A Multiline Computer Voice Response System Utilizing ADPCM Coded Speech

LEWIS H. ROSENTHAL, MEMBER, IEEE, LAWRENCE R. RABINER, MEMBER, IEEE, RONALD W. SCHAFER, MEMBER, IEEE, P. CUMMISKEY, MEMBER, IEEE, AND JAMES L. FLANAGAN, FELLOW, IEEE

Abstract—**In this paper we discuss the issues involved in implementing an automatic computer voice response system which is capable of serving up to ten independent output channels in real time. The system has been implemented on a Data General NOVA-800 minicomputer. Individual isolated words and phrases are coded at a rate of 24 000 bits/s using a hardware adaptive, differential pulse-code modulation (ADPCM) coder, and stored on a fixed-head disk as a random access vocabulary. By exploiting the features of ADPCM coding, it is possible to create and edit automatically a vocabulary for the system from an analog tape recording of the spoken entries, with minimal operator intervention. To provide ten simultaneous output lines of speech which are independent of each other required the use of an efficient scheduling algorithm. Such an algorithm was provided by the computer manufacturer in their real-time multitasking system which was part of their Fortran software. Thus almost all the programming required to implement this real-time system was in Fortran, thereby providing flexibility and ease in making changes in the system. Initial applications of the voice response system are in computer aided voice wiring, automatic directory assistance, and experiments on speaker verification, but the system is sufficiently modular to adapt readily to other applications.**

INTRODUCTION

ONE of the current goals of speech research is to simplify the problem of communication between man and machine by making it possible to use the human voice as a means of interaction. Toward this end, work has been carried out both in the area of automatic speech recognition [1] and in the area of automatic voice response [2], [3]. At the present time, speech recognition systems are still too primitive to be used in a general purpose man–machine interface. However, automatic voice response systems are now quite feasible and have numerous applications. A few examples should serve to illustrate the wide range of capabilities that such a voice response system might have.

One fairly straightforward application of a voice response system is computer-aided voice wiring [4], [5]. Conventionally, a wireman works from a printed list which contains the information for each wiring operation. However, in many wiring situations it is awkward for the wireman to divert his eyes from his work in order to consult such a list. In these cases, it becomes convenient to record the wirelist in spoken form on a cassette tape and allow the wireman to work from a spoken list. Typically, a foot switch is used to start the playback unit, and recorded tones on the tape automatically stop the unit after each wiring instruction. A major problem with this technique is that it requires a considerable amount of human effort in order to generate a spoken wirelist. One person must read the list (which may be several hours long), and another must monitor the audio output. Any errors detected must then be edited and corrected. Even after a wirelist has been recorded successfully, any future updating of the list requires that this entire process be repeated. In some cases, this could occur several times in the course of a few days or weeks. The tedium factor here is extremely high, which tends to increase human errors and make this entire procedure impractical. However, this problem could be completely eliminated by generating the spoken lists automatically with a voice response system. If such a system operated faster than real time, it would have multichannel capabilities. A number of independent lists could then be generated simultaneously, or a long single list could be generated in pieces at a faster rate. This technique for the generation of spoken wirelists is currently of interest to the Bell System but undoubtedly would find application elsewhere as well.

The availability of a Touch-Tone® telephone would make possible a large number of additional applications for automatic voice response. In these applications, the telephone could be used as a remote computer terminal, providing keyboard input and voice output. Such a system would make computer data bases accessible to anyone with a telephone. Applications include automatic telephone directory assistance, automatic verification of credit card numbers, automatic confirmation of reservations, and many other applications requiring interactive communication with a computer. Finally, such a system could be combined with a speaker verification system to provide on-line voice verification for applications such as authentication of credit card users or banking by telephone. These are only a few examples of applications for automatic voice response, but they serve to give some idea of the motivation behind the development of an automatic voice response system.

Manuscript received February 15, 1974.
The authors are with Bell Laboratories, Murray Hill, N. J. 07974.

® Registered service mark of the American Telephone and Telegraph Company.

DESIGN CONSIDERATIONS FOR A VOICE RESPONSE SYSTEM

The general form of a voice response system is illustrated in the block diagram shown in Fig. 1. The vocabulary for such a system may consist of individual words, phrases, and even entire sentences. Before voice output is possible, a vocabulary for the system must be prepared and stored. Voice response is then obtained by accessing the required vocabulary entries in sequence to form the desired message. A multiline voice response system processes several message requests in parallel, directing each speech entry accessed to the appropriate output channel.

The actual design of a voice response system depends on both the unit of vocabulary storage (e.g., words, phrases, etc.) and the method of storing these units. Efficient vocabulary storage generally requires that the vocabulary entries be individual words, since many different messages can be composed from combinations of the same words. On the other hand, highly contextual messages composed from individual words can sound very unnatural, unless a fairly sophisticated algorithm is used to interpolate pitch, amplitude and formant frequencies across word boundaries. Thus, a tradeoff exists between efficiency of vocabulary storage and the naturalness of the output speech. Nevertheless, for many applications of voice response it is adequate to generate speech by a simple concatenation or abutting of individual words and phrases. These applications generally require noncontextual output speech, the elements of which can be stored as individual words, and a small number of contextual messages, which can be stored as entire phrases or sentences. This restriction on the output speech greatly reduces the complexity of the voice response system, but it still allows for all of the applications previously discussed. In addition, it makes possible the design of a totally digital multiline voice response system, based on a medium-sized vocabulary (under 200 words or phrases) which can be edited with a minimum of effort. The design of such a voice response system, with the capability of servicing up to ten simultaneous output lines, is the subject of this paper.

Fig. 2 shows an overall block diagram of the digital voice response system which will be described here. The main feature which distinguishes this system from the general system of Fig. 1 is that all speech processing is done digitally. Input speech is converted to digital form before being edited, cataloged, and stored, and the speech remains in digital form until a desired output message is formed.

The voice response system was implemented on a Data General NOVA-800 minicomputer. The computer is a 16-bit word length machine with 32 768 words of 800 ns core memory, hardware integer multiply/divide, hardware floating point unit, and 786 432 words of fixed head disk. The computer facility also included a real-time clock, 32 high speed direct memory access channels, Tektronix 4010 terminal, two moving head disk packs, paper tape

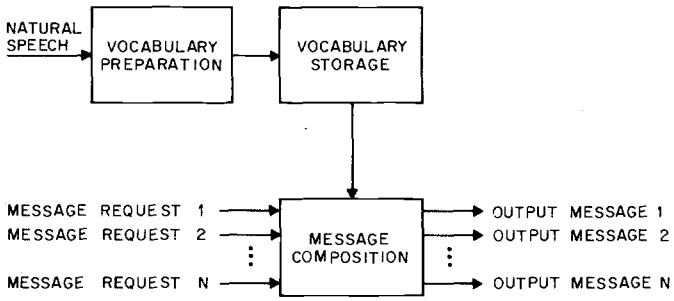

Fig. 1. Block diagram of a voice response system.

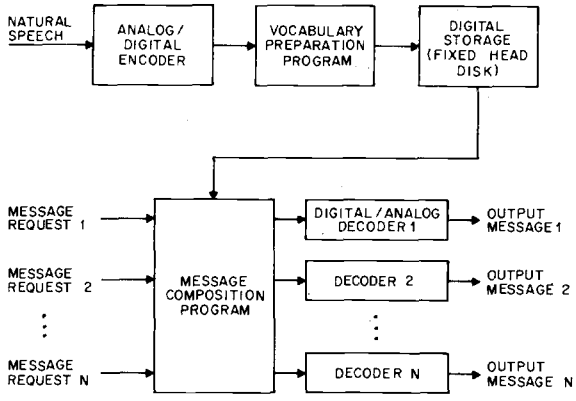

Fig. 2. Block diagram of an all-digital voice response system.

reader, line printer, and a card reader. Extensive software was also supplied with the computer including a real-time disk operating system (RDOS), and a Fortran V compiler, along with other standard programs such as an editor, an assembler, and a library file editor.

The design of the voice response system can be logically divided into two parts, the first part dealing with the creation and editing of a vocabulary and the second part dealing with the characteristics of the output, or message composition, system. These two parts will be discussed in separate sections of this paper, since their design considerations are totally different. Throughout the discussion, any factors which tend to limit the maximum capabilities of the system will be pointed out and explained.

CREATION OF A VOCABULARY

An important requirement of a voice response system is a convenient means of storing, editing and updating the vocabulary for the system. Vocabulary entries must be stored in such a way that they can be quickly accessed whenever needed, and the resulting message formed by a concatenation of several entries should sound as natural as possible. In addition, it is desirable to have a means of easily changing vocabulary entries whenever necessary. Currently existing voice response systems employ a variety of both analog and digital recording techniques, in an attempt to meet these requirements. These possibilities will now be discussed in detail, with an emphasis on the degree to which each method meets the requirements just outlined.

One means of vocabulary storage is to record speech entries in analog form on a tape, film, or magnetic drum.

This technique is used mainly in systems with relatively small vocabularies, since the entries are not stored very economically. The storage medium itself is usually divided into tracks, each of fixed duration, and an individual word or phrase is recorded on each track. This recording technique is probably the simplest of those used, but it is also the least flexible. For example, it is often necessary to compress a long word so that it can fit on a single track of the storage medium. This, plus the fact that pauses between words must also be of fixed duration, tends to add an unnatural quality to the output speech. In addition, retrieval time for a given word is usually fairly long, because of the time required to position a reading device at the beginning of the correct track. Finally, the ease in changing vocabulary entries depends on the particular storage device used. If the medium is photographic film, generation of a new vocabulary requires a new storage medium as well. It appears, therefore, that analog storage leaves much to be desired in meeting the requirements for a flexible voice response system. Nevertheless, such systems are in current use and are available commercially.

The alternative to analog storage is, of course, some form of digital storage of speech. Digital storage devices for automatic voice response systems include magnetic disks, drums, and solid state memories. These devices all provide faster access times than analog storage media and allow the possibility of variable length vocabulary entries. Solid state memories provide the fastest access times, but large ones are still economically impractical. The best compromise between access time and storage capacity is provided by a fixed head, or head-per-track, disk. This device has tracks similar to those of an analog drum, but it rotates at a much higher speed (3600 rpm is typical), thus providing rapid access to any section of a track. In addition, since the disk is a read/write memory, changing vocabulary entries is never a problem.

Another consideration when employing digital storage is the selection of a method for digitizing speech. Recent efforts in speech research have provided several possible alternatives for this choice [6]. The most direct method for digitizing speech is to use linear pulse-code modulation (PCM), which simply quantizes a band-limited analog signal both in time and in amplitude. This method does not require any special processing of the speech signal, but it does require bit rates on the order of 60 000 bits/s (6 kHz sampling rate $\times$ 10 bits/sample) to produce good quality speech. Lower bit rates can be obtained by reducing the sampling rate and/or the number of bits per sample, but only at the expense of speech quality. Thus, this technique is unacceptable for voice response systems which require either a medium-sized vocabulary or the capability to service a reasonable number of output lines simultaneously. An alternative method for analyzing and synthesizing speech is to use formant coding procedures [4]. The bit rate required for storage of the formant parameters is on the order of 1000 bits/s but considerable computation time is required both to get the formants for a given word and to synthesize a word from its formants, unless special purpose hardware is available. Thus, this technique is convenient only in applications where vocabulary size is the most important factor. Between these two extremes of bit rate are various methods for coding speech, including log PCM, differential PCM (DPCM) [7], adaptive delta modulation (ADM) [8], and adaptive DPCM (ADPCM) [9], [10]. The bit rates for these coding techniques range from 50 000 bits/s down to approximately 16 000 bits/s, depending on the sampling rate and the number of bits used to represent each sample. The speech quality at these bit rates is not as good as that of high quality linear PCM, but for many applications of automatic voice response high quality speech is not required. Instead, the goal of these techniques is to optimize speech quality for a given bit rate. These coding schemes all perform some processing of the original speech signal, but this processing can generally be done using fairly simple and inexpensive digital hardware. Thus, for applications requiring both medium-sized vocabularies and flexibility in changing vocabulary entries, one of these coding techniques provides a good compromise between bit rate and speech quality. Based on a number of considerations, ADPCM coding was the method selected.

There were several reasons for selecting ADPCM coding for the voice response system. First, it provides good quality (20 dB S/N) [9] speech at a bit rate of 24 000 bits/s (6 kHz sampling rate $\times$ 4 bits per sample). ADPCM-coded speech at this bit rate was found to be perceptually comparable in quality to PCM at a bit rate of 42 000 bits/s [9], [10][1] (7 log bits $\times$ 6000 samples/s). (At a bit rate of 24 000 bits/s ADPCM also has about a 6 dB signal-to-noise ratio advantage over adaptive delta modulation [6].) Another factor in the selection of ADPCM coding was that the entire encoding/decoding process could be performed by fairly inexpensive hardware in real time. Thus, no central processing unit (CPU) time would be needed for processing of the speech data. Finally, it has been found that an automatic editing algorithm to isolate the endpoints of an utterance can be devised for ADPCM-coded speech. This algorithm will be described later in this paper.

Before describing the principles of ADPCM coding a brief review of linear PCM will be given in order to provide a basis for comparison with ADPCM. Fig. 3 shows a block diagram for a linear PCM coder and decoder. Input to the coder is the sampled speech waveform $x(n)$, which is discrete in time but not in amplitude. Digital output is obtained by quantizing $x(n)$ in amplitude with a fixed, uniform quantizer to produce $\hat{x}(n)$ and then encoding each quantized sample to produce a binary representation. In the decoder, the quantized signal $\hat{x}(n)$ is reconstructed from the bit stream and then low-pass filtered to produce

[1] The objective signal-to-noise ratio improvement of ADPCM over log PCM is only about 8–10 dB. The subjective improvement is on the order of 18 dB or 3 bits.

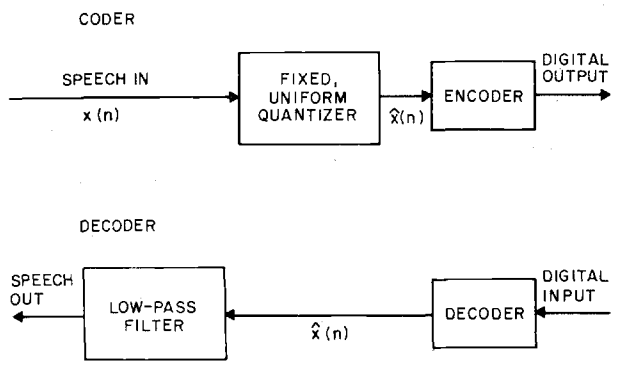

Fig. 3. PCM coder/decoder block diagram.

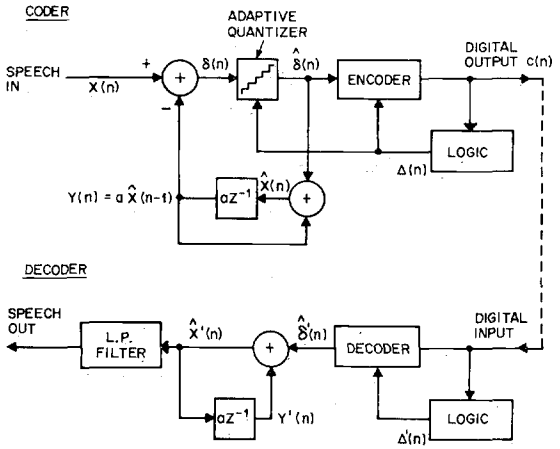

Fig. 4. ADPCM coder/decoder block diagram.

the analog speech output. Thus, it is seen that the entire encoding/decoding process for linear PCM is fairly straightforward.

The major problem with linear PCM coding is that it is a very inflexible coding technique. Once the sampling rate and the number of bits per sample have been specified, the bit rate (sampling rate × number of bits per sample) and speech quality are completely determined. If the sampling rate is reduced further, intelligibility is sacrificed. Alternatively, if fewer bits are used to represent each sample, more noise will be present in the coded signal. The latter problem is due to the limitations of a fixed, uniform quantizer. This quantizer must have both a step size small enough to provide good resolution at low signal levels and a large number of quantization levels in order to cover the wide dynamic range of an input signal. For a speech signal, which covers a very wide dynamic range, these requirements necessitate a high bit rate for linear PCM-coded speech.

In order to overcome the limitations of linear PCM coding, two separate approaches have been taken with ADPCM. First, instead of quantizing the actual speech waveform, a difference signal derived from the waveform is quantized instead. The motivation here is that there is a high degree of correlation between successive speech samples. Therefore, a difference signal will be smaller than the original signal and can be more accurately represented using a small number of quantizer levels. Second, in order to compensate for a wide range of input levels, the quantizer step size is made to adapt instantaneously to the level of the signal at its input, thus providing more resolution when the signal level is low and less resolution when the signal level is high.

Fig. 4 shows a block diagram of an ADPCM coder and decoder. Without the adaptive quantizer, the system is a conventional DPCM coder. The difference signal $\delta(n)$ at the top of Fig. 4 is obtained by subtracting from each sample of the input speech signal an estimate of that sample $y(n)$. The estimate is obtained by taking the sum of the previous estimate $y(n-1)$ and the previous quantized difference $\hat{\delta}(n-1)$, scaled by the constant a (which is close to 1.0). By using this technique for deriving the difference signal, it can be shown that quantization errors do not accumulate with successive differences [7].

In fact, for each sample of the input signal $x(n)$, the quantity $x(n) - \hat{x}(n)$, which is the difference between the actual speech sample and the sample which will be reconstructed by the decoder, is equal to $\delta(n) - \hat{\delta}(n)$, the quantization error for the current difference. It has been shown that the use of differential coding in the DPCM coder yields a 4–6 dB advantage in S/N over PCM coding for speech [9], [10].

Further improvement in the DPCM coder is obtained by using an adaptive quantizer. The input–output relation for the adaptive quantizer is given in Fig. 5. Such a quantizer modifies its step size $\Delta(n)$ on the basis of the most recent quantizer output. For the current implementation, the adaptation rule is that for each input to the quantizer $\delta(n)$, the quantizer step size $\Delta(n)$ is obtained from the formula

$$\Delta(n) = M\Delta(n-1), \tag{1}$$

where M, the step-size multiplier, is a function of the previous quantizer input $\delta(n-1)$. Meaningful adaptation requires that the step size be decreased upon detection of quantizer underload (i.e., when the quantizer input is small) and increased upon detection of quantizer overload (i.e., when the quantizer input is large). The particular values used for M in the current implementation were selected by performing computer simulations of the ADPCM coder and are given in Fig. 5. In the hardware implementation of the coder, the quantizer has a finite library of 21 possible step sizes with a range of 100 to 1. Thus, the quantizer step size $\Delta(n)$ obeys the relation in (1) only for a certain range of values, beyond which it saturates either at a maximum or minimum step size. The use of an adaptive quantizer has been shown to give an additional 4 dB advantage in S/N over PCM [9], [10]. It should be noted that only the 4-bit quantizer output level must be transmitted since the step-size adaptation algorithm in the decoder is identical to the one in the coder, and therefore the step size is completely determined from the sequence of quantizer output levels.

The remainder of the ADPCM coder/decoder of Fig. 4 is straightforward. The quantized difference signal $\hat{\delta}(n)$

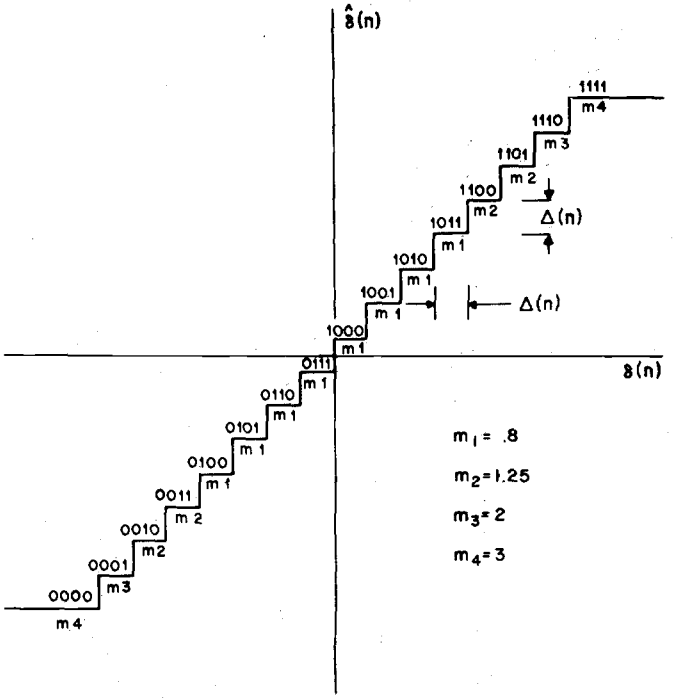

Fig. 5. Quantizer characteristic for ADPCM coder.

is encoded to produce a digital output in the same manner as with linear PCM, except that the encoder is controlled by the step-size adaptation logic. Thus, the resulting digital output $c(n)$ for each input sample is a 4-bit number which reflects the quantizer output level as well as the change in level of the input speech. In the decoder, the same adaptation logic as was employed in the coder is used, so that in the absence of transmission errors $\hat{\delta}'(n)$, the decoded quantizer output, is identical to $\hat{\delta}(n)$. A differential feedback loop identical to that used in the coder reconstructs the signal $\hat{x}'(n)$, and this signal is then low-pass filtered to produce the analog output.

For the voice response system which was implemented, 1 ADPCM coder/decoder and 9 decoders were constructed. Fig. 6 is a photograph of the actual hardware for one of these coders. Each coder was separately interfaced to the computer. The coder interface packs four 4-bit coded speech samples to each 16-bit computer word. Each decoder interface, in turn, unpacks the coded samples and clocks them to the decoder. By performing this packing/unpacking process in the interface instead of in the computer, the computation time saved can be utilized more efficiently to control the multiple, simultaneous output lines of speech.

AUTOMATIC EDITING OF ADPCM CODED SPEECH

After a file of speech has been created using the ADPCM coder and stored on a fixed head disk, it is necessary to isolate the individual words and phrases in the file, so that the speech can be stored efficiently in the word catalog without intervening periods of silence between entries. Conventionally, this editing is done manually by a combination of listening to the speech and studying

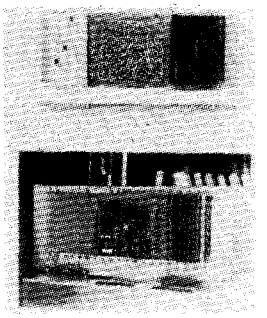

Fig. 6. Hardware for ADPCM coder/decoder.

a visual display of the speech waveform. However, this process is both time consuming and subject to inaccuracies due to human error. Furthermore, this method does not yield repeatable results. This problem is especially noticeable when an unvoiced segment of speech appears at the beginning or end of a word. As a result, manual editing usually results in the *shortening* of an utterance, both at the beginning and at the end. When such chopped words are concatenated to form a message, the effects become quite discernible and are, in fact, distracting.

ADPCM coding, however, provides an alternative to manual editing. Because of the step-size adaptation used in ADPCM coded speech, an ADPCM coder effectively has an automatic gain control (AGC). This causes the coded ADPCM samples to exhibit high energy[2] during both voiced and unvoiced speech, but not during low level background silence, which causes the ADPCM quantizer to saturate at its minimum step size. Thus a very sharp threshold between speech and no speech is obtained due to the effective AGC of the ADPCM coder. This sharp threshold is used as a means of distinguishing between silence and unvoiced segments. So long as the energy of the silence portion lies below the threshold value, then low value code word energies result. By contrast, low energy unvoiced segments that exceed the threshold will produce high value code word energies. Thus, a very important condition is that the original recordings be performed in a low noise environment. This is no real restriction for voice response applications as the vocabulary is generally recorded in a soundproof room. Using this fact, an algorithm was developed to determine automatically the beginning and end of an utterance.

Fig. 7 illustrates a block diagram of the editing procedure. For each ADPCM coded speech sample $c(n)$,[3] an energy calculation is made using the formula

$$E(n) = \sum_{i=n-50}^{n+50} [c(i) - 7.5]^2 \qquad (2)$$

where the constant 7.5 represents the dc average of the code words (see Fig. 5). Thus, the energy is computed over a 101 point window centered around the current

[2] This code word energy is not directly related to the energy of the original speech signal.

[3] $c(n)$ is the level equivalent of the 4-bit encoded quantizer output level; thus $c(n)$ is an integer in the range $0 \leq c(n) \leq 15$.

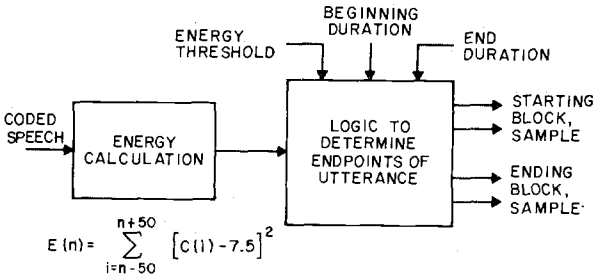

Fig. 7. Block diagram of endpoint algorithm.

speech sample, corresponding to approximately 16 ms of speech. The energy values obtained are then compared sample by sample with an energy threshold, which is set midway between the measured energy of background silence and the average energy of speech. When the code word energy exceeds this threshold for 50 ms or 300 consecutive samples, the point at which the energy first exceeded the threshold is recorded as the beginning of an utterance. The algorithm then begins at that point and continues to make the energy comparison with the threshold. When the code word energy falls below the threshold for 160 ms or 1000 consecutive samples, the point at which the energy first passed below the threshold is recorded as the end of the utterance. The 160 ms criterion ensures that the algorithm will not mistake a stop consonant within a word or phrase for the end of the utterance. A few examples will now illustrate the effectiveness of this algorithm.

Fig. 8 shows a code word plot for the beginning of the word "oh," a word which is all voiced speech. Each line of the plot corresponds to about 40 ms or 256 samples of speech. An offset of 7.5 has been subtracted from each sample in agreement with the energy calculation of (2). The vertical line in the plot marks the point at which the endpoint algorithm detected the beginning of the utterance. Because of the energy window, the algorithm was about 4 ms early in its identification of a starting point, but this error is small enough to be ignored. The energy waveform on which the identification was based is given in Fig. 9. Finally, Figs. 10 and 11 show, respectively, the decoded speech waveform and its energy for the same section of speech. Since the word "oh" is entirely voiced, it is not too difficult in this case to locate the start of speech from an observation of the decoded waveform. However, the transition from silence to speech is not nearly as abrupt as it is in the plot of the coded samples.

The next set of examples gives further evidence of the value of this technique for automatic editing of speech. Fig. 12 illustrates the coded samples for the beginning of the word "three," which begins with the unvoiced fricative "th." In this case, the transition from silence to speech is not as abrupt as it was in the previous example, even for the code words. Nevertheless, it is clear that there is significant code word activity in the vicinity of the point denoted by the endpoint algorithm. Fig. 13, which displays the energy of the code words for this segment of speech, confirms this observation. Note, however, the contrast

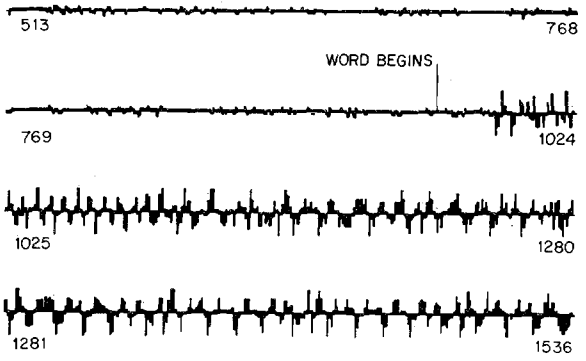

Fig. 8. Code word plot for beginning of utterance "oh".

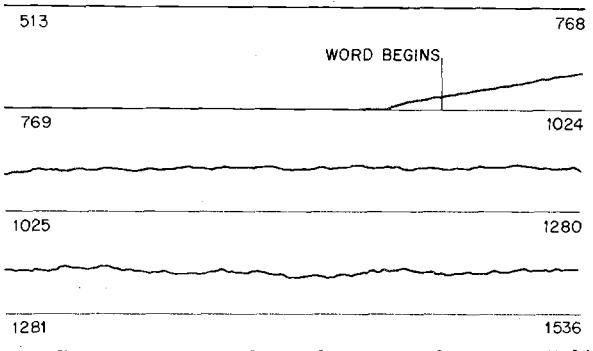

Fig. 9. Code word energy plot for beginning of utterance "oh".

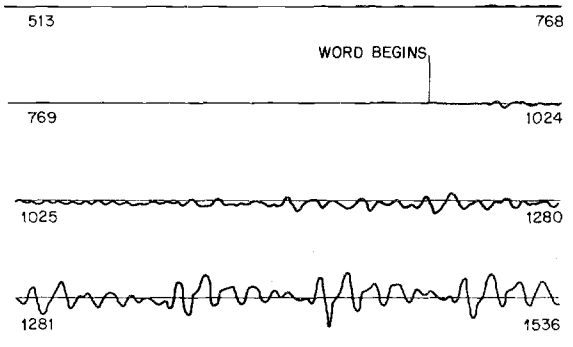

Fig. 10. Speech plot for beginning of utterance "oh".

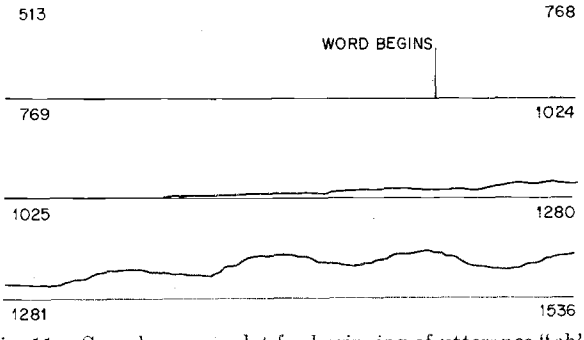

Fig. 11. Speech energy plot for beginning of utterance "oh".

between these plots and Figs. 14 and 15, which display the decoded speech waveform and its energy for the same section of speech. A trained observer may be able to detect the proper starting point of the utterance from these plots by increasing the gain of the unvoiced section, but any attempt to do this automatically would generally

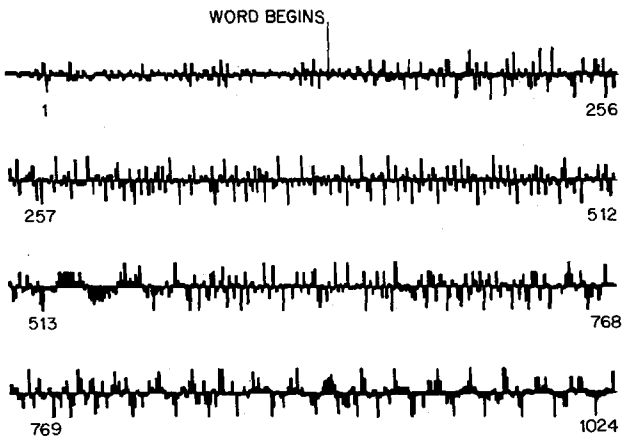

Fig. 12. Code word plot for beginning of utterance "three".

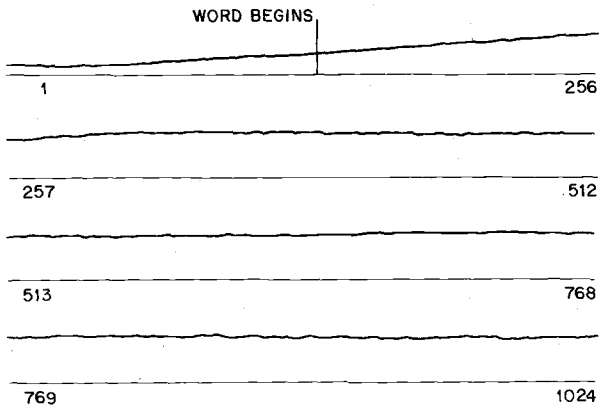

Fig. 13. Code word energy plot for beginning of utterance "three".

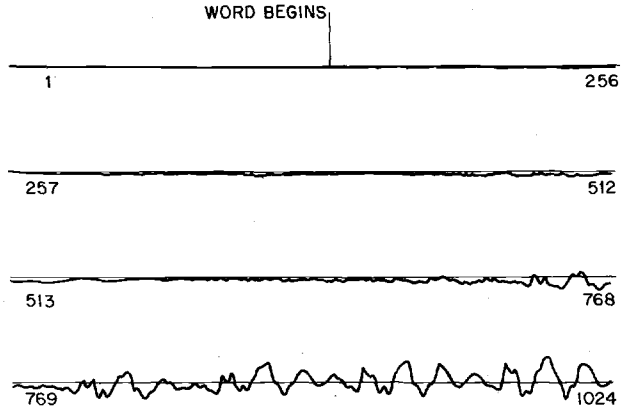

Fig. 14. Speech plot for beginning of utterance "three".

result in spurious identifications, since the energy of unvoiced speech is so low.

Similar results are obtained when this algorithm is used to detect the end of an utterance. The only requirement here is that there be a pause of at least 160 ms between spoken utterances, so that a stop consonant can be clearly distinguished from the end of a word.

The endpoint algorithm just discussed was tested on 50 typical entries for a voice response system vocabulary with essentially no errors. Auditory and visual verification indicated no evidence of shortening of any of the words. Two other measures of the coded speech signal,

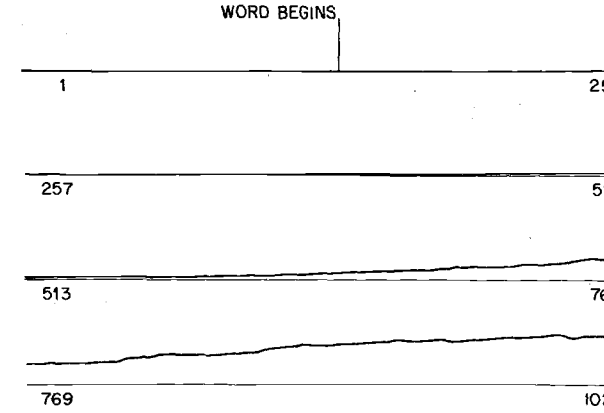

Fig. 15. Speech energy plot for beginning of utterance "three".

the energy in the difference signal and the energy in the quantizer step-size, were also studied as possible signals for use with the automatic editing algorithm. However, the results based on the coded samples themselves were far more accurate than those obtained from any of these other signals.

VOCABULARY CREATION

The first step in creating a vocabulary for the voice response system is to store coded speech on the disk. Fig. 16 shows a block diagram of how input speech is coded and stored. Coded 4-bit speech samples are packed 4 samples to a computer word by the coder interface and transferred directly to core buffers via the direct memory access (DMA) data channel. A recording program handles the disk transfers and takes care of all necessary bookkeeping functions.

Once speech is stored on the disk, a cataloging program is required to add, delete and rename entries in the vocabulary. To keep track of the entries in the speech file, a directory must be maintained. Fig. 17 shows a typical directory of a 30 entry vocabulary. The column labeled ENTRY gives the alphanumeric (ASCII) name associated with the speech entry. For example, the ASCII name PLT is associated with the word plate. The next three columns in Fig. 17 give information as to the starting disk block (NBKST) for the entry in the speech file, the total number of 256 word blocks (NBLK) occupied by that entry, and the number of samples (NSAMP) actually occupied in the last disk block. Recall that the automatic endpoint algorithm described earlier can isolate the word to within one sample and thus words need not occupy an integral number of disk blocks.

Fig. 18 shows a flow chart for the cataloging program. Basically, this program must perform bookkeeping on the speech directory, and on the speech file for the tasks of adding, deleting and renaming entries of the vocabulary. For adding entries this program makes use of the automatic endpoint algorithm previously described.

As can be seen in Fig. 17, the average storage requirement for an isolated word entry is 4 disk blocks or 1024 words. Thus a typical vocabulary of 100 words would require approximately 100 000 words of disk storage. This

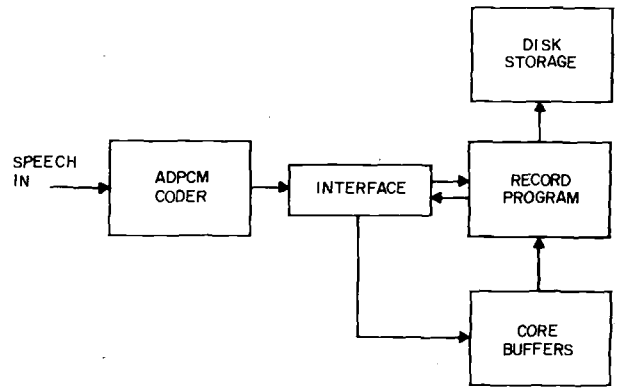

Fig. 16. Block diagram of ADPCM recording program.

	ENTRY	NBKST	NBLK	NSAMP
1	1	59	3	158
2	10	93	3	231
3	11	96	4	178
4	12	100	4	212
5	13	104	4	237
6	14	108	4	153
7	2	62	3	126
8	3	65	4	104
9	4	69	4	63
10	5	73	4	123
11	6	77	5	51
12	7	82	4	175
13	8	86	3	91
14	9	89	4	85
15	BAY	36	3	122
16	CRK	56	3	201
17	FLE	43	3	250
18	FRM	22	4	42
19	GRD	16	3	144
20	HOR	52	4	214
21	LNK	4	3	157
22	LVL	19	3	121
23	MIS	30	6	137
24	MON	46	3	243
25	NET	7	4	175
26	PLT	49	3	151
27	SWT	11	5	2
28	TRK	0	4	3
29	UVS	26	4	204
30	VER	39	4	1

NEXT FREE BLOCK IS BLOCK 112

Fig. 17. Typical speech directory for 30 entries.

amount of storage is readily available in most commercial fixed-head disk systems. The fixed-head disk used in our implementation had 786 432 words of storage. Thus, on the order of 800 words could be stored on such a disk.

THE OUTPUT SIDE OF THE VOICE RESPONSE SYSTEM

The retrieval, or message composition, section of the voice response system has a well defined function. It must access the vocabulary entries necessary to form the desired output message and output them with minimal delay and no unwanted pauses. In the case of a multiline voice response system, the retrieval section must supply each output channel with its desired message, again with minimal delay and no unwanted pauses. In order to provide at least a minimum of context in the output speech, it is also necessary to have control over the amount of silence between spoken entries. Finally, the system should be as modular as possible, so that it can be used

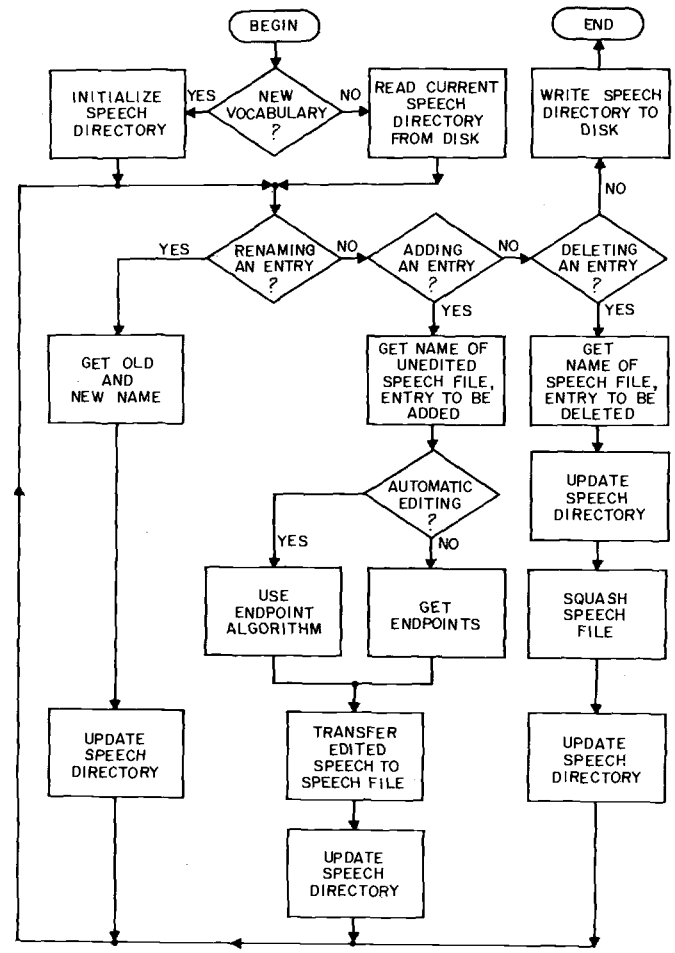

Fig. 18. Block diagram of word cataloging program.

in a variety of applications. It is believed that the output system to be described here meets all of the above requirements and the design principles discussed here can be applied in the design of a variety of sophisticated voice response system.

TIMING CONSIDERATIONS

The maximum number of output lines that a voice response system can support depends on many timing considerations. However, a first estimate can be obtained by considering the hardware constraints imposed by the disk and coders. Each track of the disk used in this implementation contains 8 sectors of 256 words each. However, because of hardware constraints, consecutive sectors cannot be written or read on the same revolution. Thus, the maximum rate of data transfer from the disk is 4×256 or 1024 words per revolution. The disk revolves at 60 revolutions/s; thus it can transfer data at the rate of approximately 60 000 words/s. The coder, as stated previously, operates at the rate of 24 000 bits, or 1500 words/s. Therefore, the ratio of disk data rate to coder data rate sets a theoretical maximum of 40 output lines for such a voice response system.

In practice, however, the above estimate is not really valid, since it assumes perfect disk read scheduling and

ignores the possibility of a conflict occurring when too many coders request data from the same sector of the disk. The size of speech buffers in core also limits the maximum number of output lines. If double buffering for each channel is assumed, so that one buffer can be filled while the other is being emptied, then the buffer size determines the amount of time available to fill the empty buffers for all the output channels in the system. For a buffer size of 256 words (the size of a disk block), this filling time is 167 ms or 10 revolutions of the disk. Buffer size can in theory be increased further, but the total buffer size required for the system quickly becomes excessive.

Another factor influencing buffer size requirements is related to the nature of the speech itself. Since each speech entry is stored in an integral number of disk blocks, there can be as much as 1/6 s trailing silence at the end of an entry. In order to control the amount of silence between spoken entries, this trailing silence should not be outputted to the coder. Instead, the coder must output the fraction of the buffer containing speech and then go right on to the next buffer, if no intermediate silence is desired. This means, however, that double buffering of the speech is inadequate, since a partial buffer may not permit enough time for the empty buffer to be refilled. Thus, control of output silence necessitates a third speech buffer for each output channel In order to maintain a continuous flow of speech with partial buffers, the voice response system must be able to make two disk reads for any channel with a partial buffer and still keep up with all the other channels. Since an average speech entry occupies 4 disk blocks, the average requirement for each channel is 1.25 reads of the disk in 10 revolutions, assuming a buffer length of 256 words.

Based on the above arguments, a goal of ten output lines was chosen for the voice response system. This number was adequate for any of the applications intended for the system yet would not overly tax the capabilities of the disk. Speech buffers of 256 words each were also chosen, in order to minimize core buffer requirements for the system. With these numbers, the average demand on the system is for approximately 12.5 disk reads in 10 revolutions of the disk. Depending on the status of the channels at a given time, the number of disk reads in 10 revolutions can vary from a minimum of 10 to a maximum of 20. In order to minimize the possibility of a disk read conflict, several steps were taken in the design of the output system. These techniques will be described later, along with more detailed timing measurements for the system.

MULTITASK PROGRAMMING CONSIDERATIONS

In order to handle up to 10 output channels in real time, a voice response system must be designed in such a way as to minimize any wasted idle time of the CPU. For example, after a speech buffer for a given channel has been filled, the system must immediately proceed to service the next channel needing a buffer. To accomplish this, it is necessary to synchronize the emptying and filling of speech buffers for each channel. On the other hand, the 10 output channels are completely independent of each other and should be kept isolated in the software. These requirements were met by using the real-time multitasking capabilities of Data General's Fortran [11]. These features include the ability to run several independent program units or tasks simultaneously on a priority basis, the ability to communicate data between tasks through Fortran Common areas, and the ability to synchronize two tasks or a task and an interrupt service routine by means of special tasking routines. In addition, all code generated by the compiler is reentrant, thus permitting different tasks to share the same code. Since these features of multitasking were relied upon heavily in the design of the output system, a brief discussion of the basic concepts of multitasking will now be given before describing the output system in detail.

In the programming sense, a task can be defined as a logically complete execution path through a program. A multitask program is then nothing more than a program which consists of several tasks, each executing independently of the others but competing with the others for the use of system resources. These resources, which include all peripheral devices and even the CPU itself, are allocated to tasks on the basis of their ability to use them. Active tasks may exist in any one of three possible states. They may be executing (in control of the CPU), ready (awaiting control of the CPU), or suspended (unable to use the CPU). When more than one task is ready to use a given resource, that resource will be allocated on a priority basis. The result of this scheduling procedure is an efficient utilization of all system resources.

In order to keep track of the tasks in a multitask program, a Task Control Block (TCB) is maintained for each task. The TCB contains status information for the task and several temporary storage locations, which are used to save the state of the processor whenever a task loses control of the CPU. The TCB's in a given program are linked together in order of priority. When several tasks exist at the same priority, the TCB for the task which most recently had control of the CPU is placed at the end of its priority class in the chain. This ordering of the TCB's in a chain facilitates the allocation of CPU time to tasks.

Allocation of system resources in a multitask program is performed by a multitask scheduler. Fig. 19 shows a simplified flow chart of this scheduler. CPU control returns to the scheduler whenever the currently executing task becomes suspended (e.g., when it is awaiting completion of an IO event), or whenever a hardware interrupt is serviced. The scheduler searches the chain of active TCB's to find the highest priority task in the ready state and turns over control of the CPU to that task. If no tasks are ready, the scheduler simply waits until one becomes ready. Because of the manner in which the TCB's are linked, tasks of equal priority receive CPU control on an equal or "round-robin" basis. This feature

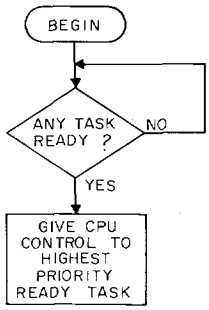

Fig. 19. Block diagram of multitasking allocator.

is convenient for programs which consist of several identical tasks, such as the output program for the voice response system.

Intertask communication can be accomplished in several possible ways. The most direct approach is for different tasks to share the same data base, either through a common disk file or a Fortran Common area. However, this method provides no synchronization between tasks and may result in a task trying to access data from another task which is not yet available. To overcome this problem, tasks can be synchronized by the sending and receiving of one-word messages.

Whenever an executing task must wait to receive a message from another task, the receiving task becomes suspended, and control returns to the task scheduler. When the message is then sent at a later time, control returns to the scheduler, which readies the receiving task which had previously been suspended. Thus, no CPU time is wasted while a task is waiting for an event to occur. This same technique for task synchronization can also be used to synchronize a task and an interrupt service routine. Normally, the task environment becomes frozen whenever an interrupt occurs. However, if an interrupt service routine sends a message to a task which is currently awaiting the message, that task will be readied before control returns to the scheduler. This feature of multitasking makes it possible to activate a task upon the occurrence of an external interrupt. Thus, intertask communication in a multitask program is greatly facilitated by the availability of several tasking routines in the Fortran library.

Fig. 20 illustrates a block diagram for a typical channel in the output system. The software for each channel consists of three distinct program units. The first of these units, the FEED task, must access the information to be converted to speech and load it into a Common buffer area. This module will be different for each application of the voice response system. The second program unit, the OUTPUT task, controls the output speech by making the appropriate disk reads to fill the speech buffers. The information obtained by the FEED task is used to determine which blocks of speech from disk are to be read. Finally, the third program unit is the driver for the speech coder. This program must direct the coder to the next full buffer of speech to be outputted and signal the OUTPUT task to fill any empty buffers. Since the drivers were not

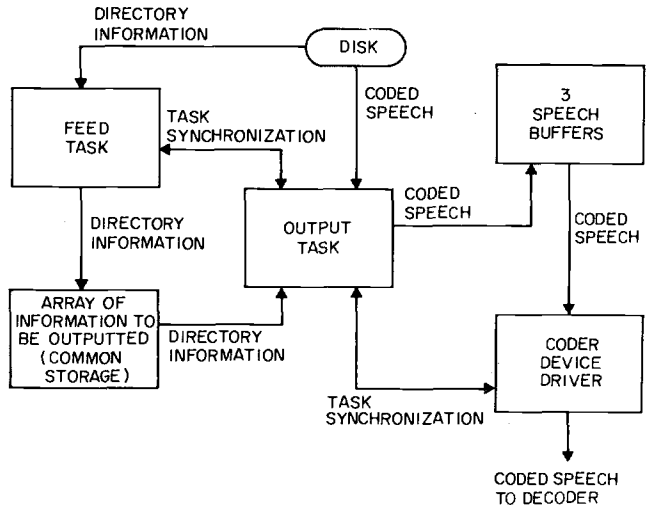

Fig. 20. Block diagram of a typical output channel.

written as reentrant subroutines a separate one is necessary for each output channel. However, most of the code for the other two program units can be shared by all channels.

CODER DEVICE DRIVER

In order to make full use of the three buffers which are required for each channel of the output system, triple buffering is used in the device driver. A flow chart of the coder interrupt service routine is shown in Fig. 21. When servicing an interrupt, the routine first checks a status flag to see if the next buffer to be outputted is full. If it is not, the coder is directed to a 256 word buffer of coded silence, and the interrupt is dismissed. If the next buffer has been filled, the service routine first changes the status of that buffer to "being emptied." It then changes the status of the buffer that has just been outputted to "being filled." Next, the routine gets the address and buffer length of the next full buffer and starts the coder. Finally, a message is transmitted to inform the OUTPUT task associated with that channel that a new buffer of speech will be needed, and the interrupt is dismissed.

The use of a buffer of silence whenever the next speech buffer is not full serves two purposes in the system. First, it indicates the error condition which occurs if the OUTPUT task cannot keep up with the coder. More importantly, however, it is used to insert deliberate silence between spoken entries. Normally, the OUTPUT task sets a buffer's status to full as soon as the buffer has been filled. When silence is desired, however, the buffer is filled but its status is not immediately set to full. This causes the coder service routine to insert silence in the output speech. The duration of the silence is determined by the delay between the filling of the speech buffer and the setting of its status flag. This duration is controlled by a real-time clock. Since coded silence is obtained from a buffer in core instead of from the disk, periods of silence reduce the number of disk read requests for a given output channel.

Triple buffering of the output speech was selected in order to be able to handle partial buffers. Normally, there is a full buffer of speech between the buffer being emptied

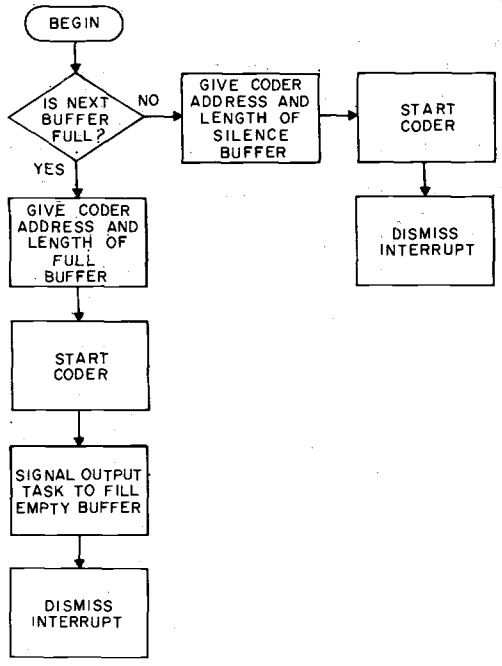

Fig. 21. Block diagram of coder service routine.

and the buffer being filled. Therefore, when a partial buffer comes along, the next buffer will have already been filled. Initially, after the outputting of a partial buffer, there may no longer be a full buffer between the two buffers being emptied and filled. However, since partial buffers occur only about once every fourth buffer, the OUTPUT task will have time to catch up with its filling of buffers before the next partial buffer occurs. Periods of silence in the output speech also help the OUTPUT task to keep up with the filling of empty buffers.

Communication of data between the device driver and the output task is accomplished through Fortran Common storage. Synchronization is accomplished by means of one-word messages transmitted by the interrupt service routine.

THE OUTPUT TASK

Basically, the function of the OUTPUT task is to control the output of speech on each of the voice channels. Input to the task is a 128 word buffer containing the information to be outputted. Each entry in this buffer is a number specifying where in the speech directory the entry can be found. This format eliminates the need for directory lookups while the speech is being outputted but at the same time minimizes the size of the information buffer. The numbers -1, -2, and -3 are also possible entries in the information buffer. The numbers -1 and -2 are used to select two different silence intervals in the output speech, and -3 designates the end of the message. The information buffer for a given channel must be filled before any speech output is possible on that channel. The means by which this buffer is filled depends, of course, on the particular application of the voice response system. The next section will discuss this procedure for the computer-aided voice wiring application.

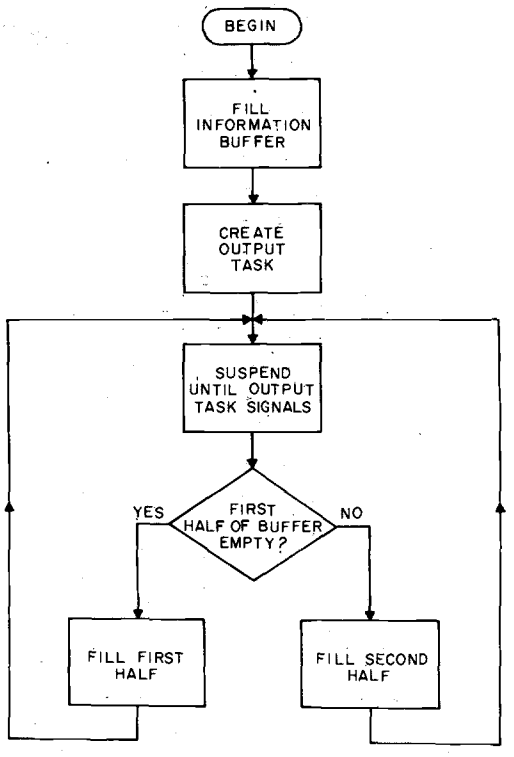

Fig. 22. Block diagram of OUTPUT task.

Fig. 22 shows a flow chart for the OUTPUT task. The speech directory resides in core while this task is executing. The normal sequence of operations is to fill a speech buffer with the next desired disk block of speech, set the buffer status to "full," and check the status of the next buffer. If the status is "being filled," the buffer is filled. If it is not, the task is suspended until a synchronizing message is issued by the coder interrupt service routine. The OUTPUT task then checks the buffer status again, taking action as above. This procedure ensures both that all empty buffers will be filled quickly and that the OUTPUT task will suspend itself after all buffers have been filled, thus freeing the CPU to allow it to service OUTPUT tasks for the other speech channels.

The logic for the OUTPUT task becomes more complex when it must consider the possible alternatives when the last block of a speech entry has been read into a speech buffer. First, the program checks the next entry in the information buffer. If the number is greater than 0, implying an entry in the speech directory, the program simply accesses this entry and continues as above. If the number is -3, no more speech needs to be accessed, and the OUTPUT task can be suspended. If, however, the number is either -1 or -2, the program records the fact that a silence interval will begin after the next speech buffer has been filled and then proceeds to the next entry. Silence is effected by filling the next buffer but not setting its status flag to full until after a specified number of real-time clock ticks, each of which is 1/10 s apart. Prior to beginning the time delay, the program checks to see if it has just finished outputting either the first or second half of the information buffer. If it has, it issues a message to the

FEED task to signal that half of the buffer can now be refilled, if necessary. If the information buffer is loaded from disk, the next access will then be made at a time when the OUTPUT task for that channel would normally be accessing speech from the disk. Therefore, no additional load is placed on the system by these additional disk accesses. At the end of the silence interval, the OUTPUT task sets to "full" the status flag for the buffer that was last filled and continues as before.

In order to minimize program size, all of the logic for the OUTPUT task is implemented as a reentrant Fortran subroutine, with the output channel number as an argument. Therefore, the OUTPUT task for each channel consists of a program which does nothing more than call this subroutine, with the appropriate channel number as an argument. This results in a significant reduction in code for the output system.

THE FEEDING TASK

The function of the FEED task is to prepare the buffer of information which will be outputted as speech. By way of illustration, the FEED task has been implemented for the application to computer-aided voice wiring. This application was selected for several reasons. First, it is a fairly direct application to implement. At the same time, however, this application represents a worst case with which to test the system. Since the generation of spoken wirelists does not depend on human interaction with a computer, it is fairly straightforward to set up the voice response system with ten simultaneous output lines that can be monitored for errors. In addition, this application does not require any additional special equipment, such as telephone answering hardware. Thus, the FEED task to be discussed here serves both as a means to check out the voice response system and as an example of how such a task must be written in order to interface with the rest of the system software.

Fig. 23 shows a flow chart for the current implementation of the FEED task. Prior to invoking the output program, wirelist information is stored in a disk file in a form suitable for the OUTPUT task. Thus, in this application the FEED task must simply keep the core information buffer full. Initially, the FEED task receives program control before the OUTPUT task is activated. This task then fills the information buffer, activates the OUTPUT task, and suspends itself. Later, when the OUTPUT task issues a message to the FEED task signaling that half of the information buffer has been emptied, the FEED task becomes readied, loads the next 64 entries of the wirelist into the empty half of the information buffer, and again suspends itself. This process continues until the entire wirelist has been outputted.

Preparation of a wirelist before it is stored on the disk is accomplished by a separate program called Wirelist. Input to this program is a wirelist deck such as the one listed in Fig. 24. The beginning columns of each card are reserved for sequence numbers, if desired. After the sequence numbers, wirelist entries are listed in the same

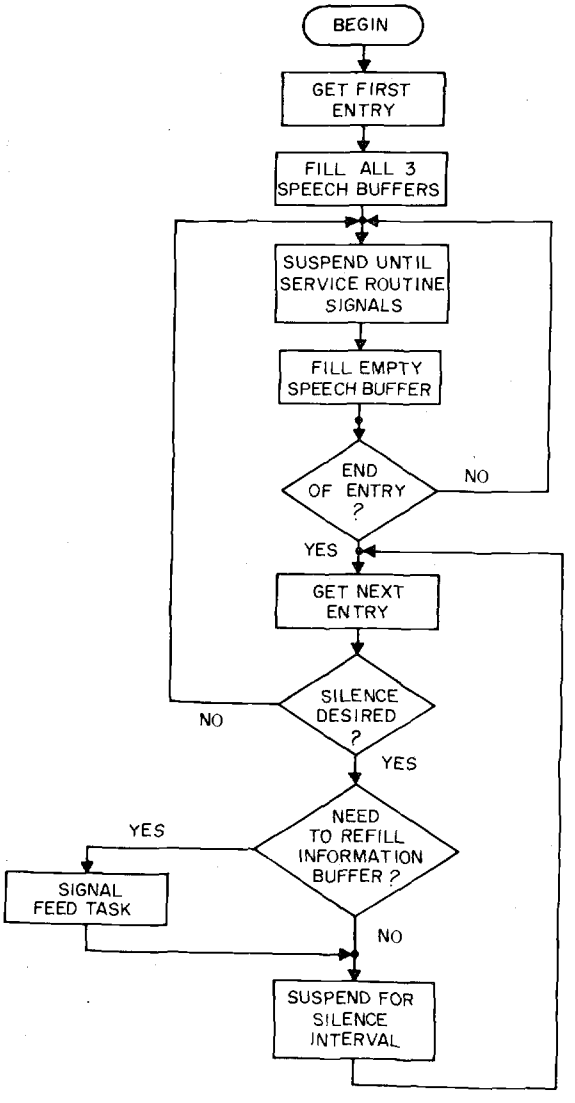

Fig. 23. Block diagram of FEED task.

```
A250 $ LVL 1  $  UVS  3  BAY  2  $   PLT   10   CRK   10
A251 $ LVL 2  $  UVS  3  BAY  0  $   PLT   5    CRK   30
A252 $ LVL 3  $  UVS  5  BAY  0  $   PLT   12   CRK   30
A253 $ LVL 4  $  UVS  4  BAY  0  $   PLT   14   CRK   20
A254 $ LVL 5  $  MIS  19 $  PLT  15   CRK   0    .  0
```

Fig. 24. Typical wirelist.

ASCII form in which they appear in the speech directory. Entries are separated by one or more spaces, and may continue through column 79. Column 80 is reserved for a continuation symbol. Silence between entries can be obtained by inserting a special silence character into the list wherever it is desired. For the list in Fig. 24 this character is a "/". Silence will automatically be placed after the entries on each card, unless the continuation symbol is used. The end of a wirelist is denoted by an end of file symbol placed in column 1 of the last card. The Wirelist program isolates each separate entry on the cards, looks it up in the speech directory, and stores on disk only the location of the entry in the speech directory. The appropriate negative number is stored instead whenever

an entry denotes either silence or the end of the list. Thus, each wirelist entry can be stored as a single number in the disk file.

As was the case with the OUTPUT task, most of the logic for the FEED task is included in a reentrant subroutine shared by all channels. However, each channel can have its own, totally independent wirelist. Separate information buffers and disk files are maintained for each channel, in order to eliminate the possibility of interaction between the channels.

Other applications for the voice response system will require a different FEED task, but the linkage to the OUTPUT task will remain the same. In applications where the information buffers need to be filled only once, the message requesting a new information buffer can simply be acknowledged without taking any action. Since many applications for voice response systems involve interaction with humans, who are slow in comparison with the system, the successful operation of a ten channel system with voice wiring output assures that other less demanding applications can be handled with the same degree of success.

DISCUSSION

Using the ideas discussed in this paper a 10 line voice response system was implemented. With up to ten simultaneous output lines of speech, no errors were detected on any of the channels. The natural question, therefore, is how many additional lines can be supported by such a system. Although insufficient hardware was available to answer this question experimentally, a look at some of the numbers for this system should give some insight into its maximum capabilities.

One practical limitation on the maximum capacity of the voice response system is the amount of available core memory. For a ten channel voice response system the core requirements are roughly as follows. The system software, which occupies the uppermost locations in core, requires approximately $5K$ words of core memory. The software for the voice response system itself requires approximately $7K$ words of core memory, which includes all the necessary Fortran library routines. In addition, each output channel requires about 900 words of memory for core buffers (3×256 or 768 words for the speech buffers and 128 words for the information buffer). For ten channels, this adds an additional requirement of $9K$ words for core buffers, thus bringing the total so far up to $21K$ words of core memory. However, this number does not include the space for the Fortran runtime stacks. These stacks provide temporary storage for all non-Common variables in a Fortran environment. In a multitask program, each task is given an equal amount of the runtime stack area for its own stacks. This means that the amount of core space available for stacks must be at least as large as the product of the number of tasks and the size of the largest stack required by any of the tasks. For the voice response system, which consists of 21 tasks (10 FEED tasks, 10 OUTPUT tasks, and an initializing task) and has available $32K$ words of core memory, each task has approximately 500 words of memory for its stacks. It is not known exactly how much of this storage area is needed, but the important point is that each additional channel added to the system would reduce the total stack area by about 1000 words of memory and at the same time reduce the fraction of the total area that would be allocated to each task. Thus, in terms of available core storage, the current implementation of the voice response system is probably running very close to its maximum capabilities.

Another limitation on the voice response system is the amount of processor time available to support all the output channels in real time. For the 10 channel voice response system, it has been found that the processor is actually running the system about 33 percent of the time. The rest of the time is still available for running other tasks, such as additional output channels. Thus, processor time is not a serious limitation on the system, since other factors would tend to limit the system capabilities before this one becomes prominent. The following discussion provides a rough estimate of how the processor time is divided while running all 10 output lines. For the purpose of simplification, the special cases of partial buffers and silence between entries have been ignored in these calculations.

Data channel transfers, which steal computer cycles in order to transfer data from the disk to core or from core to a speech decoder, consume approximately 3 percent of the available processor time with a 10 channel system. This time is divided evenly between disk transfers to core and core transfers to the decoders. On the next higher level, interrupt servicing consumes approximately 1 percent of the processor time with a 10 channel system, including system overhead. The remaining 29 percent of the time is spent actually running the programs. Of this time, between 12 and 25 percent is spent running the system in order to set up the reads of the disk. This is a fairly high degree of system overhead, but it is tolerable in this application because of the small amount of processing required in the user programs. Summing up, it is seen that processor time does not impose a very strict limitation on the maximum capabilities of the voice response system.

The most serious limitation on the voice response system is imposed by the properties of the disk. In the current implementation, disk reads are scheduled in the same order that they are requested. As a result, on a given revolution of the disk between 0 and 4 reads will be made, depending on the number and order of the read requests. This algorithm is adequate for a 10 channel system, where the average demand is for 12.5 reads in 10 revolutions, but as the number of output channels is increased, this scheduling algorithm will become inadequate. A more sophisticated queueing algorithm could be developed, but such an algorithm would have to interface with the asynchronous OUTPUT tasks in order to be able to rearrange

read requests. However, a more basic problem with the disk is that it has only 8 sectors. Even with perfect disk read scheduling, nothing can be done if too many read requests are made for the same sector of the disk. The probability of this occurrence is small for a ten channel system, but as more channels are added it becomes a real problem.

Another minor problem similar to the disk conflict problem is that of missing an interrupt from a decoder. When a decoder requests an interrupt, it must be serviced within 667 μs in order to maintain a continuous flow of speech data. This is no problem when handling a single interrupt, but should several decoders request interrupts at approximately the same time, the last one to request the interrupt may not be serviced within the required time. The probability of such an event occurring is very small, since only 0.1 percent of the processor time is spent servicing an interrupt for a single decoder, but again as the number of output channels increases, so does this probability.

The preceding paragraphs have described several factors which tend to limit the maximum number of output lines in a voice response system. However, as has been stated previously, none of these factors affected the performance of the 10 channel voice response system which was implemented.

SUMMARY

The preceding sections have described the components of a 10 line automatic voice response system. In the input side of the system, the most important feature is the ADPCM coding of speech. This coding technique makes possible both an efficient storage of speech on the disk and a means for editing the coded speech automatically. For the output side of the system, the most important feature is the capability for real-time multitask programming. This feature makes it possible for the CPU to run as many as 21 independent tasks, service interrupts for a disk, a real-time clock, and 10 decoders, and perform data channel transfers for the disk and 10 decoders in such a way that these different operations appear to be going on simultaneously. The result of this multiprogramming facility is a system which can produce ten simultaneous output lines of speech in real time. This system was implemented for the application of computer-aided voice wiring, but the design is sufficiently flexible to permit other similar applications. Finally, timing measurements for this system were made and evaluated, in order to determine the performance capabilities of the system. The results of these tests indicated that the voice response system design was fairly conservative and could probably handle more output channels. However, since a 10-channel capability was adequate for all of the intended applications of this system, no attempt was made to surpass this original goal.

ACKNOWLEDGMENT

The authors wish to thank the hardware and software assistance provided by K. L. Shipley, and D. E. Bock of Bell Laboratories, and R. H. Rabiner and G. Franzen of Data General Corporation in implementing the voice response system.

REFERENCES

[1] A. Newell *et al.*, "Speech understanding systems," Carnegie-Mellon Rep., May 1971.
[2] J. L. Flanagan, C. H. Coker, L. R. Rabiner, R. W. Schafer, and N. Umeda, "Synthetic voices for computers," *IEEE Spectrum*, vol. 7, pp. 22–45, Oct. 1970.
[3] J. L. Flanagan and L. R. Rabiner, *Speech Synthesis*. Stroudsburg, Pa.: Dowden, Hutchinson, and Ross, 1973.
[4] L. R. Rabiner, R. W. Schafer, and J. L. Flanagan, "Synthesis of speech by concatenation of formant-coded words," *Bell Syst. Tech. J.*, vol. 50, pp. 1541–1558, May–June 1971.
[5] J. L. Flanagan, L. R. Rabiner, R. W. Schafer, and J. D. Denman, "Wiring telephone apparatus from computer-generated speech," *Bell Syst. Tech. J.*, vol. 51, pp. 391–397, Feb. 1972.
[6] N. S. Jayant, "Digital coding of speech waveforms," *Proc. IEEE*, vol. 62, pp. 611–632, May 1974.
[7] R. A. McDonald, "Signal-to-noise and idle channel performance of differential pulse code modulation systems—particular applications to voice signals," *Bell Syst. Tech. J.*, vol. 45, pp. 1123–1151, Sept. 1966.
[8] N. S. Jayant, "Adaptive delta modulation with a one-bit memory," *Bell Syst. Tech. J.*, pp. 321–342, Mar. 1970.
[9] P. Cummiskey, "Adaptive differential pulse-code modulation for speech processing," Ph.D. dissertation, Newark College of Engineering, Newark, N. J., 1973.
[10] P. Cummiskey, N. S. Jayant, and J. L. Flanagan, "Adaptive quantization in differential PCM coding of speech," *Bell Syst. Tech. J.*, vol. 52, pp. 1105–1118, Sept. 1973.
[11] *Fortran IV User's Manual*, Data General Corporation, Southboro, Mass., 1973.