calculate distance matrix column_____________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     I5  -> start of library template
                              I6  -> start of distance measure routine}

{_______________calculate offset from start of library template______________}

build_vd_buff:
         MX0 = DM(y_min);
         SI  = DM(library_feature_dimension);
         SR  = ASHIFT SI BY -1 (HI);      { (# of features)/2 }
         MY0 = SR1;
         MR  = MX0 * MY0 (UU);            { 2 * y_min * (# features)/2 }

         M7  = MR0;
         MODIFY(I5,M7);             { I5 now points to y_min feature vector}
```

```
{_______________________setup________________________________________________}

         M7  = DM(library_feature_dimension);
         I3  = ^vector_distance_buffer;
         CNTR = DM(y_range);

{______________________calculate vector distances and store__________________}

         DO build_buffer UNTIL CE;
            I0  = DM(x_vector_pntr);      { location of unknown vector }
            I4  = I5;                     { location of library template vector }

            CALL (I6);                    { calls distance measure routine }

            DM(I3,M1) = SR1;              { store distance msw }
            DM(I3,M3) = SR0;              { store distance lsw, skip warp_value }

build_buffer:
            MODIFY(I5,M7);          { I5 points to next library template vector }

{_______________________reset state and return_______________________________}

         M7  = 2;
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{_______________________distance measure routines____________________________}

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

{        required inputs:     I0  -> start of DM vector
                              I4  -> start of PM vector}

{====================== full euclidean distance =============================}

full_euclidean:
         CNTR = 12;
         MR  = 0, AX0 = DM(I0,M1), AY0 = PM(I4,M5);
         DO full_sumdiffsq UNTIL CE;
            AR  = AX0 - AY0, AX0 = DM(I0,M1);         { calculate difference }

            MY0 = AR;
```

(listing continues on next page)

```
full_sumdiffsq:
            MR  = MR + AR * MY0 (SS), AY0 = PM(I4,M5); {accumulat difsq}

         SR  = ASHIFT MR2 BY 5 (HI);      { leaves seven bits for warping}
         SR  = SR OR LSHIFT MR1 BY -11 (HI);
         SR  = SR OR LSHIFT MR0 BY -11 (LO);
         RTS;

{====================== cepstral projection distance ======================}

cepstral_projection:
         CNTR = 12;
         MR  = 0, MX0 = DM(I0,M1);
         MY0 = PM(I4,M5);
         DO dot_product UNTIL CE;         {calculates negative of dot product}

            MR  = MR - MX0 * MY0 (SS), MX0 = DM(I0,M1);
dot_product:
            MY0 = PM(I4,M5);

         SR  = ASHIFT MR2 BY 5 (HI);      { leaves seven bits for warping}
         SR  = SR OR LSHIFT MR1 BY -11 (HI);
         SR  = SR OR LSHIFT MR0 BY -11 (LO);
         RTS;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}

.ENDMOD;
```

Listing 6.14 Vector Distance Routine (VECTDIST.DSP)

```
{_________________________________________________________________________}
{ DISBOOT.DSP                                               3-10-90     }
{                                                                       }
{_________________________________________________________________________}
.MODULE/RAM/BOOT=0      display;
.PORT                   led_and_bank;
.PORT                   display_base;
.INCLUDE                <vocab.h>;
.VAR/DM                 bank_select;            {stored PM EPROM bank, 1 of 4}
.VAR/DM                 hex_led;                {stores value of hex led}
.VAR/DM/STATIC          digit_count;            {passed to boot page 2}
.VAR/DM/STATIC          phone_number[16];       {passed to boot page 2}
.GLOBAL                 digit_count, phone_number;
.VAR/DM/STATIC          long_distance_flag;     {passed to boot page 2}
.GLOBAL                 long_distance_flag;
.VAR/DM                 timed_pntr;

.ENTRY   display_digit;
.ENTRY   add_a_digit;
.ENTRY   display_text;
.ENTRY   clear_display;
.ENTRY   display_number;
.ENTRY   set_local_call;
.ENTRY   set_long_distance;
.ENTRY   show_bank;
.ENTRY   inc_bank_select;
.ENTRY   set_bank_select;
.ENTRY   display_numpls;
.ENTRY   display_dial;
.ENTRY   reset_display;
.ENTRY   timed_display;
.ENTRY   reset_timed;
.ENTRY   blank_hex_led;
{_________________________________________________________________________}
{           Wait_Some                     @ 12.28 MHz                   }
{ Count = (desired time in sec)/(5*cycletime).                          }
{ AY0 = lsw of count                                                    }
{ AY1 = msw of count                                                    }
{ alters: AX0,AY0,AX0,AR                                                }
```

(listing continues on next page)

```
wait_four:        CALL wait_two;           {wait four seconds}
wait_two:         CALL wait_one;           {wait two seconds}
wait_one:         CALL wait_half;          {wait one second}
wait_half:        CALL wait_quarter;       {half second}
wait_quarter:     CALL wait_eighth;        {quarter second}
wait_eighth:      CALL wait_sixteenth;     {eighth second}
wait_sixteenth:   CALL wait_thirtysec;     {sixteenth second}
wait_thirtysec:   AY0=0X2c00;              {lsw of count for 1/32 sec}
                  AY1=0X0001;              {msw of count for 1/32 sec}
wait_some:        AX0=0;                   {for borrow}
time_loop:              AR=AY0-1;
                        AY0=AR, AR=AY1-AX0+C-1;
                        AY1=AR;
                        AR=AR OR AY0;
                        IF NE JUMP time_loop;
            RTS;
{________________________________________________________________________}
{                       Display_Text                                     }
{ I4 = ^ascii text buffer in PM                                          }
{ Format of text buffer:<# characters, ascii data;>                      }
{ alters: I4,L4,I2,L2,AR,AY0,AY1                                         }

display_text:
            CALL clear_display;
            L4=0;
            AY0=PM(I4,M5);                 {get # characters}
            CALL display_spaces;           {display leading spaces}
            CNTR=AY0;                      {#characters to display}
char_loop:        AR=PM(I4,M5);            {get character}
                  CALL disp_char;          {display one character}
                  IF NOT CE JUMP char_loop;
            RTS;

{________________________________________________________________________}
{                 Display One Character                                  }
{ AR = ascii character                                                   }
{ I2 = display pointer, decremented by one                               }
{ alters: AY0,AR,I2                                                      }

disp_char:  AY1=0x0080;
            AR=AR OR AY1;                        {WR high}
            DM(I2,M0)=AR, AR=AR XOR AY1;
            DM(I2,M0)=AR;                        {WR low}
            AR=AR OR AY1;
            NOP;
            DM(I2,M2)=AR;                        {WR high}
            RTS;
```

```
{______________________________________________________________________}
{           Clear the ASCII display with N spaces                      }
{ CNTR = number of spaces                                              }
{ I2 = returned with the current characters location                   }
{ alters: I2,L2,AR,AY0                                                 }

clear_display:    AY0 = -16;                    {Entry to clear entire display}

display_spaces:   I2=^display_base + 15;        {Entry to clear leading spaces}
                  L2=0;
                  AR=16;
                  AR=AR-AY0;                    {center the word}
                  IF LE JUMP spaces_done;
                  SR=LSHIFT AR BY -1 (LO);      {SR0=(16-#characters)/2}
                  AR  = PASS SR0;
                  IF LE JUMP spaces_done;
                  CNTR=SR0;
                  AR=0x0020;                    {space}
clear_loop:             CALL disp_char;
                        IF NOT CE JUMP clear_loop;
spaces_done:      RTS;

{______________________________________________________________________}
{                 Display Number                                       }
{ Displays digit_count characters from digit buffer in DM.             }
{ Format of text buffer:<# characters, ascii data;>                    }
{ alters: I4,L4,I2,L2,AR,AY0,AY1                                       }
{ Modified to inset dash after 3 digits.                               }

display_number:   CALL clear_display;
                  I4=^phone_number;
                  L4=0;
                  AY0=DM(digit_count);          {get # digits}
                  CALL display_spaces;          {display leading spaces}
                  CNTR=AY0;                     {#characters to display}
                  AF=PASS 0;                    {counts digits}
                  AY0=0x30;                     {ascii 0 offset}
digd_loop:              AX0=3;
                        AR=AX0-AF;
                        IF EQ CALL display_dash;
                        AF=AF+1;
                        AR=DM(I4,M5);           {get digit}
                        AR=AR+AY0;              {offset for ascii}
                        CALL disp_char;         {display one character}
                        IF NOT CE JUMP digd_loop;
            RTS;

display_dash:
            AR=0x2d;                            {ascii dash}
            CALL disp_char;
            RTS;
```

(listing continues on next page)

```
{_____________________________________________________________________________}
{                         Display a Digit                                     }
{ AY1 = index of digit to display                                             }

display_digit:
            AR=^word_catalog;
            AR=AR+AY1;
            I4=AR;
            AR  = PM(I4,M4);
            I4  = AR;
            CALL display_text;
            CALL wait_quarter;
            CALL wait_sixteenth;
            RTS;

{_____________________________________________________________________________}
{                   Add a Digit to the Phone Number                           }
{ Adds a digit to phone_number and increments digit_count                     }
{ AY1 = digit to add                                                          }
{ alters: AY0,AR,I4                                                           }

add_a_digit:
            AY0=DM(digit_count);
            AR=AY0+1;
            AY0=12;
            AF=AR-AY0;
            IF GE RTS;
            DM(digit_count)=AR;
            AY0=^phone_number;
            AR=AR+AY0;
            I4=AR;
            MODIFY(I4,M6);          {^ + # digits - 1 to get address}
            AR  = 10;
            AR  = AR - AY1;
            IF NE AR = PASS AY1;    { (oh) is tenth in the list }
            DM(I4,M4) = AR;
            RTS;

{_____________________________________________________________________________}
{                         Set Local Call                                      }

set_local_call:
            AR=0;
            DM(long_distance_flag)=AR;
            RTS;

{_____________________________________________________________________________}
{                   Set Long Distance Call                                    }

set_long_distance:
            AR=1;
            DM(long_distance_flag)=AR;
            RTS;
```

```
{________________________________________________________________________}
{                        Blank the HEX LED                               }
{ alters: SR0                                                            }

blank_hex_led:
            SR0=0x0010;
            CALL set_hex_led;
            RTS;

{________________________________________________________________________}
{                        Display to HEX LED                              }
{ SR0 = Hex value to display                                             }
{ alters: AR, SR                                                         }

show_bank:  SR0=DM(bank_select);          {entry to display bank}
set_hex_led:dm(hex_led)=SR0;              {normal entry}
            AR=DM(bank_select);
            SR=SR OR LSHIFT AR BY 8 (LO);
            DM(led_and_bank)=SR0;
            RTS;

{________________________________________________________________________}
{                  Set the PM EPROM Bank Select                          }
{ AR = bank value                                                        }
{ alters: SR,AY0                                                         }

inc_bank_select:
            AY0=DM(bank_select);          {entry to inc bank}
            AR=AY0+1;
            AY0 = 3;
            AR  = AR AND AY0;

set_bank_select:
            DM(bank_select)=AR;           {normal entry}
            SR0=DM(hex_led);
            SR=SR OR LSHIFT AR BY 8 (LO);
            DM(led_and_bank)=SR0;
            RTS;

{________________________________________________________________________}
{                  display particular text                               }

display_numpls:
            I4  = ^num_;
            JUMP display_text;

display_dial:
            I4  = ^dial;
            JUMP display_text;
```

(listing continues on next page)

```
{_________________________________________________________________________}
{                  initialize display variables                           }
reset_display:
            AR  = 0;
            DM(bank_select) = AR;
            CALL show_bank;
reset_timed:
            CALL clear_display;
            AR  = H#FF;
            DM(0X3FFB) = AR;                   { TSCALE }
            I4  = ^timed_catalog;
            AR  = PM(I4,M5);
            DM(0X3FFC) = AR;                   { TCOUNT }
            AX0 = PM(I4,M5);
            AR  = PM(I4,M5);
            DM(0X3FFD) = AR;                   { TPERIOD }
            DM(timed_pntr) = I4;
            I4  = AX0;
            CALL display_text;
            RTS;
{________________________________________________________________________}
{                  display opening on timer interrupts                   }
timed_display:
            ENA SEC_REG;
            MR1 = I4;                          { save state }
            MR0 = L4;                          { save state }
            I4  = DM(timed_pntr);
            L4  = 48;
            MY0 = PM(I4,M5);
            MY1 = PM(I4,M5);
            DM(timed_pntr) = I4;
            L4  = 0;
            DM(0X3FFD) = MY1;                  { TPERIOD }
            I4  = MY0;
            MY0 = I2;                          { save state }
            MY1 = L2;                          { save state }
            CALL display_text;
            I4  = MR1;                         { restore state }
            L4  = MR0;                         { restore state }
            I2  = MY0;                         { restore state }
            L2  = MY1;                         { restore state }
            RTI;
{________________________________________________________________________}
.ENDMOD;
```

Listing 6.15 Display Driver Routine (DEMOBOX.DSP)

```
{  DTMF Signal Generator

   ADSP-2101 EZ-LAB demonstration

   Analog Devices, Inc.
   DSP Division
   P.O.Box 9106
   Norwood, MA 02062
}

.module/boot=1    DTMF_Dialer;
.ENTRY   eight_khz;
.ENTRY   new_digit;
.ENTRY   dm_inits;
.ENTRY   make_tones;
.ENTRY   make_silence;

{ sine routine variables}
.VAR/PM  sin_coeff[5];
.INIT    sin_coeff: H#324000, H#005300, H#AACC00, H#08B700, H#1CCE00;

{ dynamic scratchpad variables }
.var     hertz1, hertz2,      { row and col frequency in Hertz }
         sum1, sum2,          { row and col phase accumulators }
         sin1, sin2;          { returned values from calling sin }

.VAR/DM  maketones_or_silence;

{ fixed variables to be loaded from booted PM }
.var     scale,               { attenuation of each sine before summing }
         hz_list[32];         { lookup table for digit row,col freqs }

{ NOTE *** put all fixed DM inits into PM and copy over into DM !! *** NOTE }

.const   PM_copy_length=33;

.var/pm  PM_scale, PM_hz_list[32];

{altered so that A == dial tone}
.init    PM_scale:      h#FFFF;
.init    PM_hz_list[00]:h#03AD,h#0538,h#02B9,h#04B9,h#02B9,h#0538,h#02B9,h#05C5;
.init    PM_hz_list[08]:h#0302,h#04B9,h#0302,h#0538,h#0302,h#05C5,h#0354,h#04B9;
.init    PM_hz_list[16]:h#0354,h#0538,h#0354,h#05C5,h#01B8,h#015E,h#0302,h#0661;
.init    PM_hz_list[24]:h#0354,h#0661,h#03AD,h#0661,h#03AD,h#04B9,h#03AD,h#05C5;
```

(listing continues on next page)

```
{——————————————————————————————————————————————————————————————————}
{                 Eight KHz Interrupt Routine                      }

eight_khz:
         ENA SEC_REG;
         AR=DM(maketones_or_silence);     {0 = quite, 1 = maketones}
         AR=PASS AR;
         IF EQ JUMP quiet;

         se=dm(scale);
tone1:   ay0=dm(sum1);
         si=dm(hertz1);                   { freq stored as Hz in DM }
         sr=ashift si by 3 (hi);
         my0=h#4189;                      { mult Hz by .512 * 2 }
         mr=sr1*my0(rnd);                 { i.e. mult by 1.024 }
         sr=ashift mr1 by 1 (hi);
         ar=sr1+ay0;
         dm(sum1)=ar;
         ax0=ar;
         call boot_sin;
         sr=ashift ar (hi);               { scale value in SE }
         dm(sin1)=sr1;

tone2:   ay0=dm(sum2);
         si=dm(hertz2);                   { freq stored as Hz in DM }
         sr=ashift si by 3 (hi);
         my0=h#4189;                      { mult Hz by .512 * 2 }
         mr=sr1*my0(rnd);                 { i.e. mult by 1.024 }
         sr=ashift mr1 by 1 (hi);
         ar=sr1+ay0;
         dm(sum2)=ar;
         ax0=ar;
         call boot_sin;
         sr=ashift ar (hi);               { scale value in SE }
         dm(sin2)=sr1;

add_em:  ax0=dm(sin1);
         ay0=dm(sin2);
         ar=ax0+ay0;

sound:   sr=ashift ar by -2 (hi);         { compand 14 LSBs only! }
         tx0=sr1;                         { send “signal” sample to SPORT }
            DIS SEC_REG;
         rti;

quiet:   ar=0;
         tx0=ar;                          { send “silence” sample to SPORT }
            DIS SEC_REG;
         rti;
```

```
{————————————————————————————————————————————————————————————————————————}
{————— S U B R O U T I N E S ————————————————————————————————————————————}
{————————————————————————————————————————————————————————————————————————}

{————————————————————————————————————————————————————————————————————————}
{                 Change the Digit                                       }
{ AX0 = digit                                                            }

new_digit:
         ay0=h#000F;
         ar=ax0 and ay0;
         sr=lshift ar by 1 (hi);
         ay0=^hz_list;
         ar=sr1+ay0;
         i1=ar;
         ax0=dm(i1,m1);              { look up row freq for digit }
         dm(hertz1)=ax0;
         ax0=dm(i1,m1);              { look up col freq for digit }
         dm(hertz2)=ax0;
            SI=0;
            DM(sum1)=SI;
            DM(sum2)=SI;
         rts;

{————————————————————————————————————————————————————————————————————————}
{                 Maketones or Makesilence                               }
make_tones:
         AR=1;
         DM(maketones_or_silence)=AR;
         RTS;

make_silence:
         AR=0;
         DM(maketones_or_silence)=AR;
         RTS;

{————————————————————————————————————————————————————————————————————————}
{                 Initialize PM                                          }

dm_inits:
         i0=^scale;
         m1=1;
         l0=0;
         i4=^PM_scale;
         m5=1;
         l4=0;
         cntr=PM_copy_length;
         do boot_copy until ce;
            si=pm(i4,m5);
            sr=lshift si by 8 (hi);
            si=px;
            sr=sr or lshift si by 0 (hi);
boot_copy:  dm(i0,m1)=sr1;
         rts;
```

(listing continues on next page)

```
{_______________________________________________________________}
{
   Sine Approximation
         Y = boot_sin(x)

   Calling Parameters
         AX0 = x in scaled 1.15 format
         M7 = 1
         L7 = 0

   Return Values
         AR = y in 1.15 format

   Altered Registers
         AY0,AF,AR,MY1,MX1,MF,MR,SR,I7

   Computation Time
         25 cycles
}

boot_sin:
         M5=1;
         L7=0;
         I7=^sin_coeff;                    {Pointer to coeff. buffer}
         AY0=H#4000;
         AR=AX0, AF=AX0 AND AY0;           {Check 2nd or 4th quad.}
         IF NE AR=-AX0;                    {If yes, negate input}
         AY0=H#7FFF;
         AR=AR AND AY0;                    {Remove sign bit}
         MY1=AR;
         MF=AR*MY1 (RND), MX1=PM(I7,M5);  {MF = x2}
         MR=MX1*MY1 (SS), MX1=PM(I7,M5);  {MR = C1x}
         CNTR=3;
         DO approx UNTIL CE;
            MR=MR+MX1*MF (SS);
approx:     MF=AR*MF (RND), MX1=PM(I7,M5);
         MR=MR+MX1*MF (SS);
         SR=ASHIFT MR1 BY 3 (HI);
         SR=SR OR LSHIFT MR0 BY 3 (LO);   {Convert to 1.15 format}
         AR=PASS SR1;
         IF LT AR=PASS AY0;                {Saturate if needed}
         AF=PASS AX0;
         IF LT AR=-AR;                     {Negate output if needed}
         RTS;
.ENDMOD;
```

Listing 6.16 DTMF Signal Generator Routine (DTMF.DSP)

```
{________________________________________________________________________}
{            Dial Phone Number and Display                               }
{________________________________________________________________________}

.MODULE/RAM/BOOT=1/ABS=0      dial_n_display;

.PORT        led_and_bank;
.PORT        display_base;
.EXTERNAL    eight_khz, new_digit, dm_inits, make_tones, make_silence;
.EXTERNAL    digit_count, phone_number, long_distance_flag;
.INCLUDE     <vocab.h>;

reset_vec:   CALL dm_inits;
             CALL init_control_regs;
             JUMP start;
             NOP;
irq2:        RTI;NOP;NOP;NOP;
s0_tx:       RTI;NOP;NOP;NOP;
s0_rx:       JUMP eight_khz;NOP;NOP;NOP;
s1_tx_irq1: RTI;NOP;NOP;NOP;
s1_rx_irq0: RTI;NOP;NOP;NOP;
timer_exp:   JUMP timeout;NOP;NOP;NOP;

start:       L0=0;L1=0;L2=0;L3=0;
             L4=0;L5=0;L6=0;L7=0;
             M0=0;M1=1;M2=-1;M3=2;         {standard setup}
             M4=0;M5=1;M6=-1;M7=0;

             CALL make_silence;
             ICNTL=0x00111;
             IMASK=b#001001;               {enable timer & rx0 interrupt}

             AX0=0xA;                      {dial tone}
             CALL new_digit;
             CALL make_tones;
             CALL wait_one;
             CALL wait_half;
             CALL wait_quarter;
             CNTR=DM(digit_count);
             AR=0;
             DM(digit_count)=AR;
             I4=^phone_number;
```

(listing continues on next page)

```
each_tone:          AX0=DM(I4,M5);
                    CALL new_digit;
                    CALL make_tones;

                    AY0=DM(digit_count);    {display sucessive digits}
                    AR=AY0+1;
                    DM(digit_count)=AR;
                    CALL display_number;

                    CALL wait_sixteenth;
                    CALL make_silence;
                    CALL wait_eighth;
                    IF NOT CE JUMP each_tone;

         CALL wait_two;
         I4=^gsm_;
         CALL display_text;
         AR=0x029b;                         {boot page 2}
         DM(0x3FFF)=AR;

{____________________Now Go To Boot Page One____________________________}

{_______________________________________________________________________}
{                       Subroutines                                     }
{_______________________________________________________________________}

{_______________________________________________________________________}
{                   Wait using timer interrupt                          }
wait_four:          CALL wait_two;
wait_two:           CALL wait_one;
wait_one:           CALL wait_half;
wait_half:          CALL wait_quarter;
wait_quarter:       CALL wait_eighth;
wait_eighth:        CALL wait_sixteenth;
wait_sixteenth:     AY0=0xFF;
                    AY1=0x0BC4;
wait_timer:         DM(0x3FFB)=AY0;         {TSCALE}
                    DM(0x3FFC)=AY1;         {TCOUNT}
                    DM(0x3FFD)=AY1;         {TPERIOD}
                    AY0=0;
                    ENA TIMER;
wait_here:          AR=PASS AY0;
                    IF EQ JUMP wait_here;
                    DIS TIMER;
                    RTS;
```

```
{________________________________________________________________________}
{                 Timer Interrupt Handler                                }

timeout: AY0=0xFFFF;               {set the timer expired flag}
         RTI;

{________________________________________________________________________}
{                       Display_Text                                     }
{ I4 = ^ascii text buffer in PM                                          }
{ Format of text buffer:<# characters, ascii data;>                      }
{ alters: I4,L4,I2,L2,AR,AY0,AY1                                         }

display_text:
         CALL clear_display;
         L4=0;
         AY1=PM(I4,M5);            {get # characters}
         AR=16;
         AR=AR-AY1;                {center the word}
         SR=LSHIFT AR BY -1 (LO);  {SR0=(16-#characters)/2}
         CNTR=SR0;
         CALL display_spaces;      {display leading spaces}
         CNTR=AY1;                 {#characters to display}
char_loop:      AR=PM(I4,M5);      {get character}
                CALL disp_char;    {display one character}
                IF NOT CE JUMP char_loop;
         RTS;

{________________________________________________________________________}
{                 Display One Character                                  }
{ AR = ascii character                                                   }
{ I2 = display pointer, decremented by one                               }
{ alters: AY0,AR,I2                                                      }

disp_char:
         AY0=0x0080;
         AR=AR OR AY0;             {WR high}
         DM(I2,M0)=AR, AR=AR XOR AY0;
         DM(I2,M0)=AR;             {WR low}
         AR=AR OR AY0;
         NOP;
         DM(I2,M2)=AR;             {WR high}
         RTS;
```

(listing continues on next page)

```
{_______________________________________________________________________}
{           Clear the ASCII display with N spaces                       }
{ CNTR = number of spaces                                               }
{ I2 = returned with the current characters location                    }
{ alters: I2,L2,AR,AY0                                                  }

clear_display:
            CNTR=16;                {Entry to clear entire display}

display_spaces:
            I2=^display_base + 15;  {Entry to clear leading spaces}
            AR=CNTR;
            AR=PASS AR;
            IF EQ JUMP exit_clear;  {Return if no leading zeros}
            L2=0;
            AR=0x0020;              {space}
clear_loop:       CALL disp_char;
                  IF NOT CE JUMP clear_loop;
            RTS;
exit_clear: POP CNTR;
            RTS;

{_______________________________________________________________________}
{                         Display Number                                }
{ Displays digit_count characters from digit buffer in DM.              }
{ Format of text buffer:<# characters, ascii data;>                     }
{ alters: I4,L4,I2,L2,AR,AY0,AY1                                        }

display_number:
            CALL clear_display;
            I4=^phone_number;
            L4=0;
            AY1=DM(digit_count);    {get # digits}
            CNTR=3;
            CALL display_spaces;    {display leading spaces}
            CNTR=AY1;               {#characters to display}
            AF=PASS 0;              {counts digits}
            AY1=0x30;               {ascii 0 offset}
digd_loop:        AX0=3;
                  AR=AX0-AF;
                  IF EQ CALL display_dash;
                  AF=AF+1;
                  AR=DM(I4,M5);     {get digit}
                  AR=AR+AY1;        {offset for ascii}
                  CALL disp_char;   {display one character}
                  IF NOT CE JUMP digd_loop;
            RTS;

display_dash:
            AR=0x2d;                {ascii dash}
            CALL disp_char;
            RTS;
```

```
{________________________________________________________________________}
{                        Init Control Registers                          }
{ Set Up SPORTS and TIMER on EZ-LAB board after RESET                    }
{ used for ADSP-2101 EZ-LAB demonstrations                               }
{ Altered Registers: I0,M1,L0                                            }

init_control_regs:
            L0=0;
            M1=1;
            I0=h#3FEF;        {point to last DM-mapped control registers }

{ h#3FEF }  DM(I0,M1)=H#0000;       {SPORT1 AUTOBUFF DISABLED}
{ h#3FF0 }  DM(I0,M1)=H#0000;       {SPORT1 RFSDIV NOT USED}
{ h#3FF1 }  DM(I0,M1)=H#0000;       {SPORT1 SCLKDIV NOT USED}
{ h#3FF2 }  DM(I0,M1)=H#0000;       {SPORT1 CNTL DISABLED}
{ h#3FF3 }  DM(I0,M1)=H#0000;       {SPORT0 AUTOBUFF DISABLED}
{ h#3FF4 }  DM(I0,M1)=   255;       {RFSDIV for 8 kHz interrupt rate}
{ h#3FF5 }  DM(I0,M1)=     2;       {SCLKDIV=2 makes 2.048 MHz
                                     with 12.288 MHz xtal}
 { h#3FF6 } DM(I0,M1)=H#6927;       {Int SCLK,
                                     RFS req, TFS req,
                                     Int RFS, Int TFS,
                                     u_law, 8-bit PCM }
{ h#3FF7 }  DM(I0,M1)=H#0000;       {TRANSMIT MULTICHANNELS}
{ h#3FF8 }  DM(I0,M1)=H#0000;
{ h#3FF9 }  DM(I0,M1)=H#0000;       {RECEIVE MULTICHANNELS}
{ h#3FFA }  DM(I0,M1)=H#0000;
{ h#3FFB }  DM(I0,M1)=H#0000;       {TIMER NOT USED, CLEARED}
{ h#3FFC }  DM(I0,M1)=H#0000;
{ h#3FFD }  DM(I0,M1)=H#0000;
{ h#3FFE }  DM(I0,M1)=H#0000;       {DM WAIT STATES}
{ h#3FFF }  DM(I0,M1)=H#101B;       {SPORT0 ENABLED}
                                    {BOOT PAGE 0, 3 PM WAITS}
                                    {3 BOOT WAITS}
            rts;

{________________________________________________________________________}

.ENDMOD;
```

Listing 6.17 Automatic Dialing Routine (DTMFMAIN.DSP)

6.6 REFERENCES

Atal, B.S. June 1974. “Effectiveness of Linear Prediction Characteristics of the Speech Wave for Automatic Speaker Identification and Verification,” *Journal of the Acoustical Society of America*, vol. 55, No. 6, pp. 1304-1312.

Gray, A.H. and J. D. Markel. October 1976. “Distance Measures for Speech Processing,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-24, No. 5, pp. 380-391.

Gray, R.M., A. Buzo, A. H. Gray, and Y. Matsuyama. August 1980. “Distortion Measures for Speech Processing,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, No. 4, pp. 376-376.

Itakura, F. February 1975. “Minimum Prediction Residual Principle Applied to Speech Recognition,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-23, No. 1, pp. 67-72.

Juang, B.H., L. R. Rabiner, and J. G. Wilpon. July 1987. “On the Use of Bandpass Liftering in Speech Recognition,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, No. 7, pp. 947-954.

Makhoul, J. April 1975. “Linear Prediction: A Tutorial Review,” *Proceedings of the IEEE*, vol. 63, No. 4, pp. 561-580.

Mansour, D. and B. H. Juang. November 1989. “A Family of Distortion Measures Based Upon Projection Operation for Robust Speech Recognition,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, No. 11, pp. 1659-1671.

Nocerino, N., F. K. Soong, L. R. Rabiner, and D. H. Klatt. December 1985. “Comparative Study of Several Distortion Measures for Speech Recognition,” *Speech Communication*, vol. 4, pp. 317-331.

Paliwal, K.K. 1982. “On the Performance of the Quefrency-Weighted Cepstral Coefficients in Vowel Recognition,” *Speech Communication*, vol. 1, pp. 151-154.

Rabiner, L.R., S. E. Levinson, A. E. Rosenberg, and J. G. Wilpon. August 1979. “Speaker Independent Recognition of Isolated Words Using Clustering Techniques,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-27, No. 4, pp. 336-349.

Rabiner. L.R. and M. R. Sambur. February 1975. "An Algorithm for Determining the Endpoints of Isolated Utterances," *The Bell System Technical Journal*, vol. 54, No. 2, pp. 297-315.

Rabiner, L.R. and R. W. Schafer. 1978. *Digital Processing of Speech Signals*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Rabiner, L.R. and J. G. Wilpon. 1987. "Some Performance Benchmarks for Isolated Word Speech Recognition Systems," *Computer Speech and Language*, vol. 2, pp. 343-357.

Schroeder, M.R. April 1981. "Direct (Nonrecursive) Relations Between Cepstrum and Predictor Coefficients," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-29, No. 2, pp. 297-301.

Tohkura, Y. April 1986. "A Weighted Cepstral Distance Measure for Speech Recognition," *Proceedings of ICASSP 1986*, pp. 761-764.