CODE-EXCITED LINEAR PREDICTIVE CODING FOR HIGH-QUALITY AND LOW BIT-RATE SPEECH

by

Armein Langi

A Thesis

presented to the University of Manitoba

in partial fulfillment of

the requirements of the degree of

Master of Science

in

the Department of Electrical and Computer Engineering

Winnipeg, Manitoba

April, 1992

© Armein Langi, 1992

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77775-3

Canada

CODE-EXCITED LINEAR PREDICTIVE CODING FOR HIGH-QUALITY AND LOW BIT-RATE SPEECH

BY

ARMEIN LANGI

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

© 1992

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's permission.

ABSTRACT

This thesis presents a study of code-excited linear predictive (CELP) coding to compress telephone-quality speech from 64 kbit/s down to 4.8 kbit/s bit rate (13:1 compression ratio), while preserving natural speech-quality. The compression is studied through (1) a utilization of a compact stochastic code book (SCB) resulted from (a) redundancy removal by an extraction of pitch and vocal tract information and (b) segmentation and normalization, and (2) a compact representation of the pitch and vocal tract information by (a) an adaptive code book (ACB) and a 10-order all-pole linear prediction (LP) filter, and (b) delta coding and line spectrum pair (LSP) quantization of ACB and LP parameters, respectively. The quality is preserved by an analysis-by-synthesis scheme through the use of (1) waveform similarity in a Euclidean sense as a compression criterion, and (2) a perceptual weighting filter to include frequency-domain closeness to the criterion. Parameter smoothing, a stability rule, data scattering, and a Hamming (15,11) code improve coding robustness. The CELP has been implemented on a low-cost personal-computer (PC-AT) equipped with a TMS320C30 evaluation module (EVM) for experimentation. To facilitate this implementation, fast algorithms and a digital signal processor had to be used. The fast algorithms include (1) incorporating the LP and perceptual weighting filters, (2) serial searching, (3) joint optimization scheme, and (4) 77% sparsity, ternary, and overlapping SCB, reducing the computation of an exhaustive closed-loop code book search by a factor of more than 175,000. The signal processor architecture includes a fast CELP engine, which was efficiently coded to fit within the 16K word available memory on the EVM. Performance measure using (1) word intelligibility, (2) sentence intelligibility, (3) subjective quality, and (4) objective quality tests, employing rhyme and category-judgment techniques with Fairbanks words and Harvard sentences results in high intelligibility (95.67% word correct identification), a mean opinion score (MOS) of 3.21, and a segmental signal-to-noise ratio (SEGSNR) of 10.10 dB.

ACKNOWLEDGEMENTS

First, I would like to thank God for all the good things I have enjoyed. This thesis is one of them. This thesis is also a result of a long and continuing study, involving interactions with many people. It is a pleasure to acknowledge the people that make the completion of this thesis possible. I would like to thank Dr. W. Kinsner, my advisor and mentor, for his invaluable guidance and help, as well as his patience during the thesis work. Dr. Kinsner has introduced to me a very exciting world of research and engineering, and at the same time prepared me through persistent guidance to enter it.

Thanks to Ken Ferens, my friend and colleague, who helped me not only in technical and writing aspects of this thesis, but also in friendship and encouraging support during difficult times. The completion of this thesis is unimaginable without his support.

Thanks to Adi Indrayanto, Budi Rahardjo, Larry Wall, and Warren Grieder, my friends and colleagues, who have been willing to help throughout the process of the thesis work.

Thanks to Dr. McLeod for the access to the TMS320C30 development system, and Prof. Yunik for the access to Music Lab. Access to experimental equipment has been made possible by Budi Rahardjo. Thanks to Mrs. Ilse Krentz for providing her wonderful voice for the test set. Thanks also to the 11 people who participated in the listening test.

I devote this thesis to my wife, Inawati, for her constant love, understanding, and prayers. This thesis is also for my two beautiful daughters, Gladys and Kezia, who wait for their father in Indonesia, and probably wonder why their father chose to work with this thesis for so long rather than to play with them.

TABLE OF CONTENTS

	Page
ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	ix
LIST OF TABLES	xi
LIST OF ABBREVIATIONS AND ACRONYMS	xii
I INTRODUCTION	1
Background	1
Problem Definition	5
Thesis Statement	5
Problem Specifications	6
Thesis Organization	7
II CODE-EXCITED LINEAR PREDICTIVE CODING	8
Review of Speech Compression	8
Principles of Data Compression	8
Difficulties in Speech Compression	10
Features of Speech Compression	10
Speech Intelligibility	10
Speech Quality	11
Implementability	11
Methods of Speech Compression	12
Waveform Coding	12
Source (Parametric) Coding	13
Filter Bank Coding	15
Transform Coding	15
Summary of Some Important Techniques	16
Basics of CELP Technique	17
Speech Production Model for Reducing Waveform Variability	19
A Human Speech Production System	19
Modelling of the Vocal Tract	21

Modelling of the Excitation Generator 25
Combining the Models of Vocal Tract and the Excitation Source 28
Pitch Filter 28
Residues (Residual Impulses) 30
CELP Speech Synthesis 31
Discussions on Code Book Design 33
Adaptive Code Book as a Pitch Filter 34
Summary of CELP Parameters 35
CELP Speech Analysis 35
Estimation of the LP Parameters 36
Problem Setup 37
Solution Using LPC Analysis 38
Estimation Enhancement using Hamming Window 40
Estimation of Code Book Parameters 40
Analysis by Synthesis 41
Perceptual Weighting Filter to Improve Distance Measure 42
Quantization of CELP Parameters 44
Quantization of LP Parameters 45
Quantization of Code Book Parameters 49
Bit Allocation of CELP Parameters for Transmission 50
A Strategy for Error Protection 50
Protection of LP Parameters 51
Protection of SCB Gain 52
Protection of SCB Entry 52
Protection of ACB Entry 52
Protection of ACB Gain 53
Forward Error Correction 53
Burst-Error Protection 53
Summary 54

III FAST ALGORITHMS FOR REAL-TIME CELP SYSTEMS 55
Restructuring of CELP Analysis 56
Joint Optimization Scheme 60
Special SCB Design 63
Creating a Special SCB 63

Effects on the Convolution Computation 64
Partial Searching 67
Computational Power Requirement for Code Book Searching 67
Summary 68

IV SYSTEM DESIGN AND IMPLEMENTATION 69
System Architecture 70
Controller 74
User Commands for Speech Communication 75
User Commands for Speech Storage 76
Command Control Phases 77
Data Processor 77
I/O Processor 78
System Organization 79
Hardware Organization 80
Capabilities of Hardware Platform 80
Block Distribution 82
Software Organization 82
Controller 82
Data Processor 84
EVM Controller 84
Data Acquisition and Distribution from A/D and to D/A Converters 86
CELP Analyzer 88
CELP Synthesizer 90
Error Protection, Data Stream Arrangement, and Data Recovery 91
HOST-EVM Data Exchanges 92
I/O Processor 92
Packet Radio Channel 93
Speech Storage 95
System Implementation 96
System Development Method 96
Hardware Setup 97
Software Development Method 97
Simulation 98
Actual Software Coding 98

EVM Routine Coding 99
Host Routine Coding 99
Integration 100
System Tests and Adjustments 100
System Realization 101
Host Controller 101
Host Serial Port Programs 102
File Access Programs 102
EVM Controller 103
Analog Data Access 104
CELP Speech Analysis 104
CELP Speech Synthesizer 106
Error Protection and Bit Scattering 106
Benchmarking of the Codes 107
Summary 108

V EXPERIMENTAL RESULTS 110
Performance Measurements 110
Features for Measurements 110
Validity of the Measurements 111
Methods of Measurements 112
Word Intelligibility Test 112
The Measure 112
Test Procedure 113
Sentence Intelligibility Test 114
The Measure 114
Test Procedure 115
Subjective Quality Test 116
The Measure 116
Test Procedure 117
Objective Quality Test 118
The Measure 118
Test Procedure 118
System Performance 119
System Intelligibility 119

System Quality 120
Summary 120

VI CONCLUSIONS AND RECOMMENDATIONS 124

REFERENCES 128

APPENDIX A: PROPOSED FEDERAL STANDARD 1016 . . . . A 1
APPENDIX B: SOFTWARE LISTING . . . . . B 1
APPENDIX C: SETS OF UTTERANCES AND QUESTIONAIRE . C 1

LIST OF FIGURES

Figure		Page
1.1.	A real-time speech communication system using a low bit-rate RF channel with a capacity as low as 6 kHz.	2
2.1.	A speech compression system.	9
2.2.	A code book of prototypes of speech waveforms, used as a synthesizer.	17
2.3.	A CELP analyzer using an analysis-by-synthesis scheme.	18
2.4.	A speech production model.	20
2.5.	A model of speech production system.	22
2.6.	Frequency response of a vocal tract.	22
2.7.	Speech spectrum as a result of envelope shaping of the spectrum of the excitation pulses by the vocal tract.	23
2.8.	Piecewise-cylindrical approximation of vocal tract with closed velum.	24
2.9.	A direct implementation of a vocal tract (LP) filter.	25
2.10.	A typical voiced excitation in (a) time domain, and (b) frequency domain.	26
2.11.	A typical unvoiced excitation in (a) time domain, and (b) frequency domain.	27
2.12.	Trajectories of the first and the second formants of three different articulations.	28
2.13.	A speech production model in a digital form.	29
2.14.	A one-tap pitch predictor filter.	30
2.15.	Waveforms of different signals.	31
2.16.	A CELP synthesizer.	32
2.17.	(a) A pitch filter, and (b) its equivalence using a code book form.	34
2.18.	A CELP synthesizer.	35
2.19.	CELP analysis.	36
2.20.	Windowing of speech data prior to autocorrelation process.	41
2.21.	Basic scheme of code book parameter searching.	42
2.22.	Frequency response of the weighting filter for a given vocal tract filter.	44
2.23.	An example of locations of zeros of $A(z)$.	45
2.24.	An example of locations of zeros for m=10.	45

3.1. An improved code book parameter searching. 56
3.2. An efficient structure of code book parameter searching for CELP analysis. 59
3.3. A code book searching diagram. 61
3.4. A joint optimization scheme. 63

4.1. A basic architecture of a CELP system for packet radio. 71
4.2. A CELP system with its main parts. 71
4.3. A basic architecture of a CELP system from a signal processor perspective. 72
4.4. The CELP system architecture. 73
4.5. The configuration of the hardware platform. 81
4.6. A flowchart of the controller. 83
4.7. A flowchart of the A/D and D/A interrupt service. 87
4.8. A flowchart of a CELP analyzer. 89
4.9. A flowchart of a CELP synthesizer. 90
4.10. Flowchart of the error protection and data stream schemes
during (a) data transmission, and (b) data reception. 91
4.11. A flowchart of the RS232C interrupt services. 94

LIST OF TABLES

Tables		Page
1.1.	Comparison of quality and bit rate of several speech coders.	3
2.1.	Bit allocation for transmitting CELP parameters.	50
3.1.	A table for construction an SCB.	64
3.2.	An estimation of computation power needed for implementing fast algorithms.	68
4.1.	Commands of the data processor to perform CELP compression and decompression.	78
4.2.	Commands for I/O processor.	80
4.3.	User commands.	85
4.4.	List of processes performed by the EVM.	86
4.5.	One frame execution time of current version.	107
4.6.	Domination of inner product computation.	108
4.7.	An example of the effect of assembler code to reduce the execution time.	108
5.1.	Difficulty levels and their scores.	113
5.2.	Percentage of correct identifications of every consonant.	120
5.3.	Articulation types of 7 most sensitive consonants.	121
5.4.	Seven first consonants of most mis-classified words from sentence intelligibility test.	121
5.5.	Subjective quality of the system.	122
5.6.	Objective quality of the system.	122

LIST OF ABBREVIATIONS AND ACRONYMS

A/D	Analog to Digital
ACB	Adaptive Code Book
ADPCM	Adaptive Differential Pulse Code Modulation
AM	Amplitude Modulation
AX-25	Amateur X-25
bps	Bit Per Second
CB	Code Book
CELP	Code-Excited Linear Predictive
CPU	Central Processing Unit
CVSD	Continuously Variable Slope Delta Modulation
D/A	Digital to Analog
dB	Decibel
DMA	Direct Memory Access
DOS	Disk Operating System
DSP	Digital Signal Processing
EVM	Evaluation Module
FEC	Forward Error Correction
FM	Frequency Modulation
FS-1016	Federal Standard # 1016
HF	High Frequency
IBM	International Business Machine™
IC	Integrated Circuit
IEEE	The Institute of Electrical and Electronics Engineers, Inc
I/O	Input/Output
Hz	Hertz
k	1000

K	$2^{10} = 1024$
kbit/s	kilo bit persecond
LAN	Local Area Network
LP	Linear Predictor
LPC	Linear Predictive Coding
LSB	Least Significant Bit
MADD	Multiply and Add
MIPS	Million Instructions Per Second
MOS	Mean Opinion Score
MSB	Most Significant Bit
N.O.T.	Number Of Trial
ns	NanoSecond
PC-AT	Personal Computer AT™
PCM	Pulse Code Modulation
RF	Radio Frequency
ROM	Read Only Memory
RS-232	Recommended Standard #232
s	Second
SCB	Stochastic Code Book
SEGSNR	Segmental Signal-to-Noise Ratio
SNR	Signal-to-Noise Ratio
TCP/IP	Transmission Control Protocol/Internet Protocol
TNC	Terminal Node Controller
UART	Universal Asynchronous Receiver Transmitter
UHF	Ultra High Frequency
US FS-1016	United States Federal Standard #1016
VHF	Very High Frequency
WAN	Wide Area Network

CHAPTER I
INTRODUCTION

1.1 Background

Transmission of high-quality speech over long-distance radio is a problem of considerable importance in voice communication research [Jaya90]. In the past, voice has been transmitted in real time using various analog forms such as amplitude modulation (AM) or frequency modulation (FM). However, the quality of analog voice transmitted over long distances deteriorates rapidly, and usually becomes too noisy after three repeaters. Consequently, real-time transmission of digital signals is considered superior to its analog counterpart due to the ability to reconstruct the signals at each repeater completely, thus achieving a higher immunity to noise [Kins89a]. In addition, the digital transmission can use error protection schemes that increase the robustness of the transmitted signals. Finally, digital transmission of speech can be used for secure communication. For example, digital data can be packetized and scrambled for private communication.

The digital form of speech is also attractive for speech storage applications such as voice mail. It is attractive because digital storage does not have moving parts, as well as reproducibility of speech sound in digital form may be high, and access speed to the stored speech may be in real time.

Digital speech may be transmitted on radio frequency (RF) channels (numerous band regions) using packet radio, including narrow band RF such as high frequency (HF), very high frequency (VHF), and ultra high frequency (UHF) [Kins89b]. In addition to being a broadcast channel, the RF channel is not as expensive as a wired channel, such as the coaxial cable. A packet radio unit usually consists of a small computer, a packet radio *modem* with protocol controller called terminal node controller (TNC), and a radio transceiver. It can be very portable and be placed almost anywhere easily.

Commercial or amateur packet radio may operate in either individual (point-to-point) or network mode. The point-to-point mode is achieved simply by connecting two packet radios. On the other hand, packet radio can be integrated into the existing local area network (LAN) or wide area network (WAN) to make use of the wide connections all over the world, since it can use standard protocols such as AX.25 and/or TCP/IP. Besides that, the packet-radio network itself grows widely among the members of the amateur radio and commercial telephone companies, providing service to amateur radio members and remote rural costumers, respectively [JoMc90], [EsRa91], [Karn89], [KuFT91].

The main problem of the real-time mode of digital speech transmission using packet radio is how to transmit speech data through a low capacity (6 kHz) RF channel in the HF region. The digitized pulse code modulation (PCM) form of telephone-quality speech requires 64 kbit/s, due to the bandwidth of 300 to 3300 Hz, sampling at 8 kbit/s, and 8-bit resolution to cover the dynamic range of the speech of either 48 dB without analog signal compression or more (72 dB) with A-law or μ-law compression. Therefore, transmitting and receiving 64 kbit/s data through the 6 kHz capacity channel is impossible. Thus, an appropriate speech compression technique must be used to code the speech data into a form that is suitable for real-time communication through packet radio (see Fig. 1.1).

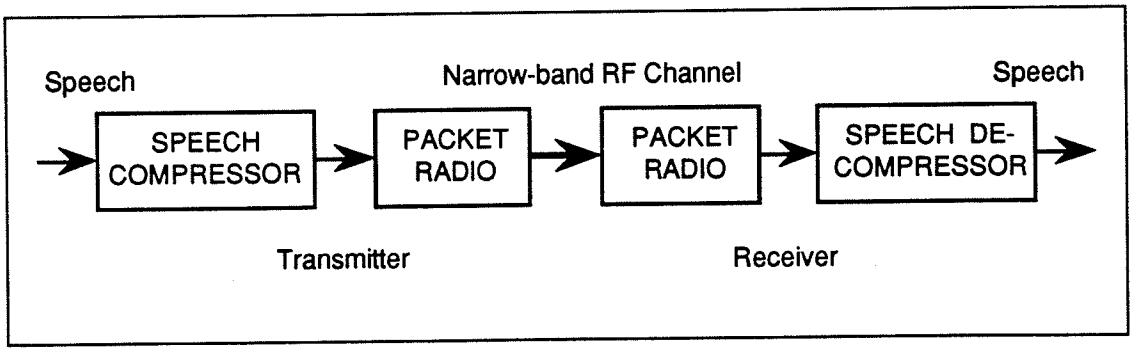


Fig. 1.1. A real-time speech communication system using RF channel with a capacity as low as 6 kHz. The system needs speech compression because telephone quality speech requires a rate of 64 kbit/s.

Another problem of packet radio transmission is how to protect the compressed speech data from noise. Since the compressed data have smaller redundancy, a few errors in the transmitted data may cause significant damage to the reconstructed data. Thus, an implementation of a forward error protection scheme becomes necessary in the packet radio transmission.

Considerable effort has been directed towards digital speech transmission using low bit-rate channels in real time [Klim87], [Swan87], [Kins91], [LaKi91a], [LaKi91b], [LaKi90a], [LaKi90b]. In most cases, there is a trade-off between the quality of speech and the bit rate (see Table 1.1). Techniques such as adaptive differential pulse code modulation (ADPCM) and continuously-variable slope delta modulation (CVSD) are well known for natural quality of the resulting speech. However, their high bit rates (≥16 kbit/s) are not suitable for the HF channels [KlKi87]. On the other hand, linear predictive coding (LPC) can successfully reduce the bit rate of a real-time speech transmission down to 2.4 kbit/s, but the reported quality is poor according to a subjective listening test [Swan87].

Table 1.1. Comparison of quality and bit rate of several speech coders.

Coders	Quality	Bit rate (kbit/s)	Storage for 60–s speech (bytes)
Pulse Code Modulation (PCM)	Natural	64 – 96	≥ 480,000
Adaptive-Differential PCM (ADPCM)	Natural	16 – 32	≥ 120,000
Continously-Variable Slope Delta Modulation (CVSD)	Natural	16 – 40	≥ 120,000
Code-Excited Linear Predictive (CELP)	Natural	4.8	36,000
Linear Predictive Coding (LPC)	Synthetic	2.4	18,000
Phonetic	Synthetic	0.2	1,500

The emergence of code-excited linear predictive (CELP) speech coding enables an implementation of a real-time, digital speech processing system for transmission of high-quality speech using packet radio [ScAt85]. This technique produces high-quality speech at rates as low as 4.8 kbit/s, which is comparable to the continuously-variable slope delta modulation (CVSD) coding with 32 kbit/s and better than any of the presently used U.S. Government standards at 16 kbit/s or below, such that it is proposed to be adopted as U.S. Federal Standard FS-1016 for digital radio application [CaTW90], [WeTC89]. Not only that, the technique also provides an error protection scheme using stability rules, bit interleaving (scattering), and a forward error correction (FEC) technique using the Hamming (15,11) code [CaWT89]. We select CELP coding because it is an optimum technique. As shown later in Chapter II, CELP optimizes three aspects: (i) bit rate, (ii) speech quality, and (iii) computational complexity.

The 4.8 kbit/s bit rate leaves enough space for further robustness improvement. For example, a land mobile radio application enhances the CELP coding with a Golay code, an interleaver and a soft decision decoder, resulting in a highly improved performance at 8 kbit/s bit rate [RTWC89]. The system also performs well on certain fading channels. With such a low bit-rate, high-performance speech communication can still use narrow band channels [Jaya90].

The CELP compression also considerably reduces the storage required in the store-and-forward communication mode or voice mail applications. As shown in Table 1.1, a speech message of 1-minute length in the PCM form requires 480 kbytes storage space. Using the CELP technique, the same space can be used to store 13-minute equivalent speech messages.

With its high performance, CELP becomes an important techniques for speech coding, and is expected to be the future standard for speech coding at 4.8, 8, and 16 kbit/s rates [Wein91], [Jaya90]. However, the original form of CELP is impractical to implement in real time because it requires very intensive computation. In an early implementation, the

coding used 125 s of Cray-1 CPU time just to process 1-s of speech [ScAt85]. Efforts have been made since then to reduce the computational requirements, leading to modifications and fast algorithms. However, even in present form, CELP is still considered to be a computationally expensive technique, such that a single digital signal processor implementation is difficult [BrBH89]. In other words, studies are still needed to implement CELP to a low-cost platform. Although the proposed FS-1016 has defined the format and the meaning of each component of the compressed speech for interoperability, it makes no attempt in defining algorithms to obtain the compressed speech and how to implement the algorithms.

This thesis studies CELP coding for a low-cost implementation of a speech processing system, that is needed for real-time speech transmission of high-quality speech over packet-radio and for efficient speech storage. A detailed study of CELP coding is needed to enable an implementation. Furthermore, the study is needed to identify the difficult parts of the coding, and to obtain or modify efficient techniques to compute those parts. An efficient system architecture must be designed to implement the fast techniques such that a real-time implementation is possible for a given resource (speed and memory space) limitation of the platform. Optimization made for computation reduction and efficient design should not sacrifice speech quality. To ensure this, the thesis must also perform quality tests on the resulting system. The test results can also be used for diagnosing weaknesses for future improvements.

1.2 Problem Definition

Thesis Statement

The object of this thesis is to develop a speech processing system that can compress speech to a low bit rate (4.8 kbit/s) and reconstruct the speech with high quality for packet radio transmission and storage, using a digital signal processor system and CELP coding.

Problem Specifications

A system must be designed for high quality speech communication through packet radio at rates down to 4.8 kbit/s. The inputs of the system consist of (i) an analog speech signal input from an analog amplifier, and (ii) an RS232 port for data stream from a TNC. The outputs of the system are (i) an analog output to drive an analog amplifier, and (ii) an RS232 port for sending a data stream to the TNC. The voltage range of the analog signals is $\pm$ 5 volts. Furthermore, the RS232 channel operates at a rate of 4.8 kbit/s. However, for non real-time applications, the rates may vary according to the network physical layer and protocol.

The system should satisfy the following requirements:

- multi mode capability (real-time and non real-time, half-duplex and full-duplex),
- forward error protection,
- low-cost system, and
- compatibility with the proposed FS-1016 [CaTW90].

The multi-mode capability is needed for choosing either real-time or network-based interactions such as voice mail. The full- or half-duplex mode is needed because the available channel may only allow one of these modes. It should be noted that, for a real-time, full-duplex implementation, a digital signal processor is needed to perform the compression algorithms. If a non real-time application is sufficient, such as for network-based voice mail applications, the digital signal processor may not be necessary, thus reducing the system cost. However, analog-to-digital (A/D) and digital-to-analog (D/A) modules are still needed. The forward error protection, such as FEC, is used because retransmission is not possible for real-time applications. The system is implemented using a low-cost personal computer (PC) because the existing packet radio is already implemented using a PC connected to a TNC and a transceiver, thus difficult upgrading can be avoided. Finally, the FS-1016 compatibility is needed to be able to communicate with all the anticipated standard FS-1016 digital radio communication systems.

1.3 Thesis Organization

The organization of this thesis reflects the natural process used in this study. Chapter I provides the background and motivation of the thesis, the problem definition, and the thesis organization, that are necessary to introduce what is explored in the thesis.

Chapter II explains how the CELP technique compresses speech down to 4.8 kbit/s. It begins with a review of speech compression methods to put the CELP technique in a data compression perspective. A speech production model is then used to characterize the speech data. This chapter shows how the CELP optimally uses the characteristics to reduce the bit rate. Finally, it describes a special strategy in improving the robustness.

A CELP system has been designed and implemented on the PC-AT with a digital signal processor module. The system needs a fast algorithm, because the CELP technique has high computational-complexity. Chapter III discusses fast algorithms used to enable a fast implementation (possibly real-time) on such a platform.

Chapter IV describes the architecture, the organization, and the implementation of the CELP system. The system optimally distributes the processing tasks between the PC-AT and digital signal processor module. This chapter explains the hardware and software aspects of the system implementation, including their development process.

Results of a standardized listening-test confirm the high speech-quality of the system. Chapter V describes the experimental design, presents the results, and analyzes them to give the improvement possibilities.

Finally, Chapter VI presents conclusions, and highlights the achievements and the contributions that are made by this thesis. Recommendations are also given in this chapter.

References and Appendices, containing a document of the proposed FS-1016 standard, software listings of the system, and the listening test materials, can be found at the end of this thesis.

CHAPTER II
CODE-EXCITED LINEAR PREDICTIVE CODING

CELP coding optimally uses techniques from several speech compression methods to compress speech down to 4.8 kbit/s and still achieve natural quality of the reconstructed speech. The coding is also robust. This chapter first reviews several speech compression methods to identify problems in speech compression and to study techniques used to solve the problems. Some techniques require knowledge of the human speech production system, while others require tools from signal processing, functional analysis, or probability theory. The chapter then shows how the CELP technique selectively combines the techniques. After showing that the compressed speech indeed has a 4.8 kbit/s rate, this chapter shows how an error protection strategy increases the robustness.

2.1 Review of Speech Compression

2.1.1 Principles of Data Compression

Speech compression falls in a class of data compression. A data compression system consists of two functional blocks, a *compressor* and a *decompressor* [Kins91a]. A compressor maps original data, consisting of symbols, to other sets of symbols called *codewords*. The codewords can be either transmitted through a communication link or stored on a storage device. A decompressor maps the codewords back to data in the original form. In speech compression, the compression procedure is called *coding*, a compression system is called *coder*, a compressor is called a *speech analyzer*, a decompressor is called a *speech synthesizer*, and the codewords are called either *speech parameters* or simply *compressed speech* (see Fig. 2.1).

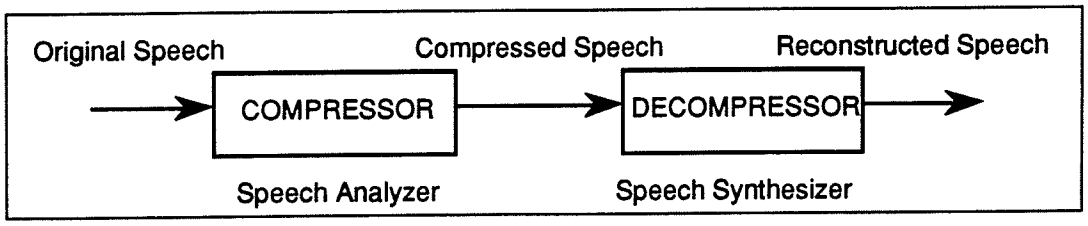


Fig. 2.1. A speech compression system.

The main feature of the system is that the codewords require less bit representations compared to that of the original data. The feature is measured by the *compression ratio* defined as

$$R_c = \frac{pk}{qn} \qquad (2.1).$$

Here, k is the number of bits per symbol in the original data, p is the number of symbols in the data, n is the number of bits in a codeword, and q is the number of codewords produced by the compressor [Kins91a]. For example, the CELP coding can compress speech from a rate of 64 kbit/s down to a rate of 4.8 kbit/s. In this case, R_c is over 13:1, or the file is compressed by 92.5%.

Data compression schemes work by identifying essential and redundant information in the original data, and encoding them differently. A compressor compactly encodes the redundancy using fewer bits. In certain methods, there is no bit required to represent the redundancy because it can be implied by the code structure. Both types of redundancy encoding are called *lossless* (or *reversible*) compression because they lead to perfect data-reconstruction. Another type is *lossy* (or *nonreversible*) compression which goes even further by eliminating the redundancy. Clearly, this lossy compression introduces some degradation on the reconstructed data. However, the essential information is still available and, more over, the compression ratio increases.

2.1.2 Difficulties in Speech Compression

In spite of the considerable effort in speech research for the last 50 years, speech compression is still difficult, mainly because of the non-stationary nature and variability of speech [Kins91a], [RaSc78], [Mari89], [Jaya90]. Methods from lossless compression may be used, but they cannot reach the required bit-rate. This is because they are not designed for the type of redundancy contained in speech. For example, a PKZIP compression based on Shannon-Fano and Lempel-Ziv-Welch techniques only results in a 47 kbit/s rate when applied to a PCM file (`ken1.pcm`).

Fortunately, speech samples can be categorized as imperfect data, e.g., minor losses in the speech data do not cause significant losses in the speech information. Therefore, research in speech compression concentrates on development of lossy type of compression methods. Indeed, current lossy methods have been able to achieve high compression ratios, such as 13:1, 26:1, or more [Wein91].

Criteria to determine the essential and redundant parts of speech play a critical role in lossy compression. Different criteria result in different performances of the compression methods. Better understanding of the data characteristics usually leads to a better criterion. We classify lossy compression methods based on criteria they use. We can then review the methods by observing the effects of the criterion on the compression features. To review the methods, we first define the features.

2.1.3 Features of Speech Compression

There are several features of speech compression we must consider for the review. Besides the compression ratio, which is the main feature, we still need features such as speech *intelligibility*, speech *quality*, and *implementability*, as defined below.

Speech Intelligibility

Speech intelligibility is the system ability to deliver correct speech-messages to

human listeners. It is measured based on the responses of a jury of human listeners [Pars86]. The jury must identify test words that are passed through a speech compressor and a speech decompressor, and the number of correct words is used for the intelligibility measurement. The techniques of measurement include rhyme test [Fair58], modified rhyme test (MRT), and diagnostic rhyme test (DRT) [JaNo84]. High speech intelligibility is desirable in normal, public speech communications, and especially essential in speech communications that either require higher word precision, or use very noisy channels.

Speech Quality

Speech quality is a subjective assessment of human listeners as to how close the sound of the reconstructed speech matches that of the original, pre-defined speech. IEEE has issued a guide on recommended practice for speech quality measurement. The recommendation includes relative preference, category-judgment (including mean opinion score (MOS)), and isopreference techniques [IEEE69]. The other techniques of measurement include binary decision tests and subjective signal-to-noise ratio (SNR) [JaNo84]. High speech quality is desirable in normal, public speech communications. It should be noted that one obstacle in using low bit rate speech coding for public communication is that the users are often annoyed by the machine-type speech sound produced by the coding.

Implementability

Implementability relates to resources needed to perform algorithms of the technique. The resources include computational power and memory space, in terms of million instructions per second (MIPS) and bytes, respectively.

Implementability is essential in real-time applications. Some techniques with excellent quality and high compression ratios may not be implementable by a given processor. In our case, a technique has an acceptable implementability if it can be

implemented using the given platform (a PC-AT and a single digital signal processor module).

2.1.4. Methods of Speech Compression

We now review and compare four classes of speech compression methods: (i) waveform, (ii) source (or parametric), (iii) filter-bank, and (iv) transform coding. It should be noted that despite this categorization, some speech compression schemes may use similar techniques in their process.

Waveform Coding

The criterion of waveform coding is the closeness (in a certain metric sense) of the reconstructed speech waveform to the original speech waveform. It relies on waveform manipulations such that the reconstructed waveform resembles the original waveform. The coding may also utilize a speech production model, but the main objective is to reduce the Euclidean error of the two time-domain waveforms [JaNo84].

Two ways of reducing the bit rate are (i) reducing the effect of quantization noise, and (ii) reducing information of the codewords [Pars86]. The reduction can continue until a predefined maximum Euclidean error is exceeded, thus limiting the compression rate the method can achieve. The former includes *dither* [JaRa72], A-law or μ-law compressor [Smit57], and adaptive quantization [Jaya73]. The latter includes differential PCM (DPCM) [Jaya74], adaptive DPCM (ADPCM) [JaNo84], adaptive predictive coding (APC) [Atal82], and multi-pulse linear predictive coding (MPLPC) [AtRe82].

The CELP coding, developed by Atal and Schroder in 1984 [ScAt85], [AtSc84], is often categorized as a waveform coding because it uses the same criterion [CoKK89]. However, since it heavily employs the speech production model, it is considered as a hybrid of waveform and source (parametric) coding techniques [MiAh87], [Atal86], [Jaya90].

In general, waveform coding is categorized as high-quality and relatively low computational complexity coding. However, the bit rates of the techniques are high (more than 8 kbit/s, except for CELP), and the low rates are achieved through heavy computational effort and sacrificing quality.

Source (Parametric) Coding

The criterion of source coding is the closeness (in a certain metric sense) of some parameters of the original and the reconstructed speech. Those parameters represent the essential information that control speech generation, according to a model of the human speech production system. Those parameters include *pitch* and *vocal tract* parameters (explained later in Section 2.3). During speech analysis, the coding extracts the parameters from the original speech. At the synthesizer, the received parameters control a speech production system developed according to the model [O'Sha87].

This coding can highly compress speech because the parameters require few representational bits. The reconstructed and the original speech may be different in the time domain, but they have the same parameters. With vector quantization techniques, this coding can even reach a rate of 0.4 kbit/s [LiRo89], [WoJC83].

Although the coding can maintain high intelligibility, it produces poor quality. The inability to extract perfect pitch and vocal tract parameters results in an incorrect synthesis [Flan72]. Furthermore, to increase the compression rate, the coding uses a simplified model that further reduces the speech quality and intelligibility.

The first electrical synthesizer, called formant synthesizer (1922), models the vocal tract according to its characteristic of frequency-transmission [Stew22]. It can produce a limited class of vowel sounds. A better formant synthesizer called Voder (Voice Operation Demonstrator) served as a foundation for formant synthesis research during the next three decades [FlRa73], [DuRW39]. Formant synthesizers can achieve a compression ratio of 50:1 [Lawr53]. A digital formant synthesizer, proposed by Rabiner from Bell Telephone

Laboratory in 1968, has been implemented in real-time implementation as reported in 1971 [Rabi68], [RJSC71]. However, formant synthesis coding suffers from difficulties of obtaining exact frequency characteristics (poles and zeros) of the vocal tract, affecting directly the resulting speech quality and intelligibility. In other words, a high-quality, real-time formant synthesis coding is difficult to implement.

The LPC technique mentioned in Chapter I overcomes this difficulty. In this technique, a linear filter approximates the vocal tract [FlRa73], [ItSa68], [ItSa70], [AtHa71], [Mark72]. There is no effort in finding the exact poles and zeros, thus a more accurate speech representation can be efficiently obtained. This technique has been widely used in many applications, such as in secure speech communication and speech recognition systems [Wein91]. An LPC version, called LPC-10e, has been adopted as the U.S. Government standard FS-1015 for speech communication at a rate of 2.4 kbit/s, with a compression ratio of more than 26:1 [WeTC89], [Swan87].

The compression ratio and the intelligibility of the LPC-10e are sufficient for our application. The filter approximation approach used in this technique reduces the computational complexity so that a real-time, full-duplex LPC system can be implemented using a single, moderate power DSP [Swan87]. However, the speech quality is rated as low to fair, mainly because of difficulties in finding correct excitation to the linear filter [Wein91], [AtRe82].

Other techniques, such as *phonetic* coding, and *synthetic-by-rule* coding can achieve very high compression ratios, but require very sophisticated analyzers [PiDo89], [Mari89]. A system employing synthetic-by-rule coding requires a continuous phoneme recognizer that is very difficult to implement. In addition, empirical rules are needed to combine phonemes during the reconstruction process. The higher the required output quality is, the more complicated the synthesis rules will be [Alle73], [Mari89].

Filter Bank Coding

The criterion of filter bank coding is the closeness (in a certain metric sense) of the frequency representation of original and synthetic speech. A bank of filters analyzes the frequency representation in several different ranges according to the filter ranges [O'Sha87]. Thus the codewords are the contents of frequency bands into which the frequency content (spectrum) of the speech is divided [Pars86], [JaNo84]. At the synthesizer, the speech is reconstructed by summing the outputs of a set of filters having the same frequency responses with those of the bank filters, with a proper common excitation. The coding compresses the speech by reducing the number of filters, thus reducing the total representation bits of the frequency content.

The earliest filter bank coding is the *channel* voice encoder/decoder (vocoder) [Dudl39], [Schr66], [GoBM81]. The channel vocoder has high intelligibility and is robust to background noise. However, the speech quality is poor. This vocoder has been used by the military for many years [Pars86], [Wein91].

Other filter bank vocoders are the *formant* vocoder [Schr66], [Flan72], the *phase* vocoder [FlGo66], [ScRa73], and the *subband* vocoder [Croc81], [EsGa77]. The two latter vocoders have potentially excellent quality. However, the phase vocoder needs rates of more than 16 kbit/s, while the subband vocoder needs rates of more than 9.6 kbit/s [JaNo84]. It should be noted that the subband vocoder has attracted new attention due to its relationship with a process that involves a new and efficient signal representation called the *wavelet* [VeHe90], [Mall89].

Transform Coding

The criterion of transform coding is similar to that of waveform coding. However the evaluation of the closeness is not limited in the time domain. Instead, the closeness can be measured in other domains determined by a transform pair. Here, speech waveforms can be represented by several contiguous segments of speech samples, and each segment can be represented by a linear combination of orthogonal vectors. The codewords are the

results of projecting a block of the speech samples on the orthogonal vectors, through a transform process.

Low bit rates can be achieved by reducing the number of the orthogonal vectors. However, since such a projection process is the same as quantization, reducing the number of the orthogonal vectors increases the quantization noise, thus reducing the speech quality.

Transform coding includes the discrete Karhunen-Loeve transform (KLT), discrete Fourier transform (DFT), Walsh-Hadamard transform, and discrete cosine transform (DCT) [JaNo84]. A new candidate is the discrete wavelet transform (DWT) [Mall89].

A typical problem of the transform coding is the high computational power needed in the transformation process. In addition, finding a good set of the orthogonal vectors for efficiently representing speech is difficult. In this case the DWT seems promising because of the inherent nonstationary approach embedded in wavelets.

2.1.5 Summary of Some Important Techniques

We now summarize the important charcteristics of the various techniques, especially those that are adopted later by CELP.

- Waveform coding achieves high quality because it uses time-domain matching of the waveform as a coding criterion.
- Source coding reaches low bit-rate because it uses parameters of a human speech production model as its essential information. In addition, LPC uses a linear filter to approximate the vocal tract efficiently.
- Filter bank coding uses frequency domain matching to increase the intelligibility.
- Transform coding divides speech to segments and encodes only one segment at a time to reduce complexity and computation. Here, a careful choice of the signal basis for speech reconstruction can reduce the bit rate while preserving the quality.

2.2 Basics of CELP Technique

CELP uses a code book (CB) of waveform prototypes, source parameters, and scalar quantization to achieve a low bit-rate, and it also uses closeness in the time domain waveform as its criterion (similar to waveform coding) to preserve quality. To increase the intelligibility, the metric (Euclidean) is perceptually weighted, as described later in Section 2.5.2.

The code book consists of prototypes of speech waveforms. Each prototype is located by a specific entry (address/index), as shown in Fig 2.2. The code book appears in both analyzer and synthesizer. In fact, the synthesizer is simply the code book, and each word in the compressed speech (codeword) is a particular entry.

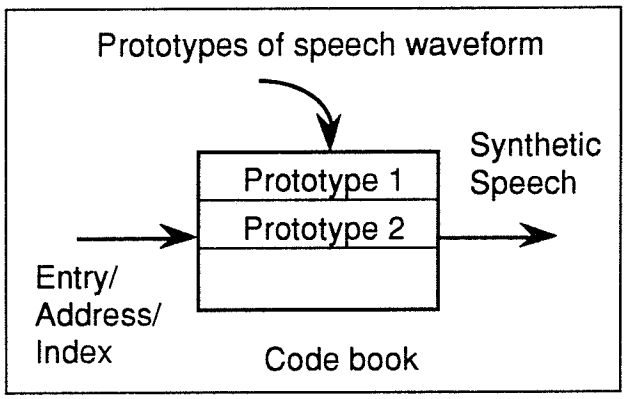


Fig. 2.2. A code book of prototypes of speech waveforms, used as a synthesizer.

To compress speech, an analyzer searches the code book for a prototype that matches the original speech (see Fig. 2.3). The scheme is called *analysis by synthesis*. The analyzer selects the prototype that has minimum error magnitude, and then sends the corresponding entry to the synthesizer. Since the synthesizer has the same code book, it

can completely reconstruct the best prototype. Thus, if the code book stores all possible speech waveforms, the synthetic speech has perfect quality. If the entry requires less representation bits than the representation bits of the original signal, we have a compression.

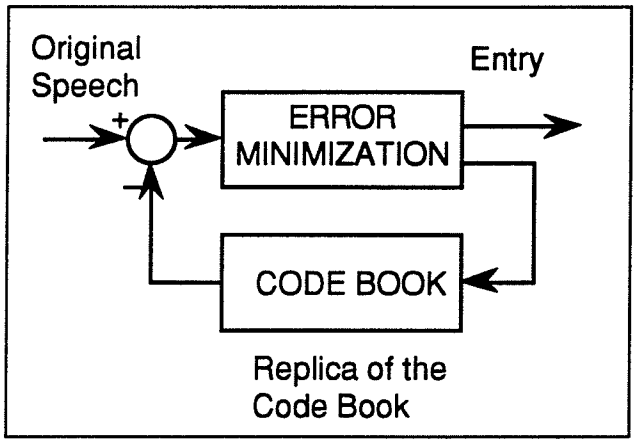


Fig. 2.3. A CELP analyzer using an analysis-by-synthesis scheme.

Although this type of coding provides high speech quality, it has a difficult problem. We recall that speech waveforms have high variability, which means there are many distinct speech waveforms that must be considered (see Section 2.1.2). To accommodate those waveforms, the size of the code book (i.e., the number of prototypes) becomes large. Consequently, we need more bits to represent the entry. If the size exceeds 2^N, where N is the number of bits of the original speech, the compression fails and instead we have an expansion. On the other hand, limiting the size increases the number of prototypes that must be eliminated, resulting in a larger synthesis error.

To overcome this difficulty, we must reduce the variability of the target waveform, that is the waveform that has to be matched by the prototypes. This reduction increases the redundancy of the target waveform such that a smaller number of prototypes is sufficient to correctly match the target, resulting in a smaller size of the code book. Indeed, the next basic concept of CELP after the code book is the variability reduction. Here, the CELP

technique turns to source coding and uses its techniques to extract pitch and vocal tract information from the original speech. The resulting waveform, called *residue*, has such a high redundancy that it can be treated simply as a noise signal. Now, the prototypes must match the residue. A small number of prototypes having similar statistics with the residue is adequate, resulting in a small size of the code book.

Since the synthesizer must apply the pitch and vocal tract information back to the speech waveform during a synthesis, the information must be transmitted together with the code book entry. Thus, the efficient size of the code book is not very useful if the extracted information still requires a high bit-rate. CELP addresses this problem through an efficient representation and quantization of the information.

To explain the variability reduction of the target waveform and efficient representation of pitch and vocal tract information, we need to understand the human speech production system, how it can be modeled, and how CELP utilizes the model. Furthermore, an efficient quantization needs understanding characteristics of the representation parameters.

2.3 Speech Production Model for Reducing Waveform Variability

From the acoustical point of view, speech production can be seen as a filtering process of air pulses. Thus, we can model the speech production system as a combination of filters and a pulse generator. The source parameters are then the coefficients of the filters and generator parameters.

2.3.1 A Human Speech Production System

The speech sound is generated by interactions of speech organs, as shown in Fig 2.4. The speech organs are grouped into three sections. The first section, called *pulmonary tract*, consists of *lungs* and *trachea*, that provide an air flow. The second section, called *larynx*, consists of *vocal cords* and their muscles. The space between the

vocal cords are called *glottis*. This section can convert the air flow into air pulses. The third section, called *vocal tract*, consists of a *pharynx*, an *oral cavity*, and a *nasal cavity*. The vocal tract acts as a resonance cavity for the incoming air pulses. The *velum* determines whether the nasal cavity is included in the production process or not. If the velum is open, which includes the nasal cavity in the process, the speech is *nasalized*. Otherwise, the speech is *unnasalized*. The output of the vocal tract is radiated by lips to become audible acoustic waveforms.

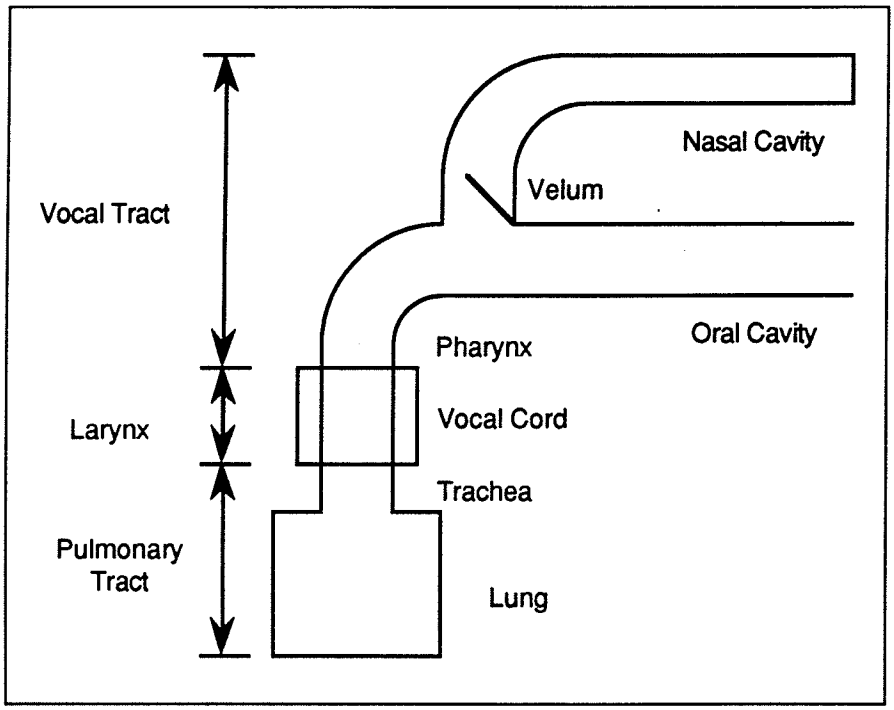


Fig. 2.4. A speech production model.

There are three important ways of generating the speech sounds, called speech *articulations*. They are either voiced, fricative, or stop articulations. *Voiced* sounds are produced as follows. The lungs create the air pressure, resulting in an air flow passing through the trachea and the glottis of the vocal cords. This air flow causes the vocal cords to vibrate, interrupting the air flow. The interrupted flow creates air pulses that are

quasiperiodic and broad in spectrum. The pulses excite the vocal tract, where they receive anti resonances from the nasal cavity and resonances from the other vocal tract organs. The sounds are then radiated at the end of the vocal tract.

Fricative sounds are produced as follows. A constriction is formed at some point in the tract. The air flow, that passes through the vocal cords (usually open and nonvibrated), is then forced through the constriction such that air turbulence can be created. This results in wide-band noise sounds.

Stop (plosive) sounds are generated as follows. A complete closure is formed at a place in the vocal tract, usually toward the front. The air flow, also passing through the vocal cords (usually open and nonvibrated), is forced to build up the pressure behind the closure. The closure is suddenly opened to produce an abrupt release. Fricative noise may follow after the release. Stop articulations also result in sounds with a noise-like spectrum.

Most of the fricative and stop sounds do not involve the vibration of the vocal tracts. Those fricatives and stops are usually called *unvoiced*, or *voiceless*, sounds.

In all articulations, the vocal tract provides resonances and anti resonances to the air pulses produced by each articulation. The glottis air pulses, air turbulence, and the abruptly released air, are called *excitations*. They are acoustical signals that are relatively broad in spectrum. The vocal tract acts as a *time-varying* acoustic filter that shapes the excitation spectrum according to the frequency characteristics of the vocal tract [Flan72].

The sound sources (excitation generators) and the vocal tract can then be linearly modeled as two separated functional blocks, as shown in Fig. 2.5. This separation gives a convenient way of individual examination of their acoustical properties. The examination results in a digital modelling of both blocks, including the parameters of each block.

2.3.2 Modelling of the Vocal Tract

A vocal tract gives the resonances and anti resonances to the incoming excitation pulses. The frequencies where the resonances and anti resonances take place correspond to the poles and zeros of the frequency-transmission characteristics of the vocal tract. The

frequency response of a vocal tract then has peaks and valleys because of the interaction of the poles and zeros. The frequencies where the peaks occur are called *formant* frequencies. Figure 2.6 shows a typical frequency response of a vocal tract.

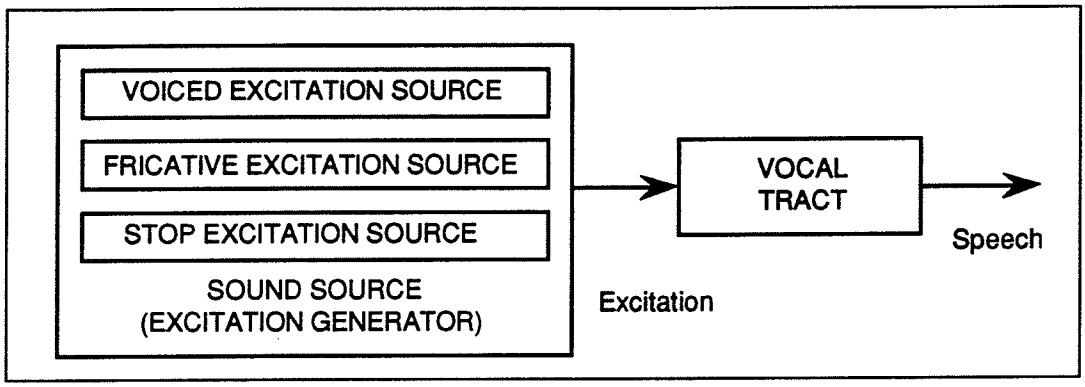


Fig. 2.5. A model of speech production system. The sound source produces three types of excitations: voiced, stop, and fricative excitations. The spectrum of the resulting excitation signal is shaped by the vocal tract to provide speech waveforms.

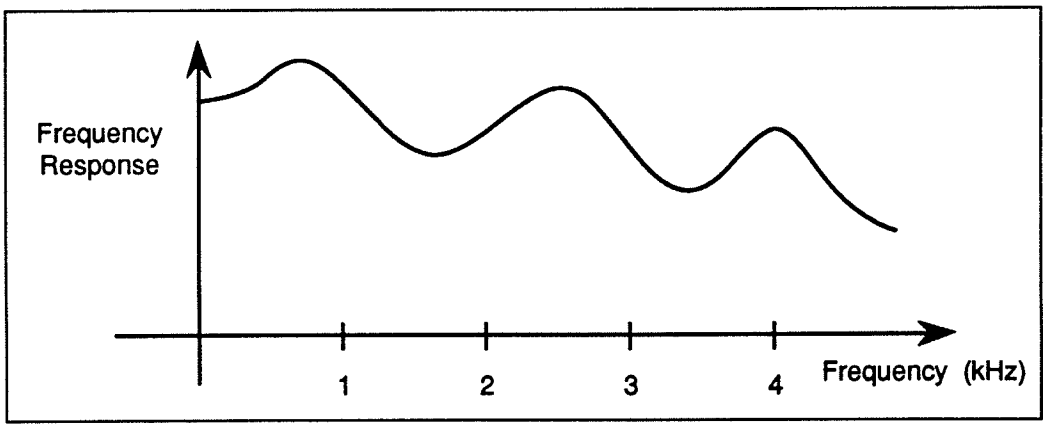


Fig. 2.6. Frequency response of a vocal tract. The frequencies where peaks occur are called formant frequencies.

As a filter, the vocal tract shapes the envelope of the spectrum of the excitation pulses. So, the spectrum of the speech has an envelope that shares the same peak and

valley frequencies with the frequency response of the vocal tract, as shown in the Fig. 2.7.

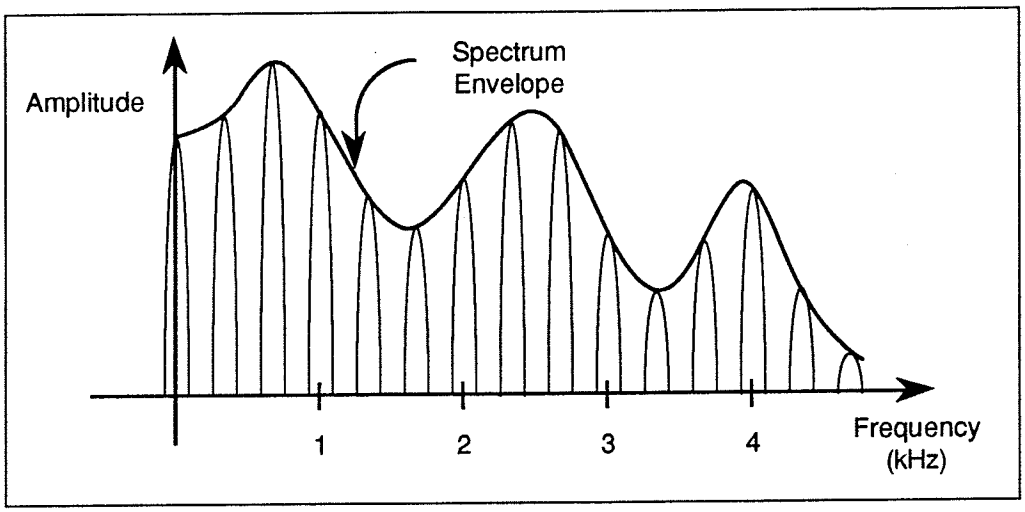


Fig. 2.7. Speech spectrum as a result of envelope shaping of the spectrum of the excitation pulses by the vocal tract. Notice that the spectrum and the frequency response of the vocal tract have the similar formant frequencies.

The frequency-transmission characteristics of a vocal tract can be approximated by a linear system with poles and zeros using acoustical analysis. The vocal tract can be further simplified, resulting in an all-pole approximation, explained as follows.

A special, nonuniform cylindrical tube models the vocal tract. Let us consider the tube with the closed velum. In practice, it is difficult to analyze this tube because the shape of the tube is not regular. Fortunately, a well developed acoustical analysis is available for a single *lossless, cylindrical* tube with a uniform shape. The vocal tract tube is then approximated by a *piecewise-cylindrical* tube. The pieces of such a tube are single, lossless, cylindrical tubes, that are cascaded to each other. An example of an approximated tube is shown in Fig. 2.8.

A transfer function of the vocal tract can be derived from a transfer matrix of a cylindrical tube [AtHa71]. The transfer matrix relates the air movement at the input side of the vocal tract with the air movement at the output side of the vocal tract, in term of volume

velocity. Using this approach, the transfer function of all the cascaded single sections is an all-pole function [Pars86]. The order of the transfer function is m, where m is the number of sections used for approximating the vocal tract.

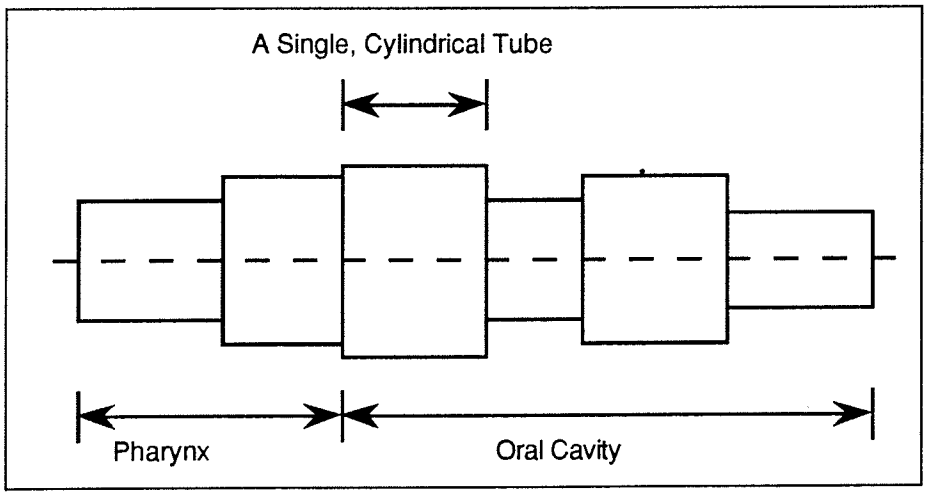


Fig. 2.8. Piecewise-cylindrical approximation of vocal tract with closed velum.

When the velum is open and nasal sounding takes place, a different model must be considered. We must add another acoustical element into our model, that represents anti resonances contributed by the nasal cavity. Because of this addition, a general pole-zero system must be used. An approach with one zero between 1 and 2 kHz gives good experimental results [Pars86].

However, we can eliminate the nasal articulations from modelling consideration. In many languages, including English, people still can completely understand unnasalized articulations. It should be noted also that a zero in a transfer function can be approximated by a large order, all-pole filter. So, to some extent, the nasal articulations are taken care of by the all-pole vocal tract filter.

Figure 2.9 shows an all-pole filter implemented in a direct form. This filter is called a *linear predictor* (LP) filter, representing the vocal tract. Let $H(z)$ be the transfer function of the LP filter. Then $H(z) = A^{-1}(z)$, where

$$A(z) = \sum_{i=0}^{m} a(i)\, z^{-i} \tag{2.2}$$

$A(z)$ is called *inverse* filter. The $a(i)$ are called LP filter coefficients, that vary over time according to the changes of the vocal tract shape. Here, $a(0) = 1$. A typical m is 10 because the speech quality drops significantly for any number below 10 [KrAt87].

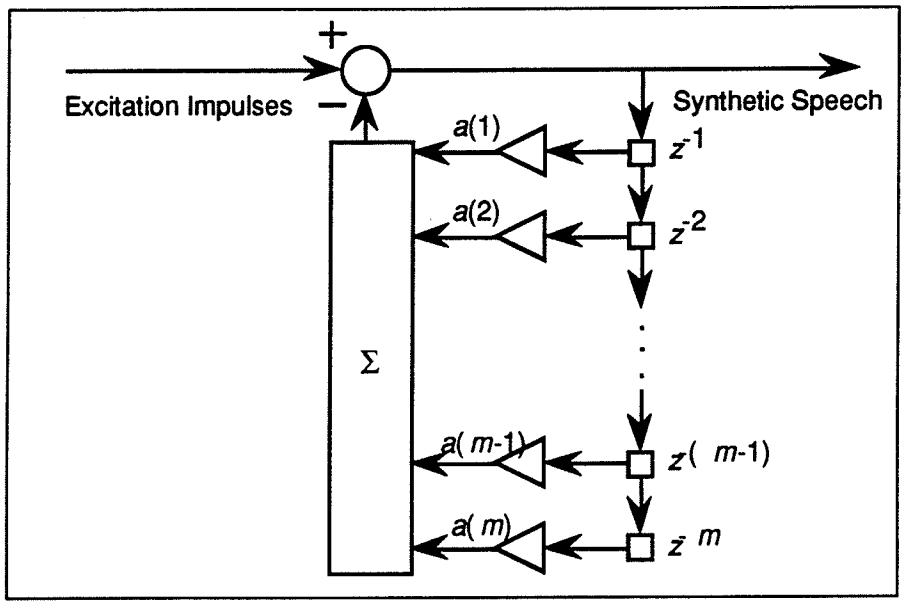


Fig. 2.9. A direct implementation of a vocal tract (LP) filter.

Although $H(z)$ varies over time, it can be considered unchanged for a short period of time, approximately between 10 to 50 ms, due to the limitation in the response time of the human vocal tract [O'Sha87]. This is an important observation that can be utilized in creating an efficient analysis. We can obtain a fixed set of coefficients for a period of time. Since the filter coefficients can completely define the filter, they become the parameters of the vocal tract.

2.3.3 Modelling of the Excitation Generator

Our model must accommodate the voiced, fricative, and stop sound sources.

Voiced articulations are characterized by vibrations of the vocal cords. The vibrations cause one important feature: the quasiperiodic pulses produced by the sound source have a relatively constant (regular) repetition time, defined as *pitch.* The rate of the repetition is called *pitch frequency.* The pulses also have a broad spectrum that has approximately 10 to 12 dB/octave rolloff above 0.8 to 1.0 kHz. The pulses and the spectrum are shown in Fig. 2.10. Similar to the vocal tract, the pitch can be considered stable for 10 to 50 ms due to limitation in the response time of the vocal cord.

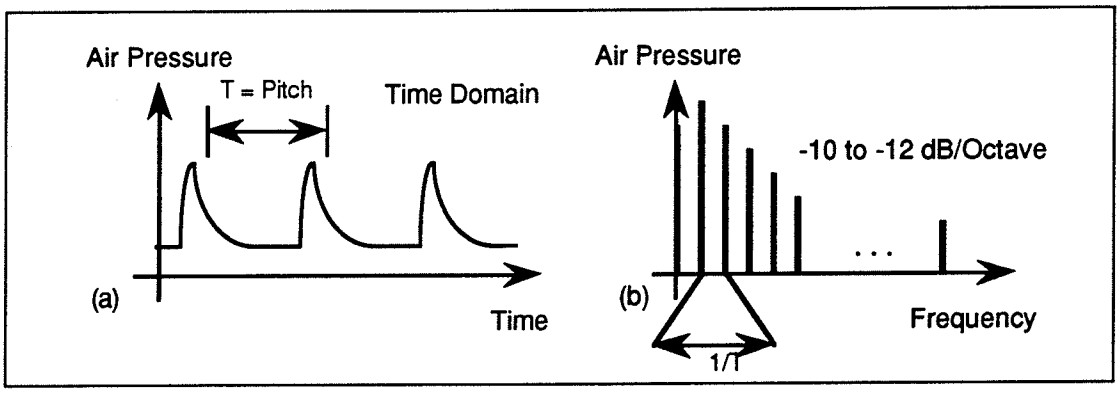


Fig. 2.10. A typical voiced excitation in (a) time domain, and (b) frequency domain. (After [Flan72]).

For modelling purposes, a digital pulse generator can be constructed to produce pulses with the same frequency characteristics. The generator consists of an impulse generator and a shaping filter. An impulse train having the same repetition rate with the pitch frequency is passed through the shaping filter, resulting in pulses that have the same spectrum with the actual pulses.

There is another factor that affects the voiced speech spectrum but not considered as a part of vocal tract, called *lips radiation* of speech. The radiation causes a 6-dB/octave gain. The radiation effect must then be included in the shaping filter, resulting in a shaping filter with a spectrum of 4- to 6-dB/octave rolloff. Since the shaping filter is relatively fixed, the parameter of voiced sounding is then the impulse repetition rate, or the pitch

information.

The unvoiced excitations give noise-like spectrum, as shown in Fig. 2.11. The sources of the noise can be inside the mouth (*alveolar/dental*), in the pharynx or velum (*pharyngeal/velar*), or at the lips (*labial*), resulting in a different spectrum. However, it has been found experimentally that we still can well recognize synthetic stops or fricatives without preserving exactly the same spectrum of the original stops or fricatives. In other words, stops and fricatives can be generated using one noise source with a fixed spectrum.

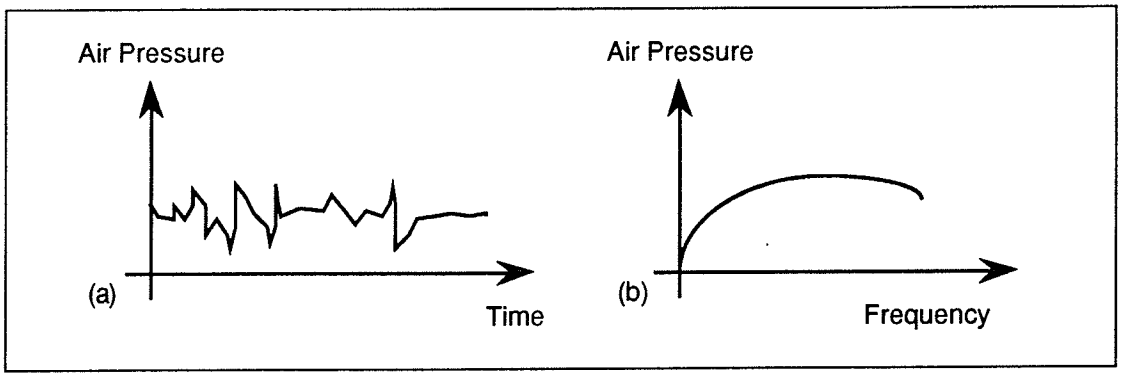


Fig. 2.11. A typical unvoiced excitation in (a) time domain, and (b) frequency domain. (After [Flan72]).

The above results lead to an idea that at least there is another feature needed to distinguish the different stops and fricatives. *Formant transitions* have been found to become a distinct feature of stops and fricatives [Pars86]. Ideally, the formant trajectories of a voiced sound should be flat with respect to time, since the vocal tract shape is stable. But in practice, this flatness only occurs in cases such as in isolated vowel articulations. If the vowel is beside a stop or a fricative, the formant trajectories become curves with respect to time. The curves are called formant transitions. Different stops and fricatives result in different formant transitions. Figure 2.12 shows an example of formant transitions of three different articulations /ba/, /da/, /ga/.

The above results give an important advantage in our model development. The essential information of the unvoiced sound source is provided by the vocal tract filter

coefficients. The perceptual distinctions of stops or fricatives rely on the vocal tract estimation of the next, or in between, unvoiced articulations. Consequently, precision in noise statistics to excite stops and fricatives is not essential, and we can concentrate on precision of voiced articulations.

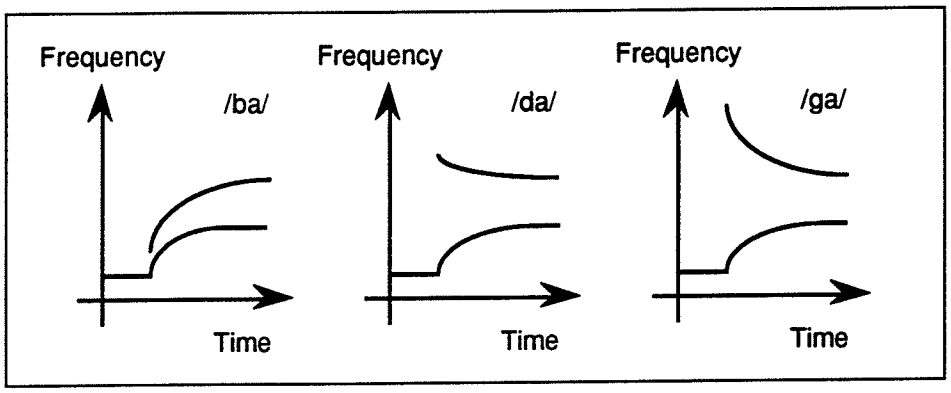


Fig. 2.12. Trajectories of the first and the second formants of three different articulations. Notice that the second formant can be used to distinguish the three unvoiced articulations /b/, /d/, and /g/. (After [Pars86]).

2.3.4 Combining the Models of Vocal Tract and the Excitation Source

Now, we combine the vocal tract and excitation pulse generator into a speech production model, as shown in Fig. 2.13. The shaping filter has been included in the LP filter. The pitch information is applied by a new filter called *pitch filter*. To compensate for the changes, special impulses, called *residual impulses*, or simply *residues*, are introduced to provide the excitations for all articulations. Since the LP filter has been described in Section 2.3.2, it remains to describe the pitch filter and the residue.

Pitch Filter

This pitch filter must impose periodicity according to the pitch period, especially during voiced articulations. A filter can perform this function by enhancing the frequency components of the residues corresponding to the desired periodicity. The filter must be

adaptive because the pitch is time varying. A one-tap, all-pole adaptive filter is the simplest implementation. Although filters with more than one tap can be used, they do not provide significant improvements [KrAt87].

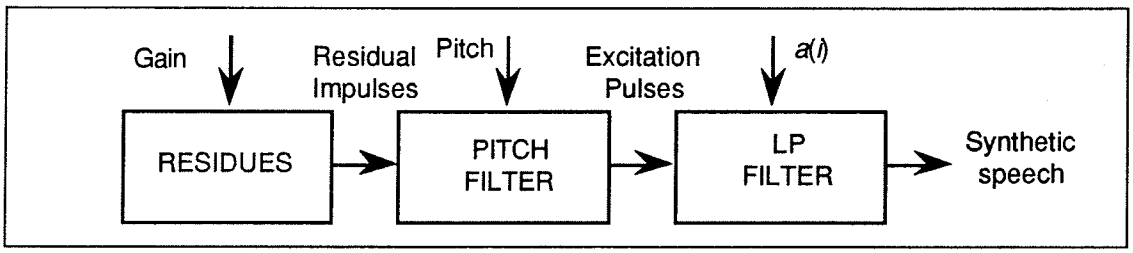


Fig. 2.13. A speech production model in a digital form.

As shown in Fig. 2.14, the parameters of this one-tap pitch filter are a gain factor g and a tap position d. This filter satisfies

$$y(n) = g\, y(n-d) + x(n) \tag{2.3}$$

where $y(n)$ and $x(n)$ are the n^{th} output and input of the filter, respectively. A filter with such a structure always enhances the pitch frequency with periodicity determined by d. The y(n) is the same with the y($n - d$) (scaled by g) plus $x(n)$.
Thus, the periodicity the filter imposes is

$$\text{Pitch} = \frac{d}{\text{Sampling Frequency}} \text{(s)}$$

Consequently, the pitch frequency imposed by the filter is

$$\text{Pitch frequency} = \frac{\text{Sampling Frequency}}{d} \text{(Hz)} \tag{2.4}$$

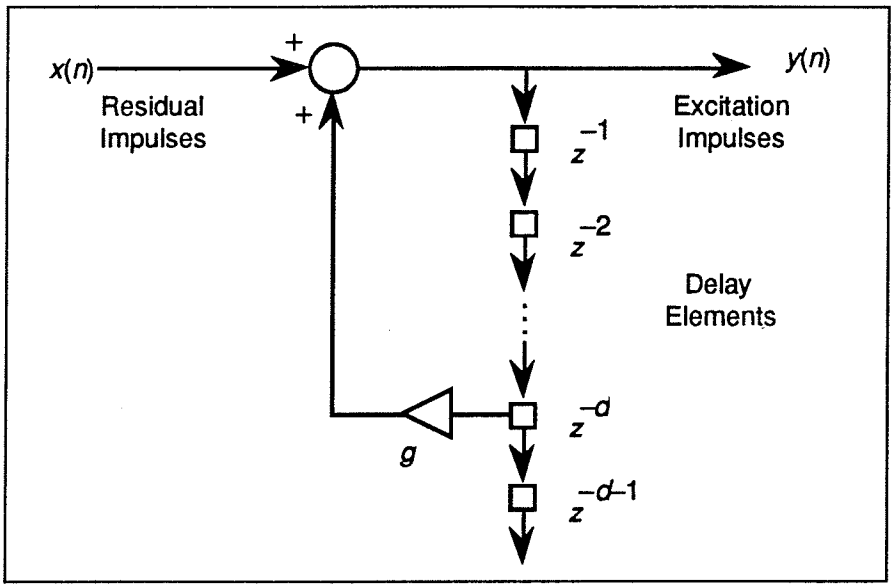


Fig. 2.14. A one-tap, all-pole pitch filter.

As mentioned in Section 2.3.3, the pitch can be considered stable for a period of time less than 50 ms due to the limited response time of the vocal cord. That implies the values of d and g must be updated at least every 50 ms. More frequent updating results in a better speech quality.

Residues (Residual Impulses)

The residues function as excitation impulses for the cascade of the pitch and LP filters. We define residues as the results of applying original speech as input to a cascade of two inverse filters, the inverse of the LP filter and the inverse of the pitch filter. In other words, the residues are the speech samples without vocal tract and pitch information (see Fig. 2.15). Observation shows that the residues have noise-like property for both voiced and unvoiced articulations [Atal82]. Furthermore, for voiced articulations, the residual impulses have a zero-mean, Gaussian distribution.

Fig. 2.15. Waveforms of different signals. (a) Original speech. (b) Original speech after an LP inverse filter, representing the excitation impulses. (c) Residual impulses as a result of applying the original speech to a cascade of the inverses of LP and pitch filters. The amplitudes of (b) and (c) are magnified by approximately 5 and 2 relative to (a), respectively. (After [ScAt85]).

Thus, a noise generator can be used for both voiced and unvoiced articulations. Furthermore, since the noise precision of unvoiced articulation is not critical, we can use a zero-mean, Gaussian noise generator for both articulations.

The CELP uses the above model to reduce the variability of the code book target waveform. The structure described in Section 2.2 is now modified to accommodate the change in target waveform from speech waveforms to a zero-mean, Gaussian-distributed noise signal, as shown next.

2.4 CELP Speech Synthesis

A CELP synthesizer functions as a waveform generator whose output is produced

by applying a prototype from a code book to a cascade of pitch and vocal tract filters, as shown in Fig. 2.16. The synthesizer is similar to the model in Fig 2.13, except a small size code book replaces the residual pulse generator. The code book prototypes have the same statistics as the residue, i.e., zero mean and a Gaussian distribution. Thus, the size of the code book can be much reduced.

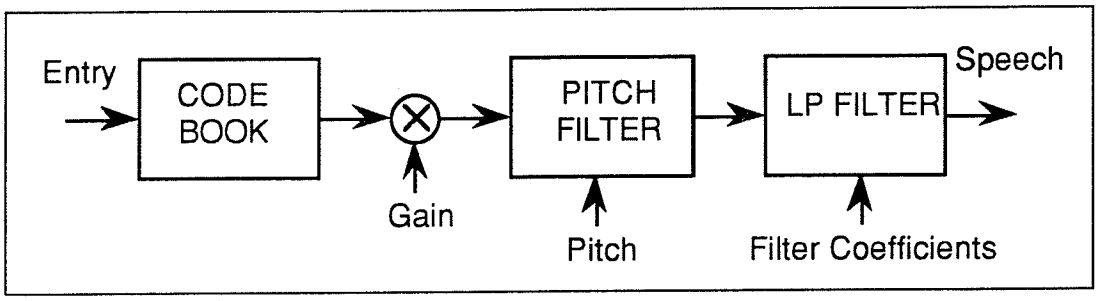


Fig 2.16. A CELP synthesizer.

Segmentation similar to the transform coding (see Section 2.1) further reduces the variability of the target waveform, resulting in a smaller code book size. To illustrate this reduction, consider two identical segments of residual impulses with variability N each. If the two segments must be considered as one larger segment, the combined variability becomes N^2. However, if we consider them separately, the total variability is only $2N$. Thus, segmentation reduces the variability from geometric to algebraic type of complexity.

Normalization of the target waveform also reduces the variability. Here, instead of having a code book with NM variability, we have prototype and gain code books with only N and M variability, respectively. As a result, the target waveform variability effectively reduces to N.

With the reductions, we have a CELP synthesizer used by many CELP coders. Those coders usually vary only in code book design and the pitch filter. To communicate with other FS-1016 compatible systems, we use a code book and a pitch filter that are compatible with the standard. The following is discussion on the code book design and an

FS-1016 pitch filter.

Discussions on Code Book Design

There are two ways in generating prototypes: (i) *deterministic*, and (ii) *random* ways. The deterministic way applies speech into a cascade of the inverses of the LP and pitch filters, and chooses sequences of residual impulses that most frequently occur to be the prototypes.

The other way, called random way, utilizes the fact that residual impulses are noise-like, and have a zero-mean, Gaussian distribution for voiced articulations. A Gaussian random generator creates the code book elements. If the impulses stored inside the code book satisfy the distribution, the code book can be expected to provide proper residual impulses. Indeed, CELP systems with such a code book have been reported to have high quality of speech [KlKK90]. A code book created in this way is called *stochastic code book* (SCB).

Creating a deterministic code book is more difficult because we have to provide speech utterances as a training set, and we have to choose most frequent sequences or their representatives [KlKK90]. Choosing the sequences is difficult because it is more likely that a sequence occurs only once. However, a designer has more freedom to apply a structure to a deterministic code book by creating the representatives according to a certain rule. The structure can be utilized to have a fast analysis algorithm, which is important in real-time applications.

As a compromise, a structure can be imposed on an SCB. This results in a high quality CELP, yet the code book is still simple, thus fast analysis is possible. All the details of implementing this type of SCB and its effects on the analysis are described in Chapter III when we describe fast CELP algorithms.

Adaptive Code Book as a Pitch Filter

The one-tap pitch filter shown in Fig 2.14 can be implemented using an adaptive

code book (ACB) that is similar to the SCB, as shown in Fig. 2.17. The ACB also has its own entry and gain. The difference is merely in the prototypes of the code books. The advantage of implementing pitch filter in this ACB form is that its analysis can share tools with fast SCB analysis.

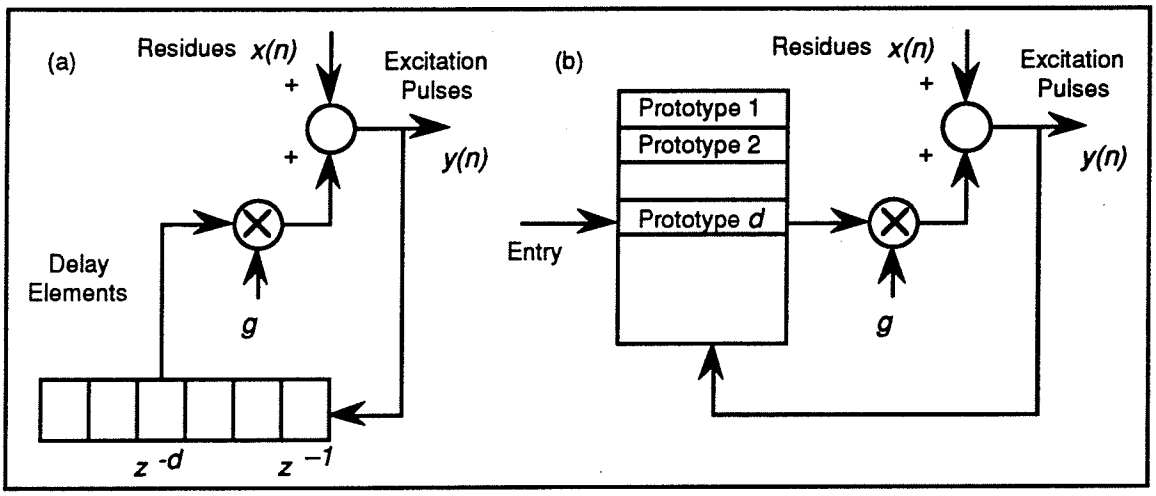


Fig. 2.17. (a) A pitch filter, and (b) its equivalence using a code book form. The code book imitates the function of the delay elements. The code book prototypes are then overlapping. Since the elements of the prototypes must be updated, the code book is called adaptive code book (ACB).

According to Fig. 2.14, the pitch filter has several delay elements, that is updated by the output of the filter. By redrawing the pitch filter to an equivalent filter in Fig. 2.17a, we can construct a code book that imitates the delay element function. Here, the i^{th} prototype of the code book in Fig. 2.17b consists of all impulses that must be supplied by delay elements when the i^{th} tap is selected. Consequently, the prototypes overlap each other. In addition, the prototypes must be updated by new excitation impulses. Because of this, the code book is called an adaptive code book.

Using this pitch filter implementation, the CELP synthesizer is redrawn to become a scheme shown in Fig. 2.18. This scheme is a practical CELP synthesizer that shall be used

throughout this thesis.

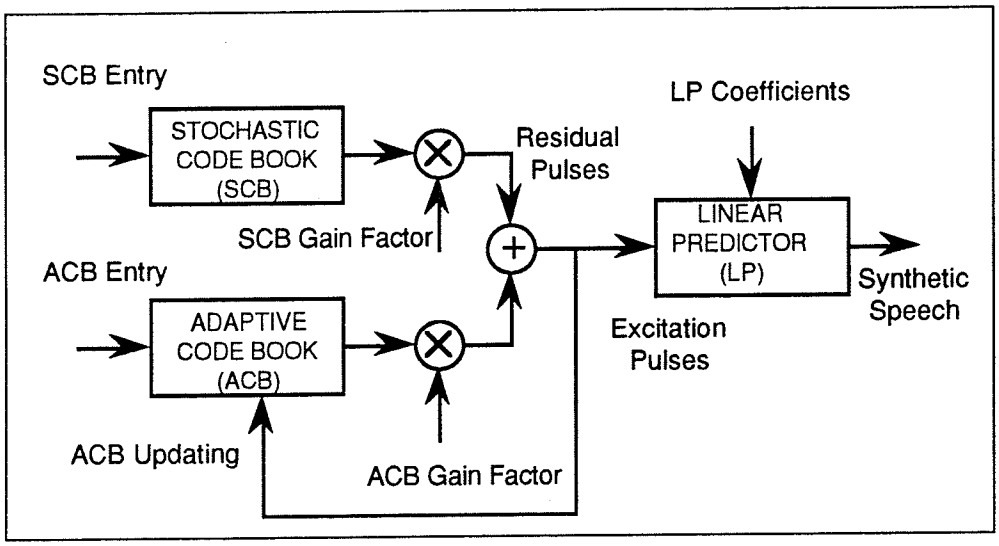


Fig. 2.18. A CELP synthesizer. The pitch filter is implemented using an ACB.

Summary of CELP Parameters

We can now summarize the CELP parameters. The CELP parameters consist of ACB, SCB, and LP parameters.

- The ACB parameters are an ACB entry (address/index) and an ACB gain.
- The SCB parameters are an SCB entry (address/index) and an SCB gain.
- The LP parameters are the 10 LP filter coefficients $a(i)$.

These parameters are compatible with the FS-1016 standard, and they control a CELP synthesizer. Thus, a CELP analyzer must extract those parameters.

2.5 CELP Speech Analysis

A CELP analysis maps a block of speech samples into a set of CELP parameters. Since we have introduced LP parameters, the scheme in Fig 2.3 must include an LP analysis. Furthermore, since the pitch filter has become a code book, we can employ the same code book search to find its parameters. Thus, there are two processes, namely, LP filter analysis, and code book parameter searching, as shown in Fig. 2.19. The LP filter

analysis, which is an open-loop process, is used to obtain the LP filter coefficients (LP parameters), while the code book parameter search, which is a closed-loop process, is used to obtain the ACB and SCB gain factors and entries.

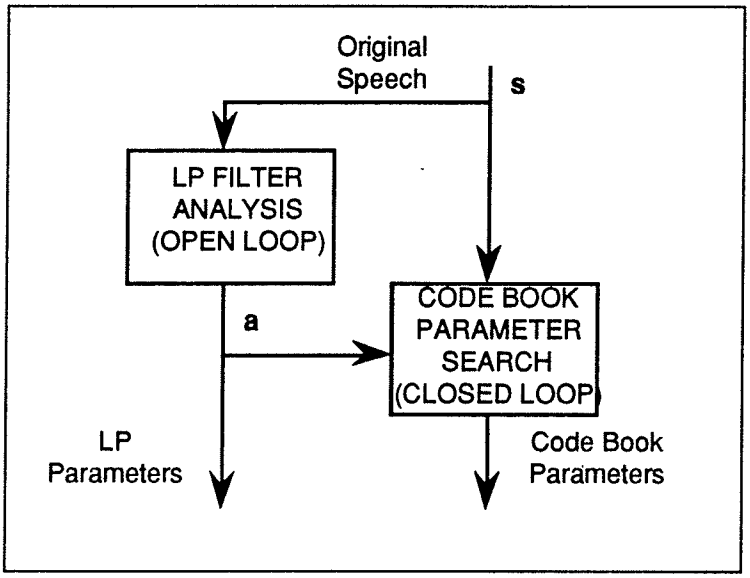


Fig. 2.19. CELP analysis.

The original speech samples $s(n)$, denoted by a vector **s**, are first applied to the LP filter analysis process to estimate 10 LP parameters $a(i)$ mentioned in Eq. (2.2), denoted by a vector **a**. The code book parameter searching then uses **a** and **s** to produce the code book (ACB and SCB) parameters corresponding to **s**.

2.5.1 Estimation of the LP Parameters

Given a frame of original speech samples **s**, an LP analysis estimates the LP filter coefficients **a** in a mean square error sense. In this sense, we can utilize the *orthogonal projection* principle to estimate **a** [Krey78]. Let the output of the LP filter be **s'**. The orthogonal projection principle then guarantees that there exists the best and unique **a** that minimizes $\|\mathbf{s}-\mathbf{s}'\|$.

Problem Setup

Given m previous speech samples $s(n)$, $-\infty < n < \infty$, the LP filter uses Eq. (2.2) to produce the estimation sample $s'(n)$ according to

$$s'(n) = -\sum_{i=1}^{m} a(i)\, s(n-i) \tag{2.5}$$

Here, we assume that the input of the filter is a zero-mean and stationary process, thus it can be omitted during this estimation [Proa83]. Since the actual output is $s(n)$, the filter output error $e(n)$ is

$$e(n) = s(n) - s'(n)$$

which means

$$e(n) = s(n) + \sum_{i=1}^{m} a(i)\, s(n-i) = \sum_{i=0}^{m} a(i)\, s(n-i), \qquad a(0) \equiv 1 \tag{2.6}$$

The problem becomes finding **a** that minimizes Eq. (2.6) in a mean square sense. We have the advantage from section 2.3 of assuming **a** is stable for a period of time. Let the number of samples during this period be N. The desired **a** is then an **a** that minimizes Eq. (2.6) for all N samples.

Let a sequence of speech samples having the first element $s(n-N+1)$ and the last element $s(n)$ be an N-element vector $\mathbf{s}_n$. With this construction, $\mathbf{s}_n$ is independent of $\mathbf{s}_l$, $n \neq l$. The same construction is used for the $e(n)$ to create a vector $\mathbf{e}_n$. The Eq. (2.6) can then be extended in vector notation as

$$\mathbf{e}_n = \sum_{i=0}^{m} a(i)\, \mathbf{s}_{n-i}, \qquad a(0) = 1 \tag{2.7}$$

Let us now construct the estimation space on which the **a** exists. The space is a m dimensional space which is spanned by $\{\mathbf{s}_{n-i}\}$, $i = 1, ..., m$. According to the orthogonal projection principle, $\|\mathbf{e}_n\|$ is minimized if and only if $\mathbf{e}_n$ is orthogonal to all of the $\{\mathbf{s}_{n-i}\}$. Furthermore, there is only one **a** that makes $\mathbf{e}_n$ orthogonal to the $\{\mathbf{s}_{n-i}\}$. We are interested in obtaining such **a**, thus obtaining the best estimation of the LP filter coefficients.

Solution Using LPC Analysis

The following technique of obtaining such an **a** is known as LPC analysis [Proa83], [Pars86]. An inner product, denoted by $\langle \bullet, \bullet \rangle$, can be defined in the space containing $\mathbf{s}_n$, $\mathbf{e}_n$, and the estimation space because the space is also an N-dimensional real space (R^N). In an inner product notation, $\mathbf{e}_n$ is orthogonal to the $\mathbf{s}_{n-i}$ if and only if

$$\left\langle \mathbf{s}_{n-j} , \sum_{i=0}^{m} a(i)\, \mathbf{s}_{n-i} \right\rangle = 0, \qquad j = 1, ..., m. \tag{2.8}$$

By inner-product properties and assuming $a(i)$ is real, Eq. (2.8) is equivalent to

$$\sum_{i=0}^{m} a(i) \langle \mathbf{s}_{n-j} , \mathbf{s}_{n-i} \rangle = 0, \qquad j = 1, ..., m \tag{2.9}$$

To obtain the values of the inner-product terms in Eq. (2.9), we can utilize one of two methods, the *covariance* or the *autocorrelation* methods. Although both result in almost the same speech quality, the latter method requires less computation and shall be used.

In the covariance method, the inner product is calculated directly using the definition of an inner product over N samples, where N is finite. Using this method, the magnitude of $\mathbf{e}_n$ is minimized within a block of N speech samples.

The autocorrelation method modifies the way we look at $\mathbf{s}_n$. Here, $\mathbf{s}_n$ is seen as a vector with infinite elements but only N of them are nonzero. The inner product must then be calculated over infinite samples, providing $s(n) = 0$, for $n < 0$ and $n \geq N$. The advantage of this modification is that the inner product terms in Eq. (2.9) become

$$\begin{aligned} \langle \mathbf{s}_{n-j} , \mathbf{s}_{n-i} \rangle &= \sum_{n=-\infty}^{\infty} s(n-j)\, s(n-i) = \sum_{n=-\infty}^{\infty} s(n)\, s(n+j-i) \\ &= \sum_{n=i}^{N-1} s(n)\, s(n+j-i), \quad i = 0, \cdots, m; \; m < N \end{aligned} \tag{2.10}$$

Let us define r_i as

$$r_i = \sum_{n=i}^{N-1} s(n)\, s(n-i) \tag{2.11}$$

The linear system Eq. (2.9) can then be written as

$$\sum_{i=0}^{m} a(i)\, r_{i-j} = 0, \qquad j = 1, ..., m, \quad a(0) = 1 \tag{2.12}$$

Since $s(n)$ is known, r_i can be computed using Eq. (2.11). Eq. (2.12) then becomes a system of m linear equations with m unknowns, $a(i)$, which can be solved using any method, such as Gauss-Jordan elimination. Thus, Eq. (2.12) gives the desired **a**.

Notice that Eq. (2.11) is similar to an autocorrelation process of $s(n)$. (The method is called an autocorrelation method because of this fact). The matrix of r_{i-j} in the above equations is called the autocorrelation matrix. The matrix is Toeplitz and symmetric. This permits fast and efficient techniques of solving the system of equations, such as Levinson-Durbin, LU Cholesky decomposition, Burg, and Schur techniques [Pars86]. It

should be noted that if we use the covariance method, we still have a system of linear equations similar to Eq. (2.12). However, the corresponding matrix is not Toeplitz, thus solving the system of equations takes longer time.

Estimation Enhancement using Hamming Window

When we use the autocorrelation method, notice that the method assumes $\mathbf{s}_n$ to have infinite elements. However, since we only have N speech samples, we introduce distortion by providing a truncated $\mathbf{s}_n$. The distortion is similar to the Gibbs phenomenon in a short time digital Fourier transform [DeLH88]. To reduce the distortion, we must avoid such an abrupt truncation by introducing smoothing window. The window is a sequence of M data points satisfying a certain smoothing shape. The $\mathbf{s}_n$ is said to be windowed if it is element-wise multiplied by the window sequence. Speech processing commonly uses the Hamming window, whose elements are defined as [DeLH88]:

$$w(n) = 0.54 - 0.46 \cos(2\pi n/\mathrm{N}), \quad 0 \le n \le \mathrm{M} - 1 \tag{2.13}$$

Thus, before Eq. (2.11) is applied, the windowed $s(n)$ is obtained by multiplying the original $s(n)$ with $w(n)$. The proposed FS-1016 recommends the use of a 30 ms non overlapped Hamming window spanning from the middle of the current frame to the middle of the next frame, as shown in Fig. 2.20. That means M = N, and the windowed $s(n)$ is obtained by multiplying $s(n+\mathrm{N}/2)$ with $w(n)$. In other words, the LP analysis does not work strictly on speech samples from the current frame, but it uses half of the speech samples from the current frame and half of the speech samples from the next frame. This scheme produces parameters corresponding to a smoother synthetic speech.

2.5.2 Estimation of Code Book Parameters

The estimation of ACB and SCB parameters, e.g. gain factors and entries, is done for each block of $N/4$ speech data, called a subframe. Thus, for every **a**, there are four

sets of ACB and SCB parameters. Each set of parameters is obtained using an analysis-by-synthesis scheme.

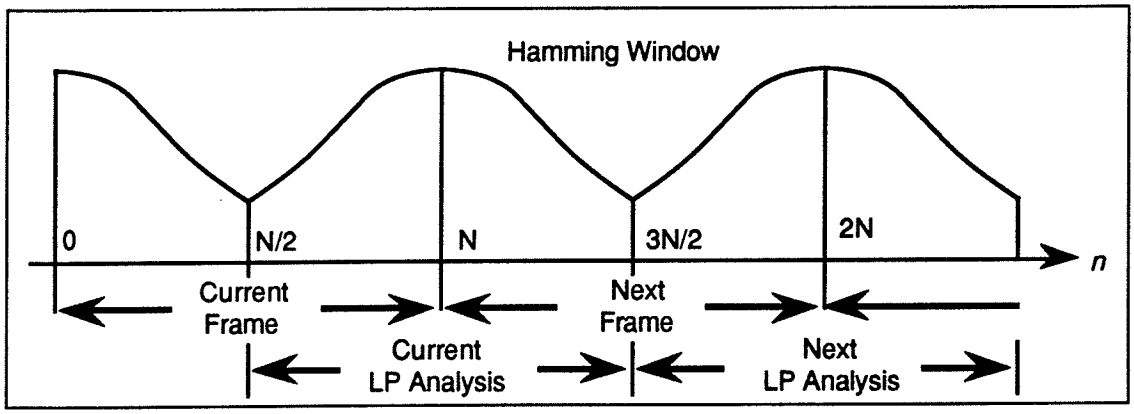


Fig. 2.20. Windowing of speech data prior to autocorrelation process. The speech samples used for current LP analysis is the samples from the third and fourth of current subframes and the first and second of the next subframes.

Analysis by Synthesis

Figure 2.21 shows the process of obtaining a set of code book parameters. A CELP synthesizer is set up and its LP filter uses the **a** resulting from the LPC analysis. The error minimization block has a list of all possible sets of code book parameters. The block starts applying the first set to the CELP synthesizer. The synthesizer then produces $N/4$ samples of synthetic speech called **s'**. The synthetic speech is compared with the original speech **s**. The difference is applied to a perceptual weighting filter to become a weighted error vector **e** according to

$$\mathbf{e} = P(\mathbf{s} - \mathbf{s}') \tag{2.15}$$

Here, P represents the perceptual weighting filter. The $\|\mathbf{e}\|$ is measured using the Euclidean norm and stored by the error minimizing block. The block then applies the next

set of parameters until all possible sets have been tried. The set that minimizes ||e|| is chosen as the result of the searching process.

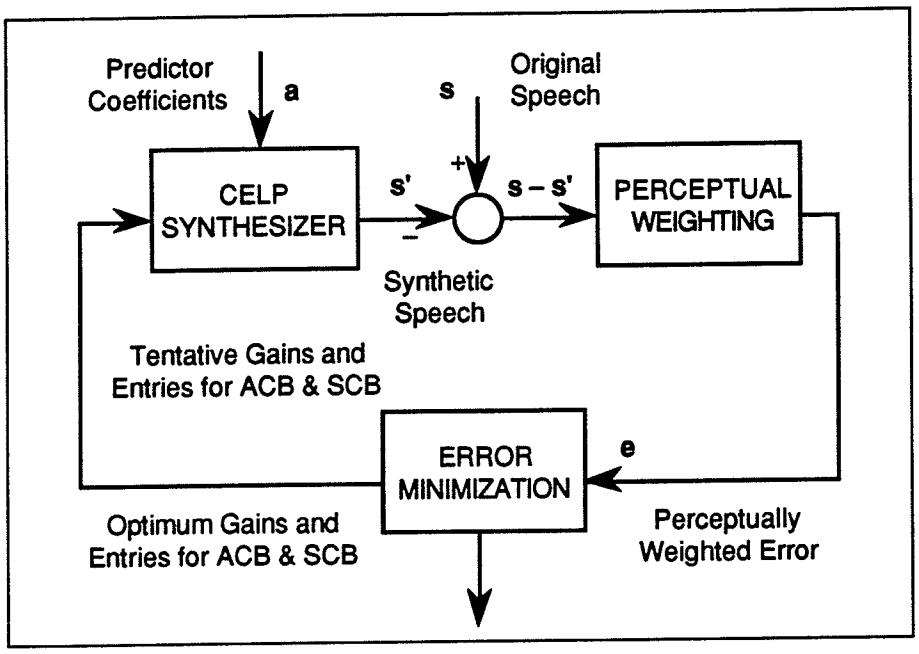


Fig. 2.21. Basic scheme of code book parameter searching.

Perceptual Weighting Filter to Improve Distance Measure

The perceptual weighting filter is a digital filter that weights the error signal such that a Euclidean measure of the weighted error becomes more perceptually meaningful. When the analyzer compares the synthetic speech with the original speech during the code book parameter searching, it makes a time-domain matching process in a mean square error sense. However, the same intelligibility does not necessarily mean the same speech waveform. (Although the inverse is true, i.e., the same speech waveform does result in the same intelligibility). Thus, a direct mean square error measure may fail to find the perceptually best match.

A better measure for speech intelligibility is a frequency weighted error measure recommended by [AtRe82], and defined as

$$\|e\| = \left(\int_0^{f_s} |S(f) - S'(f)|^2 P(f)\, df \right)^{\frac{1}{2}} \tag{2.16}$$

Here, $S(f)$ and $S'(f)$ are the Fourier transform of **s** and **s'**, respectively. The f_s is the sampling frequency. Without the positive weighting function $P(f)$, Eq. (2.16) is just another mean square measure in the frequency domain. However, a suitable $P(f)$ makes the measure perceptually meaningful.

To obtain $P(f)$, let us consider again the spectrum of the speech shown in Fig 2.6. In the formant regions, the energy of the speech is high, while in the valley frequencies (in-between formant areas), the energy is low. So, a small error in the valley frequencies gives a higher degradation in signal-to-noise (SNR) than that in the formant regions. Consequently, the effect of an error in the valley frequencies is perceptually worse than that in the formant areas.

The $P(f)$ is constructed to accommodate this formant frequency dependent characteristic to a measurement criteria. The $P(f)$ must penalize error in the valley frequencies more than error in the formant areas. Consequently, the frequency response of the $P(f)$ must be proportional to the inverse of the vocal tract transfer function which introduces the peak and valley regions to the envelope of the speech spectrum. Using the inverse filter $A(z)$ defined in Eq. (2.2), [AtRe82] shows that such a weighting filter has a transfer function $P(z)$ as follows

$$P(z) = \frac{A(z)}{A(\frac{z}{\sigma})}, \qquad 0 \le \sigma \le 1. \tag{2.17}$$

The factor σ determines the degree of error emphasis/deemphasis in the regions. The σ is determined experimentally, and a typical σ is 0.8. Figure 2.22 shows a frequency response of the weighting filter, for a given vocal tract filter.

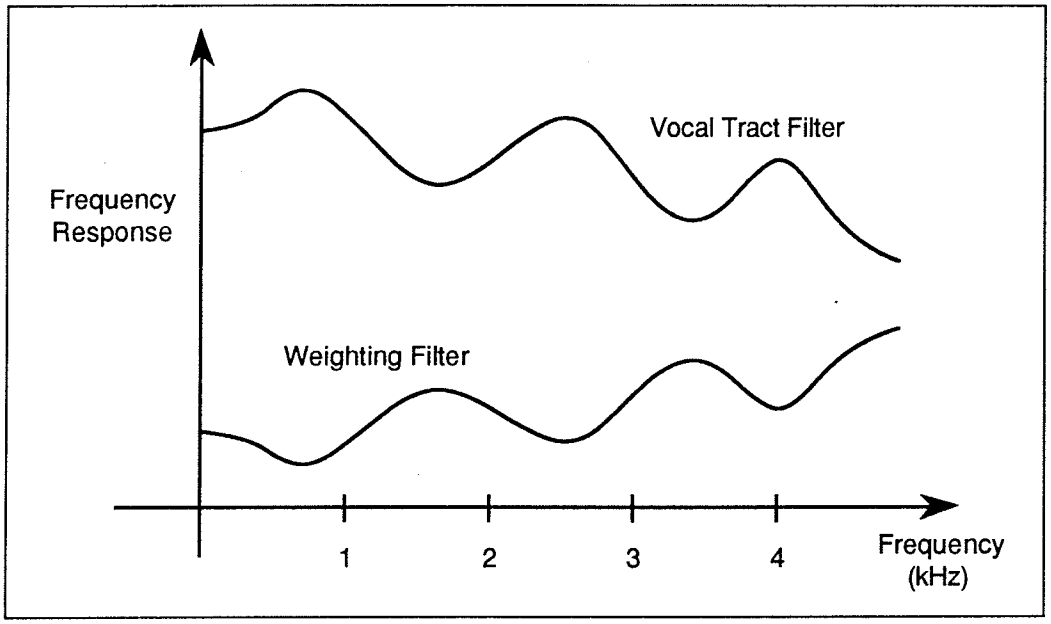


Fig. 2.22. Frequency response of the weighting filter for a given vocal tract filter.

A direct measurement using Eq. (2.16) is not practical because it involves a Fourier transform for each trial of the code book parameter search. The solution is to apply the difference vector $\mathbf{s} - \mathbf{s}'$ into the filter $P(z)$, to obtain a weighted error $\mathbf{e}$. Since the envelope of the spectrum of $\mathbf{e}$ has been perceptually shaped, a mean square magnitude of $\mathbf{e}$ in the time domain is equivalent with Eq. (2.16) by Parseval's theorem [Fran69]. This approach is used in the analysis scheme of Fig. 2.21.

2.6 Quantization of CELP Parameters

Since the analyzer and synthesizer have been defined, we must show that the CELP parameters have a 4.8 kbit/s rate. Unfortunately, in the present form the parameters (especially the LP coefficients) still require more than 4.8 kbit/s to be transmitted. A further quantization reduces the representation bits. The LP coefficients are quantized using an efficient, fast, and robust scheme called line spectrum pair (LSP). The ACB entries are quantized using a simple delta quantization, while the remaining parameters are

left unquantized.

Quantization of LP Parameters

The LP parameters still have high variability. As shown in Fig. 2.23, the zeros of the inverse filter $A(z)$ in Eq. (2.2) lie inside the unit circle. Although the unit circle has bounded the variability of the zeros, they can still be anywhere inside the unit circle. This zeros variability directly translates to variability of LP parameters.

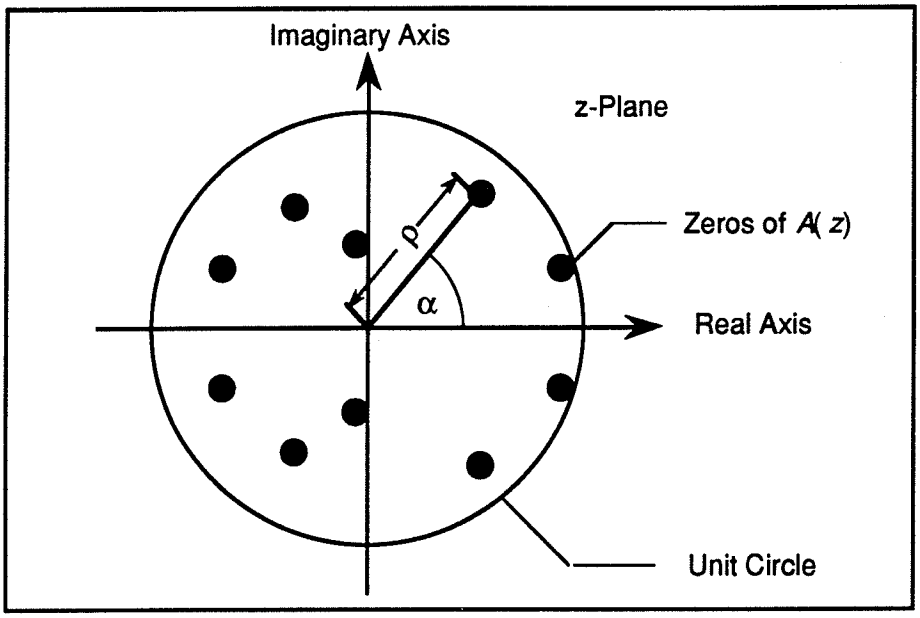


Fig. 2.23. An example of locations of zeros of $A(z)$.

Similar to the code book case mentioned earlier, an efficient quantization may fail if the input to the quantization has high variability. For example, the modulus ρ and argument α of every zero must be transmitted. Since a zero can be anywhere inside the unit circle, many quantization levels are needed for both ρ and α, resulting in high bit-rate. The variability of the quantization input must then be reduced prior to quantization. Different methods use different variability reduction.

Techniques of quantizing LP parameters can be categorized as either scalar

quantization or vector quantization [BGGM80]. In scalar quantization, the LP parameters are individually quantized [GrMa76]. This results in a low distortion quantization but high bit rates. On the other hand, a vector quantizer maps the LPC parameters as a vector into one subset of a partition set according to a certain equivalence relation [LiBG80]. This approach can provide lower bit representation and distortion. However, in practice the vector quantization requires higher complexity. In addition, the resulting representation is more sensitive to channel noise [AtCK89]. Despite its promising future, vector quantization is excluded from current CELP system due to those two shortcomings.

Techniques of scalar quantization includes arcsine of reflection coefficients (ASRC) [ViMa75], log area ratio (LAR) [GrMa76], and LSP [SoJu84]. It has been shown experimentally that for a CELP system, all of the above provide almost similar performance [AtCK89]. The LSP technique is chosen because, as shown later, the resulting representation is efficient and robust to noise [KaRa86], [SoJu84], and can be quickly obtained [CaTW90], [HaHe90]. In addition, it is possible to apply an interpolation for spectral smoothing during a transition between consecutive blocks.

Instead of working on the LP coefficients, the LSP technique works on the zeros of $A(z)$ defined in Eq. (2.2). The technique reduces the variability of the zeros by representing them with zeros of another polynomials $P(z)$ and $Q(z)$ which now lie strictly on the unit circle. Here, $P(z)$ and $Q(z)$ must completely define $A(z)$, and, consequently, $A(z)$ can be reconstructed from $P(z)$ and $Q(z)$. For $A(z)$ of order 10, Soong and Juang proposed such $P(z)$ and $Q(z)$ as [SoJu84]:

$$\begin{aligned} P(z) &= A(z) + z^{-11}\ A(z^{-1}) \\ Q(z) &= A(z) - z^{-11}\ A(z^{-1}) \end{aligned} \tag{2.18}$$

(This $P(z)$ should not be confused with the perceptual weighting filter). Conversely, $A(z)$ can be obtained easily from $P(z)$ and $Q(z)$ by:

$$A(z) = \frac{P(z)+Q(z)}{2} \tag{2.19}$$

Thus we can transmit zeros of $P(z)$ and $Q(z)$ instead of coefficients or zeros of $A(z)$. Efforts are made now to efficiently represent the zeros of $P(z)$ and $Q(z)$.

The $P(z)$ and $Q(z)$ have 11 zeros each [SoJu84] [KaRa86], satisfying:

(1) The zeros of $P(z)$ and $Q(z)$ lie on the unit circle of the z-plane.

(2) The zeros of $P(z)$ and $Q(z)$ are interlaced with each other on the unit circle.

(3) $z = -1$ and $z = 1$ are zeros of $P(z)$ and $Q(z)$, respectively.

(4) The other zeros occur in complex conjugate pairs.

Figure 2.24 shows the properties of the zeros of the polynomials. These restrictions considerably reduce the variability of the zeros that can further be utilized as follows.

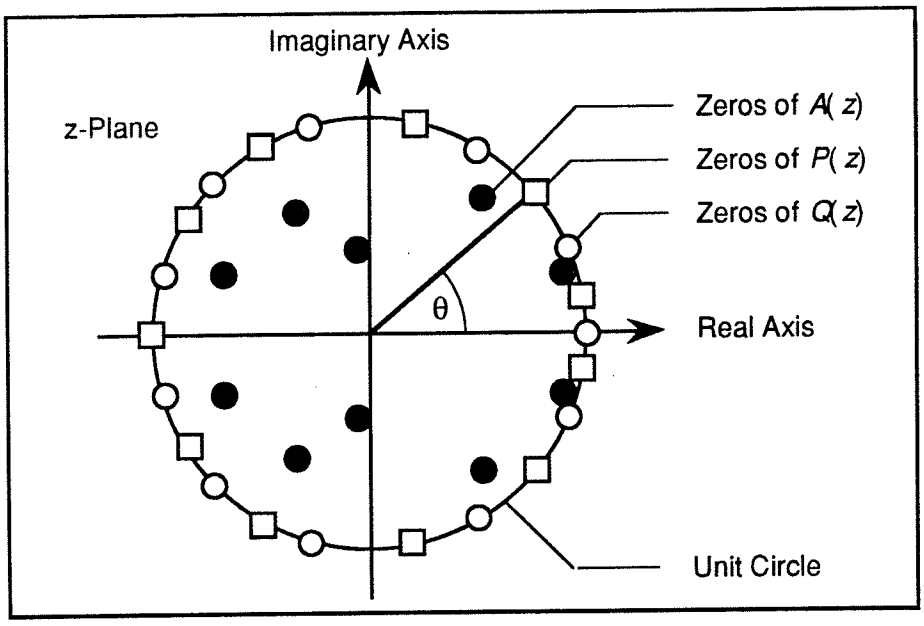


Fig. 2.24. An example of locations of zeros for $m = 10$. (After [KaRa86] with modifications).

First, since the 10 zeros of each polynomial occur in complex conjugate pairs (except for $z = -1$ and $z = 1$), it is sufficient to consider the zeros that are located on the upper semicircle of the unit circle only.

Second, since the $z = -1$ and $z = 1$ are always the zeros for $P(z)$ and $Q(z)$, respectively, they are considered redundant and can be eliminated. The resulting polynomials are:

$$P'(z) = \frac{P(z)}{1+z^{-1}} = \sum_{i=0}^{5} p_i\, z^{-i} + \sum_{i=6}^{10} p_{10-i}\, z^{-i}$$
$$Q'(z) = \frac{Q(z)}{1-z^{-1}} = \sum_{i=0}^{5} q_i\, z^{-i} + \sum_{i=6}^{10} q_{10-i}\, z^{-i} \qquad (2.20)$$

The $P'(z)$ and $Q'(z)$ polynomials still share the same properties with $P(z)$ and $Q(z)$ except for property (3), but have less number of coefficients.

Third, every point on the unit circle can be precisely determined by the angle (argument) of the corresponding point in a polar coordinate system. This eliminates the need to transmit the modulus of the point. The unit of each angle can be expressed in Hz by taking the frequency sampling (8 kHz) as 360 degree. The frequency term is used to express unit circle points of the zeros of the $P'(z)$ and $Q'(z)$ polynomials. The frequencies of the zeros are called LSP frequencies.

The zeros of $P(z)$ can be obtained by substituting $z = e^{j\omega}$ in the Eq. (2.20) and finding five ω between $0 \le \omega < \pi$ that make $P'(z) = 0$. The zeros of $Q(z)$ can also be obtained similarly. If we define $x = \cos(\omega)$, Eq. (2.20) becomes:

$$\begin{aligned} P'(x) &= 32x^5 + 16p_1x^4 + 8(p_1-5)x^3 + 4(p_3-4p_1)x^3 + 2(p_4-3p_2+5)x \\ &\quad + p_5 - 2p_3 + 2p_1 \\ Q'(x) &= 32x^5 + 16q_1x^4 + 8(q_1-5)x^3 + 4(q_3-4q_1)x^3 + 2(q_4-3q_2+5)x \\ &\quad + q_5 - 2q_3 + 2q_1 \end{aligned} \qquad (2.21)$$

The desired ω corresponds to the roots of Eq (2.21). The corresponding LSP frequency for each desired ω is:

$$f_i = \frac{\omega_i}{2\pi} \times \text{sampling frequency} \qquad (2.22)$$

We now have 10 LSP frequencies, 5 from $P'(z)$ and 5 from $Q'(z)$, that completely represent $P(z)$ and $Q(z)$. We transmit the 10 LSP frequencies instead of the 10 LP coefficients to the receiver. At the receiver, the 10 LSP frequencies are used to reconstruct $P(z)$ and $Q(z)$ which further reconstructs $A(z)$ using Eq. (2.20). The LP parameters are recovered from the coefficients of $A(z)$.

Fourth, since the zeros of $P(z)$ and $Q(z)$ are interlaced with each other, and all 10 zeros must share the upper semicircle, the range of an angle of a zero is limited. This is very useful in quantizing the LSP frequencies. Studies on dynamics of the angles result in efficient allocations of representation bits of every LSP frequency. One way of allocation that is adopted by FS-1016 provides 16 quantization levels for 4 LSP frequencies and 8 quantization levels for the remaining 6 LSP frequencies (see Appendix A). Thus a set of LP parameters require total 34 bits.

Quantization of Code Book Parameters

In present form, SCB entries are not expected to have an efficient quantization, because there is no knowledge regarding the SCB entries that can be utilized to reduce their variability. Furthermore, since the SCB entries are already in an integer form, they are ready for transmission. Thus SCB entries are left unquantized. For every subframe, 9 bits are needed for an SCB entry due to the SCB size of 512 recomended by FS-1016.

In contrast, an ACB entry directly corresponds to pitch period (Sections 2.34 and 2.4). Since the pitch period does not change immediately due to limited response time of the vocal cord, we can expect that consecutive entries do not differ tremendously. Thus a *delta* quantization can be introduced, i.e., the entry is represented by the difference between the actual entry and the previous entry. For an ACB with 256 entries as recommended by FS-1016, 8 or 6 bits are allocated with or without delta quantization, respectively. For a transmission through noisy channels, delta quantization of all ACB entries is not recommended because a single error propagates through all the future frames.

A solution adopted by FS-1016 interleaves the use of delta quantization. For every odd subframe, the actual entry is transmitted, while for every even subframe, delta quantization is applied.

The gain factors of the two code books are quantized into 32 levels each. Thus each gain factor needs 5 bits.

Bit Allocation of CELP Parameters for Transmission

Since speech can be considered stationary within 30 ms, the LP coefficients are obtained every 30 ms. As mentioned before, new code book parameters are obtained even more frequently, (7.5 ms). To reach 4.8 kbit/s, the maximum number of bits to be sent within 30 ms is:

30×10^{-3} s $\times$ 4.8 kbit/s = 144 bits.

With this quantization, the 4.8 kbit/s bit rate can be achieved. We even have 6 extra bits that can be used for synchronization, error correction, and future expansion, as shown in Table 2.1.

Table 2.1. Bit allocation for transmitting CELP parameters.

TYPE	ALLOCATION		TOTAL
Linear Predictor	3,4,4,4,4,3,3,3,3,3		34
Adaptive CB	Entry : 8+6+8+6	Gain: 5 x 4	48
Stochastic CB	Entry : 9+9+9+9	Gain: 5 x 4	56
Synchronization	1		1
Future Expansion	1		1
Error Correction	4		4

2.7 A Strategy for Error Protection

A strategy for error protection is needed because the 4 bits available are simply not sufficient to protect all data. Fortunately, not all of the CELP parameters affect speech

intelligibility to the same degree, and not all the bits of the CELP parameters have the same significance. So, only the most sensitive bits or parameters are protected, as proposed by FS-1016. However, for mobile communications this approach may not be adequate, and the bit rate must be increased to accommodate more protection bits.

The details on the strategy for a FS-1016 CELP, including identification of parameter sensitivity, are described in [CaWT89], and we outline the strategy in this section. The error protection scheme first concentrates on preventing errors that are perceptually disturbing, i.e., squeaks (sharp and high pitch sounds), blasts (loud, clipped sounds), and roughness. Having controlled these problems, the error protection scheme applies FEC to the most sensitive bits.

Protection of LP Parameters

The LSP can be used to ensure the stability of the LP parameters. The noise may alter the parameters such that the resulting LP filter becomes unstable, i.e., the roots are outside the unit circle of the z-plane. An unstable LP filter causes disturbing errors. Fortunately, if the LSP frequencies are monotonically increasing, the corresponding LP filter is stable. Thus a simple stability rule can be applied. If an LSP frequency causes nonmonotonicity, it is replaced by the same LSP frequency from the previous frame. If the corrected LSP are still nonmonotonic, all LSP frequencies are replaced by the previous, stable LSP frequencies.

An additional smoothing also helps reduce the effects of error. Here, the LP filter coefficients are changed every subframe with interpolated values between the previous coefficients and the current coefficients. The interpolation rule for every LSP frequency is:

$$\text{LSP}(i) = \left(\frac{9-2i}{8}\right) \text{Previous LSP} + \left(\frac{2i-1}{8}\right) \text{Present LSP} \tag{2.23}$$

Here, LSP(i) is a specific LSP frequency for the i^{th} subframe, i = 1, 2, 3, or 4.

Furthermore, FS-1016 CELP avoids the first and the last LSP frequencies from

being coded close to unstable values, and two adjacent LSP frequencies from being coded closer than 15 Hz. Those two constraints limit the gain of the resulting LP filter. Thus, the blasts and squeaks caused by undesired high gain due to the LP filter error are avoided.

Protection of SCB Gain

Errors on the SCB gain can be annoying because they can cause blasts. The effects are more perceived during quiet speech. A smoother can be used to reduce the error effects. One smoothing algorithm computes the average and variance of several previous gain magnitudes. The gain average and variance are used to set a threshold with which the current gain is compared. If the gain exceeds the threshold, it should not be used and should be replaced by the average value. During the quiet speech, where the gain average is low, the threshold must be tighter. This algorithm can eliminate the majority of the error effects.

Protection of SCB Entry

Since the SCB contains random impulses, smoothing is inappropriate. The errors in selecting a sequence of the random impulses can create noisy speech. Fortunately, it is found that this noisy speech is less perceptually disturbing. Thus, the SCB entry bits are left unprotected.

Protection of ACB Entry

Pitch has a continuity behaviour, thus smoothing of ACB entry is appropriate. An intensive test concludes that the pitch parameter is the most sensitive parameter because the errors cause very rough speech [CaTW89]. Two smoothing processes are applied. First, the smoothing is inherent when the delta quantization is embedded in the ACB searching process. Here, the searching is done only on the range allowed by the delta quantization. Thus sudden pitch changes are not allowed. Second, the same smoothing algorithm used

in SCB gain can be used. Here, the threshold can be fixed, thus simplifying the algorithm.

The smoothing is enhanced by a Hamming (15,11) code. Only 3 MSB bits of an entry are protected by the FEC because the smoother can compensate the error in the other bits. Furthermore, since the regular entry is more significant than the delta entry, the FEC protects the bits of the regular entry only.

Protection of ACB Gain

The ACB gain is another sensitive parameter. The ACB gain also exhibits a smooth property although not as much as the SCB gain or the ACB entry. Thus, the same smoothing algorithm can be used. However, the smoother cannot well protect the gain because the gain has many regions where the smoother fails. So, the Hamming code is also used to enhance the error protection of the ACB gain.

Forward Error Correction

A Hamming code (15,11) is used to enhanced the smoothing strategy. It only needs 4 bits as allocated in Table 2.1. The code protects the parameters which are insufficiently handled by the smoother. The code protects 11 bits consisting of all the MSBs of the ACB gain factors, the 6^{th}, 7^{th}, and 8^{th} bits of odd subframe ACB entries, and a reserved (future expansion) bit. There are 4 parity bits, and each of them represents an even parity on 7 of the 11 bits (see Appendix A). The parity bits can be used to correct a single bit error in any of the 11 bits.

The parity bits are also used to estimate the error rate of the channel. If the estimation concludes that the channel is error free, it can turn off all the smoothers. In this error free case, an unnecessary smoothing may alter the correct parameters.

Burst-Error Protection

Finally, prior to the transmission, the data bits of CELP parameters are scattered

according to a table defined by the proposed FS-1016. This scattering process can reduce the effects of burst errors.

2.8 Summary

In this chapter, we show that the CELP technique can compress speech down to a rate of 4.8 kbit/s. Such a low bit rate speech representation is possible through a utilization of a code book. A model of the human speech production system reduces the variability of the target waveform, resulting in a code book with a size of 512. Furthermore, the use of the waveform coding criterion results in a high speech quality.

To obtain the compressed speech (CELP parameters), the CELP analysis uses a filter estimation technique (linear prediction) and an analysis-by-synthesis method. The speech quality of the CELP system is enhanced using a better speech measure during the analysis. The use of a perceptual weighting filter makes a mean square error minimization more meaningful.

Quantization of CELP parameters including LSP, delta coding, and gain quantization reduces the representation bits to a 4.8 kbit/s rate. Six extra bits are even available that can be used for simple protocol and error protection.

An error protection strategy is applied to the CELP parameters. The bit limitation in the data stream restricts us from protecting all the data bits using FEC. The error protection scheme exploits the smooth behavior of the parameters to protect the synthetic speech from the disturbing effects, such as squeaks, blasts, and roughness. An FEC is used to enhance the scheme by protecting the most sensitive parameters and the parameters that have many unsmooth regions.

CHAPTER III
FAST ALGORITHMS FOR REAL-TIME CELP SYSTEMS

The main problem of implementing the CELP technique in real time is that the analysis requires very-high computational power. In an earlier implementation, a CELP system needed 125 s of Cray-1 CPU time to process 1-s speech [ScAt85]. Most of the computational power was consumed by the code book searching due to the exhaustive closed loop process mentioned in Chapter II.

Considerable effort has been made since then to reduce the computational power needed to perform CELP compression [InLK91], [LaIK91]. Oyama introduced an open-loop approach of obtaining SCB parameters, but the resulting quality was poor [Oyam85]. Several methods were proposed by Trancoso and Atal [TrAt86]. The methods include the restructuring of the code book search that leads to a scheme called *joint optimization* used by most of the CELP systems today. There are several algorithms proposed later, but most of them can be seen as improvements of the Trancoso and Atal methods [KlKK90].

[KlKK90] classifies the fast algorithms according the aspects on which the algorithms are based: (i) special code book design, including center clipping [DaGe86], [CaTW90], (ii) multistage searching [DaGe88], [CoSe86], (iii) domain transformation [TrAt86], [MoHo87], (iv) autocorrelation methods [TrAt86], [KlKK88], (v) overlapping code book entries [KlKK88], and (vi) look-up tables [AMDM87], [ChGe87]. In practice, some of the algorithms may be combined to yield a faster algorithm. We combine three algorithms: the special code book design, domain transformation, and overlapping code book entries. The other algorithms are not chosen because they either cannot be combined, or need more resources [KlKK90].

We present here fast search algorithms used by the system. First, we restructure the CELP analysis scheme. We then describe the joint optimization scheme, which reduces total search computation by a factor of 175,000. A special code book is explained,

followed by the utilization of the special code book, which reduces a filtering computation inside the joint optimization by a factor of 123. Finally, we discuss a suboptimal scheme that can be used if the algorithms are still not sufficient.

3.1 Restructuring of CELP Analysis

The analysis structure, especially the code book parameter search shown in Fig. 2.21, is not efficient. For every tentative trial, two filtering processes must take place. The first is the LP filtering inside the CELP synthesizer block. The second is the perceptually weighting filtering.

The structure can be improved by moving the weighting filter before the summation (comparison) block. The LP filter can then be combined with the weighting filter. To compensate for the change, the original speech, **s**, must be filtered by the weighting filter before entering the summation block. Using this approach, shown in Fig. 3.1, each tentative trial needs only one filtering, thus reducing the computation.

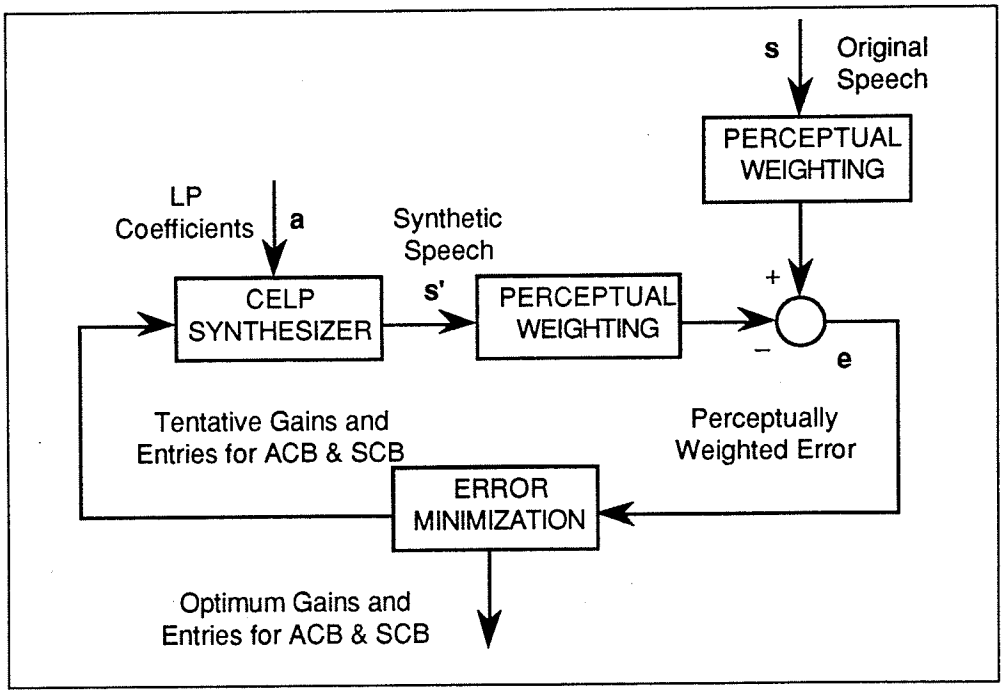


Fig. 3.1. An improved code book parameter search.

Furthermore, the filtering process can be replaced by a matrix multiplication. It is shown later that the matrix structure can be exploited to provide fast searching. Let $h(i)$ be an impulse response of the receiver's LP filter at a sampling instant i. The output of the filter is then a convolution between the excitation impulses and $h(i)$. If the excitation impulses and the output samples are denoted as $t(n)$ and $q(n)$, n = 1, ..., 60, respectively, we have:

$$q(n) = \sum_{i=1}^{n} h(n\text{-}i)\, t(i), \qquad n = 1, ..., 60. \tag{3.1}$$

If we see $t(n)$ and $q(n)$ as 60-element vectors **t** and **q**, respectively, Eq. (3.1) is equivalent with a matrix multiplication [Atal89]:

$$\mathbf{q} = H\,\mathbf{t} \tag{3.2}$$

Matrix H is a 60×60 matrix defined as:

$$H = \begin{bmatrix} h(0) & 0 & \cdots & 0 \\ h(1) & h(0) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h(59) & h(58) & \cdots & h(0) \end{bmatrix} \tag{3.3}$$

However, **q** is not exactly the synthetic speech, **s'**, that is produced in the receiver. We recall that if the LP filter is implemented in a direct form, as assumed by the proposed FS-1016, there are data stored inside the delay elements resulting from the previous excitation. The effect of this data on the synthetic speech is called ringing [ChGe87]. The ringing signal is obtained by applying a 0 input to the current LP filter. We call the ringing signal the zero response signal. If we name this zero response signal as **z**, the synthetic speech is

$$\mathbf{s}' = \mathbf{z} + \mathbf{q} = \mathbf{z} + H\mathbf{t} \tag{3.4}$$

In the analyzer, the cascade of LP and weighting filters can also be represented by a matrix W in a similar way. Let the impulse responses of the cascaded filter be $w(i)$. If we apply $\mathbf{t}$ to this cascaded filter and call the output as $\mathbf{x}$, we have relationships

$$x(n) = \sum_{i=1}^{n} w(n-i)\, t(i), \qquad n = 1, \ldots, 60. \tag{3.5}$$

and

$$\mathbf{x} = W\,\mathbf{t}, \tag{3.6}$$

where W is a 60×60 matrix defined as

$$W = \begin{bmatrix} w(0) & 0 & \cdots & 0 \\ w(1) & w(0) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ w(59) & w(58) & \cdots & w(0) \end{bmatrix} \tag{3.7}$$

It follows from the construction of W that the weighting filter can be seen as a matrix P satisfying:

$$P = WH^{-1} \tag{3.8}$$

The code book parameter search is supposed to minimize Eq. (2.15). Substituting Eq. (3.4) and (3.8) into Eq. (2.15), we have

$$\mathbf{e} = P(\mathbf{s} - \mathbf{s}') = P(\mathbf{s} - \mathbf{z} + H\mathbf{t}) = P(\mathbf{s} - \mathbf{z}) + W\mathbf{t} \tag{3.9}$$

Eq. (3.9) represents the searching process in Fig. 3.1, where the matrix W represent the cascade of the LP filter and the weighting filter. Here, the ACB and SCB produces excitation $\mathbf{t}$, based on the set of code book parameters provided by the error

minimization block. If we define

$$\mathbf{y} = P(\mathbf{s} - \mathbf{z}) \tag{3.10}$$

we can redraw Fig. 3.1 to be the structure in Fig. 3.2. This structure is used in developing the joint optimization scheme.

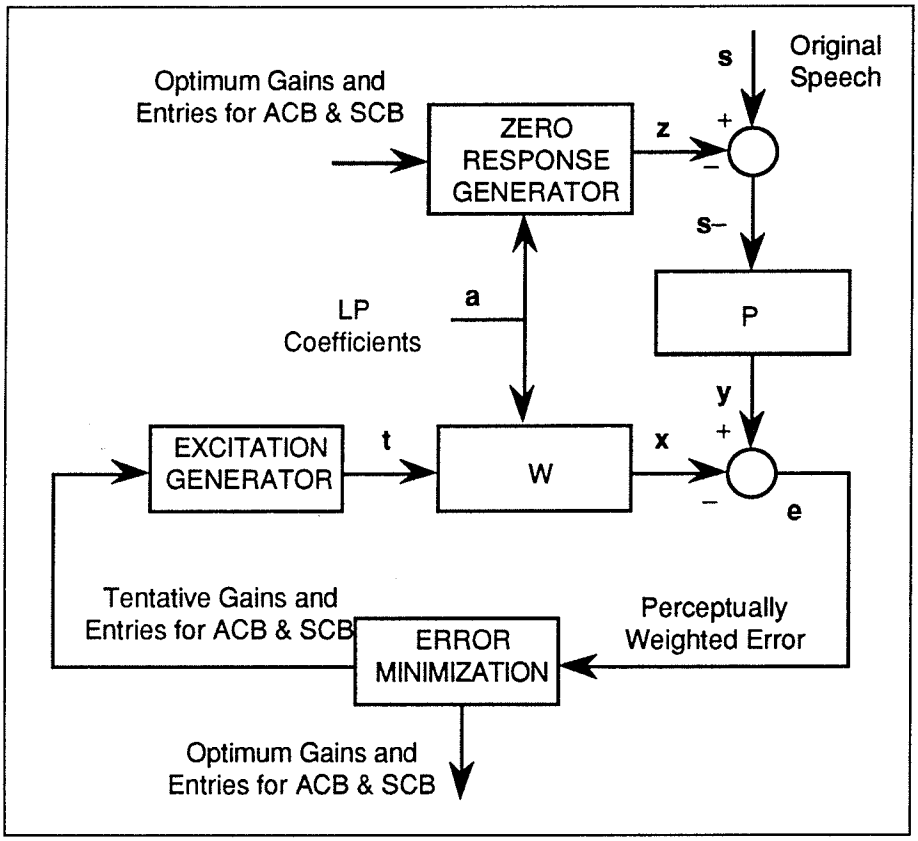


Fig. 3.2. An efficient structure of code book parameter searching for CELP analysis. The cascade of LP filter and the weighting filter is represented by a matrix W.

It should be noted that the zero response generator applies the obtained CELP parameter to a replica of the synthesizer to keep tracking the contents of delay elements. Once the replica obtains proper values of the delay element content, it applies zero impulses to its LP filter to get the zero responses.

3.2 Joint Optimization Scheme

To reduce the computation, the analyzer searches one code book at a time. To illustrate this, let the size of the SCB and ACB be K_s and K_a, respectively. The numbers of possible gain values for SCB and ACB are L_s and L_a, respectively. We define one trial as a complete process from the error minimization block supplying a set of code book parameters, to the measurement of its corresponding ||e||. To complete the code book parameter search for one odd subframe, the number of trials (N.O.T.) is:

$$\text{N.O.T.} = K_s \times K_a \times L_s \times L_a \tag{3.11}$$

Now, if we search the ACB before we search the SCB, we can see the process as two identical code book searches done serially. Thus, the N.O.T. of one odd subframe becomes:

$$\text{N.O.T.} = K_s \times L_s + L_a \times K_a \tag{3.12}$$

Typical values for K_s, K_a, L_s, and L_a in a full implementation of the proposed FS-1016 are 512, 256, 32, and 32, respectively. Using the original searching, the N.O.T. is 134,217,728. However, using the serial searching, the N. O. T. is 24,576, which is 0.018% of the former N. O. T.

With joint optimization [ScAt85], we can find the best gain factor for each entry without searching. Consider a scheme of parameter searching for one code book shown in Fig. 3.3. Here, minimizing ||e|| is equivalent with minimizing $||e||^2$. Thus, given matching target **y** and LP parameter **a**, the minimal error detector block must find the best combination of entry i and gain factor g such that the error energy $||e||^2$ is minimized. Since we have

$$\mathbf{e} = \mathbf{y} - \mathbf{x} \tag{3.13}$$

we have to minimize:

$$\| \mathbf{e} \|^2 = \langle \mathbf{e},\mathbf{e} \rangle = \langle \mathbf{y}-\mathbf{x},\mathbf{y}-\mathbf{x} \rangle = \langle \mathbf{y},\mathbf{y} \rangle - 2\langle \mathbf{x},\mathbf{y} \rangle + \langle \mathbf{x},\mathbf{x} \rangle \tag{3.13}$$

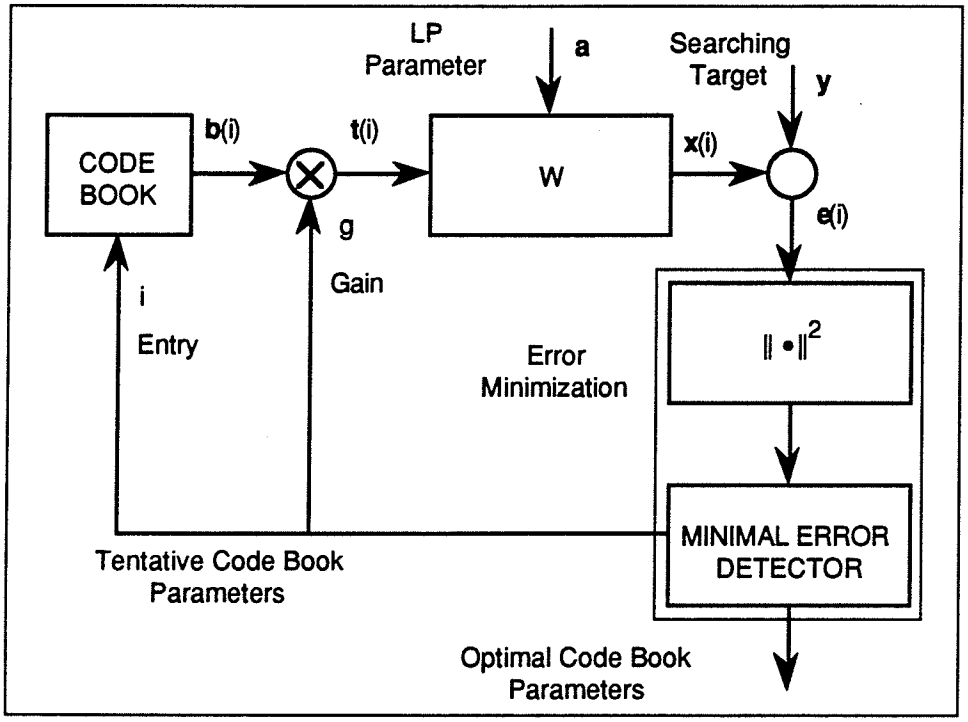


Fig. 3.3. A code book searching diagram.

Since the choice of i and g does not affect the term $\langle \mathbf{y},\mathbf{y} \rangle$, while $\langle \mathbf{x},\mathbf{x} \rangle$ is always positive, minimizing $\|\mathbf{e}\|^2$ in (3.13) is equivalent to maximizing p, which is defined as

$$p = 2\langle \mathbf{x},\mathbf{y} \rangle - \langle \mathbf{x},\mathbf{x} \rangle \tag{3.14}$$

Instead of trying every combination of i and g to maximize p, the joint optimization scheme assigns a g for each i. The assigned g is the best g that maximizes p for a certain i. In other words, g is not a free parameter anymore, instead it is a function of i. We call this g as $g(i)$. To find the $g(i)$ we define

$$\mathbf{v}(i) = W\,\mathbf{b}(i), \tag{3.15}$$

where **b**(i) is a code book prototype addressed by i. Calling the p for the specific **b**(i) as $p(i)$, we can rewrite Eq. (3.14) to become

$$\begin{aligned} p(i) &= 2\langle\, \mathbf{y}, g(i)\mathbf{v}(i)\,\rangle - \langle\, g(i)\mathbf{v}(i), g(i)\mathbf{v}(i)\,\rangle \\ &= 2g(i)\langle\, \mathbf{y}, \mathbf{v}(i)\,\rangle - g(i)^2\langle\, \mathbf{v}(i), \mathbf{v}(i)\,\rangle \end{aligned} \tag{3.16}$$

Using calculus, $p(i)$ is maximized by $g(i)$ if: (i) $g(i)$ is the root of the first derivative of $p(i)$ over $g(i)$, and (ii) the value of the second derivative of $p(i)$ over $g(i)$ evaluated by the $g(i)$ in (i) is negative. Working on Eq. (3.16), we find the $g(i)$ that satisfies the two conditions does exist as

$$g(i) = \frac{\langle \mathbf{y}, \mathbf{v}(i) \rangle}{\langle \mathbf{v}(i), \mathbf{v}(i) \rangle} \tag{3.17}$$

For this $g(i)$, we can rewrite Eq. (3.16) as

$$p(i) = g(i)\,\langle \mathbf{y}, \mathbf{v}(i) \rangle \tag{3.18}$$

The search process then becomes a search for i only. For every i, a $g(i)$ obtained from Eq. (3.17) is used to compute $p(i)$ using Eq. (3.18). An i is chosen (with its $g(i)$ automatically) if its corresponding $p(i)$ is maximum over all i in the code book. The implementation scheme of Eq. (3.18) is called joint optimization, as shown in Fig. 3.4.

The N.O.T. for one odd subframe becomes

$$\text{N.O.T} = K_s + K_a \tag{3.19}$$

For the specific values of M and N, N.O.T. is 768, which is 0.00057% of the earlier

N.O.T., or a reduction by a factor of 175,000. Here, one trial consists of selecting a code book sequence $\mathbf{b}(i)$, computing $\mathbf{v}(i)$, computing inner products $\langle \mathbf{y},\mathbf{v}(i) \rangle$ and $\langle \mathbf{v}(i),\mathbf{v}(i) \rangle$, obtaining $g(i)$, obtaining $p(i)$, and finally testing the $p(i)$ to order the i.

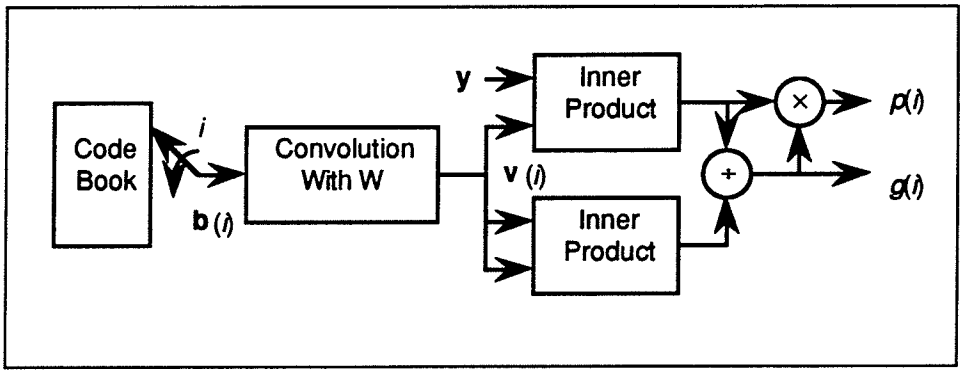


Fig. 3.4. A joint optimization scheme. For every i, there are corresponding $p(i)$ and $g(i)$. The i is chosen when its $p(i)$ attains maximum. The gain factor $g(i)$ is provided automatically.

3.3 Special SCB Design

The joint optimization computation can be reduced by designing a special SCB. There are three features of the special SCB used by the proposed FS-1016. First, the elements of the SCB created by a Gaussian noise generator are center clipped. Second, the elements are also ternarized so that the positive and negative elements are replaced by 1 and –1, respectively. Third, sequences stored inside the SCB overlap each other.

Creating a Special SCB

In the proposed FS-1016, the special SCB is created using the following procedure. A Gaussian noise source creates a set of 1082 random impulses, with unit variance, such that 77% of the impulses have absolute amplitudes not exceeding a threshold of 1.2. All the impulses with absolute amplitudes ≤ 1.2 are replaced by 0. The remaining positive and negative impulses become 1 and –1, respectively. The resulting set of

impulses, denoted by $r(i)$, becomes a source of SCB sequences. If the 512 sequences of an SCB are $\mathbf{b}(0)$, ..., to $\mathbf{b}(511)$, the standard defines the elements of the sequences as shown in Table 3.1.

Table 3.1. A table for construction an SCB.

Entry	Sequence	Element of sequence $\mathbf{b}(i)$			
511	$\mathbf{b}(511)$	$r(0)$	$r(1)$	$\cdots$	$r(59)$
510	$\mathbf{b}(510)$	$r(2)$	$r(3)$	$\cdots$	$r(61)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$\mathbf{b}(i)$	$r(2(511-i)+1)$	$r(2(511-i)+2)$	$\cdots$	$r(2(511-i)+59)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	$\mathbf{b}(1)$	$r(1020)$	$r(1021)$	$\cdots$	$r(1079)$
0	$\mathbf{b}(0)$	$r(1022)$	$r(1023)$	$\cdots$	$r(1081)$

Effects on the Convolution Computation

To understand how the three features help reduce the computation, we consider the joint optimization scheme shown in Fig. 3.4. The scheme requires computations of $\mathbf{v}(i)$ using Eq. (3.15), as represented by the convolution block in the figure. For each N-element $\mathbf{v}(i)$, the convolution must do at least $N(N+1)/2$ multiplications and $N(N-1)/2$ additions. Thus, for 512 entries and N equal to 60, there are $512\times60(60+1)/2 = 936{,}960$ multiplications and $512\times60(60-1)/2 = 906{,}240$ additions.

With the special SCB, the number of multiplications and additions are considerably reduced. Using Table 3.1 and Eq. (3.15), we compute the column vectors $\mathbf{v}(i)$ for all i. Two examples of the matrix multiplication results, $\mathbf{v}(511)$ and $\mathbf{v}(510)$, are shown as follow:

$$\mathbf{v}(511) = \begin{bmatrix} r(0)w(0) \\ r(0)w(1)+r(1)w(0) \\ r(0)w(2)+r(1)w(1)+r(2)w(0) \\ r(0)w(3)+r(1)w(2)+r(2)w(1)+r(3)w(0) \\ r(0)w(4)+r(1)w(3)+r(2)w(2)+r(3)h(1)+r(4)w(0 \\ \vdots \end{bmatrix}$$

$$\mathbf{v}(510) = \begin{bmatrix} r(2)w(0) \\ r(2)w(1)+r(3)w(0) \\ r(2)w(2)+r(3)w(1)+r(4)w(0) \\ r(2)w(3)+r(3)w(2)+r(4)w(1)+r(5)w(0) \\ r(2)w(4)+r(3)w(3)+r(4)w(2)+r(5)h(1)+r(6)w(0) \\ \vdots \end{bmatrix}$$

Notice that the terms inside the dashed lines of the **v**(511) are identical to the terms in the **v**(510) due to the overlapping property of the **b**(i). This property is repeated for the rest of the **v**(i). If we already have **v**(510), we do not have to compute again all the terms for **v**(511). Instead, we just compute the terms containing r(0) and r(1), and add them to the results of **v**(510), thus reducing the computation of **v**(511).

The algorithm thus begins with a computation of **v**(0), that is used by the joint optimization to compute $g(0)$ and $p(0)$. The **v**(1) is then computed by first computing the additional terms containing $r(1020)$ and $r(1021)$ because those terms are not available in **v**(0), as implied by Table 3.1. The result is a column vector **v**'(1) where the j^{th} element is $r(1020)w(j)+r(1021)w(j-1)$. The vector **v**(0) is modified into **v**"(1) by replacing every i^{th} element with its $(i-2)^{th}$ element, and the first two elements with 0. The **v**(1) is then obtained by a vector addition of the **v**'(1) and **v**"(1). The process is repeated in a similar fashion for all other entries.

The computation of the convolution for the first entry is similar to the previous computation, i.e., $N(N+1)/2$ multiplications and $N(N-1)/2$ additions. However, for each of the other entries, there are $2N-1$ multiplications and $2N-3$ additions. Since there are K_s-1 of them, the convolution requires $N(N+1)/2 + (K_s-1)(2N-1)$ multiplications or 62,639 multiplications, and $N(N-1)/2 + (K_s-1)(2N-3)$ additions or 61,557 additions. The numbers are 6.8% of the previous computations for both multiplications and additions.

It should be noted that the ACB searching can also utilize the overlapping algorithm since its sequences are inherently overlapping, as described in Sections 2.3.4 and 2.4.

Most likely we do not even have to compute many of the additional terms, because 77% of $r(i)$ are zero. The algorithm then checks an $r(i)$. If an $r(i)$ is zero, the multiplications and additions corresponding to the $r(i)$ are skipped. Taking this into consideration, the convolution requires less than $N(N+1)/2 + 23\%(K_s-1)(2N-1)$ multiplications or 15,816 multiplications, and $N(N-1)/2 + 23\%(K_s-1)(2N-3)$ additions or 15,521 additions. The numbers are 1.7% of the computations for both multiplications and additions using a regular code book.

Furthermore, the other nonzero $r(i)$ are either 1 or –1. This means we do not need multiplications anymore, because the operations can be done by additions or subtractions. If the additions and subtractions are assumed to be computationally equivalent, the computation needed for the convolution is less than $N(N-1)/2 + 23\%(K_s-1)(2N-3)$ plus $N(23\%N+1)/2$ (from the approximately maximum multiplication of the first entry) or 15,961 additions. This is a reduction by a factor of 123.

In summary, the three features of the special SCB incorporated with the matrix structure of WH greatly reduce the computation of the joint optimization scheme. It is reported that the algorithms lead to a single processor implementation [CaHG90].

3.4 Partial Searching

The required computation power heavily depends on the number of code book entries. If all the algorithms mentioned above are still not sufficient, we can reduce the number of entries searched during the analysis. This partial search approach was used in early implementations of CELP systems [BrBH89].

The delta search of the ACB can be considered as a partial search. Here, the previous entry is used as the centre of the subset of entries used for searching. The delta search works on 64 entries in the neighborhood of the centre. The computation needed is one-quarter of the regular search. Thus, besides reducing the bit rate, delta coding also reduces the computation.

3.5 Computational Power Requirement for Code Book Searching

If we apply all the fast algorithms above, we can estimate the number of instructions, i.e. multiplications or additions, required for a complete code book search. The following is an estimation for one search of one subframe.

For a SCB search, the convolution needs $N(N-1)/2 + 23\%(K_s-1)(2N-3) + N(23\%N+1)/2$ instructions. Since some DSPs include a combination of multiply and add (MADD) instructions, the convolution needs $N(N-1)/2 + 23\%(K_s-1)(2N-3)$ MADD instructions. The two inner product terms need K_sN MADD instructions each.

For an ACB search, the convolution needs approximately $N(N-1)/2 + (K_a-1)(N-1)$ MADD instructions. The inner-product terms require K_aN MADD instructions each. It should be noted that for each even subframe, N is one quarter of that of odd subframe because of the delta encoding.

Thus for one frame, we need approximately 427,000 MADD. Since we have 33.33 frames each second, we need 14.2 MIPS. We still have to add computation for LP parameters, quantization, error protection, protocol, and overheads that are not included in the above calculations. [CaTW90] gives estimation values of the computation power

needed based on the number of entries that is more optimistic than our calculation, shown in Table 3.2.

Table 3.2. An estimation of computation power needed for implementing CELP using the fast algorithms. (Source [CaTW90]).

SCB Size	SCB Searching (MIPS)	Full Duplex (MIPS)	DSP Rating (MIPS)
64	1.1	5.5	11
128	2.1	6.5	13
256	4.2	8.5	17
512	8.3	12.6	25

The most dominant computation is the computation of inner-product terms, despite their very simple computation structure. For every frame, inner product terms for ACB and SCB need approximately 322,560 MADD (10.8 MIPS), which is 75% of the total computation. Future fast CELP must either introduce better algorithms for obtaining those terms, or use special hardware for computing inner products.

3.6 Summary

CELP systems cannot be practically implemented without introducing sub-optimal schemes and fast algorithms. Single DSP implementations become possible after restructuring the CELP analysis, converting the filters into matrix forms, implementing code book searches in serial, utilizing a joint optimization scheme, and designing a special SCB. Furthermore, the computation can be reduced by reducing the number of code book entries during the searching process. The approximation of the processor power needed to perform a real-time CELP system provides the feasibility of implementing the CELP system using a single digital signal processor system.

CHAPTER IV
SYSTEM DESIGN AND IMPLEMENTATION

A CELP speech compression system has been designed and implemented on an IBM PC-AT equipped with a TMS320C30 DSP evaluation module (EVM). A systematic design results in a simple and flexible architecture, a manageable organization, and an efficient implementation. A signal processor architecture overcomes difficulties caused by algorithm complexity and resource limitation. The design approach leads to a fast CELP system. In its present form, CELP analysis is still not real time. However, it can be shown that real-time operation is possible.

In this chapter, we first describe how the functional requirements, specified in Chapter I, and methods for designing a signal processing system [DeLH88] guide the architectural design. System architecture is a description of the system attributes, i.e., its conceptual structure and functional behaviour, and more specifically, its major processing units (functional blocks), interconnection of the functional blocks, a set of commands, methods of issuing the commands, and procedures of data access (input and output, I/O) [Kins90], [HiPe87]. An architectural design translates the system requirements to the description of the system attributes. It is a user oriented description, thus it explains how combinations of processes of the functional blocks perform all user requirements.

Besides performing all the required functions, the architecture must remain simple and implementable. It must be flexible regardless of the implementation platform. For example, if the platform has a powerful processor, the system must be able to take advantage of the processor without modification of the architecture.

The platform and the task complexity of the functional blocks determine the distribution of the process to the platform resources, resulting in a description of system organization. A system organization includes system partitioning, actual components,

actual interconnections, software partitioning, and software code [Kins90], [HiPe87]. In other words, a system organization decides which part of the platform is to be responsible for performing a functional block. It also decides whether a particular functional block is implemented in hardware or software. To some extent, it tells us how we should implement a functional block, e.g., its algorithm or method, and how the communication and data transfer between blocks should be done, e.g., the protocol and data format. Finally, a system organization provides a link between an architecture and an actual system realization. Thus, a system organization is a platform (e.g., technology) and process (e.g., algorithm) oriented description. It should be noted that due to the design complexity, we develop the software code during system implementation instead of during system organization.

Finally, an implementation converts each process to hardware circuits and/or software codes. Despite the guidance from the system organization, the realization possesses its own difficulties, due to the complexity of the algorithms and the platform. We resolve the difficulties using a development methodology.

4.1 System Architecture

Since the system must compress and reconstruct speech, it consists of a CELP transmitter and a CELP receiver, as shown in Fig. 4.1. The CELP transmitter contains a CELP analyzer as the compressor, while the CELP receiver contains a CELP synthesizer as the decompressor. The CELP parameters can be transmitted through low bit-rate digital-radio.

In addition to a synthesizer and analyzer, a CELP system also has data converters, and data communication blocks, as shown in Fig. 4.2. Figure 4.1 shows that the transmitter accepts analog speech. Since CELP compression works on digital speech, the CELP transmitter must include an analog-to-digital (A/D) converter. Furthermore, a digital communication system usually uses a communication protocol scheme, including

synchronization and error protection. The scheme is also included in the CELP transmitter. Consequently, a CELP receiver also includes a digital-to-analog (D/A) converter and a communication protocol scheme (including error detection and correction).

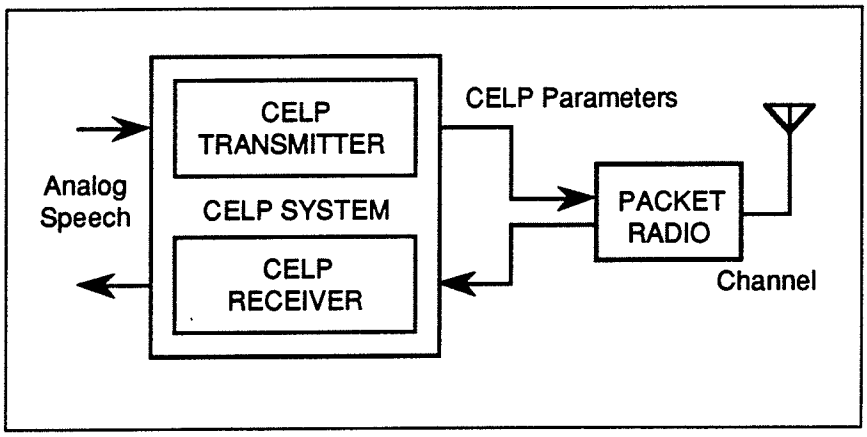


Fig. 4.1. A basic architecture of a CELP system for packet radio. The CELP transmitter maps an analog speech signal into CELP parameters. The CELP receiver maps back the CELP parameters into an analog speech signal.

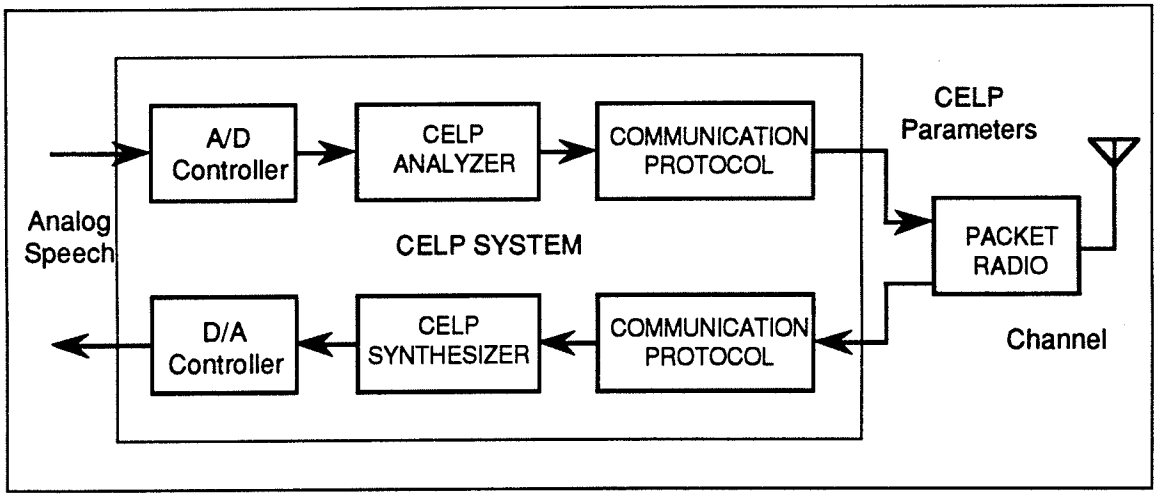


Fig. 4.2. A CELP system with its main parts. The upper section is the CELP transmitter, while the lower section is the CELP receiver. The system needs an A/D converter and D/A converter because CELP compresses and decompresses data digitally. The communication protocol includes an error protection scheme.

From a signal processing perspective, the system consists of a controller, a data processor, and an input/output (I/O) processor, that is a typical architecture of a digital signal processor [Kins90], [DeLH88] (see Fig. 4.3). The controller accepts user commands and provides corresponding process controls, including process scheduling and data flow. The system needs the controller because the CELP system must respond to user commands interactively.

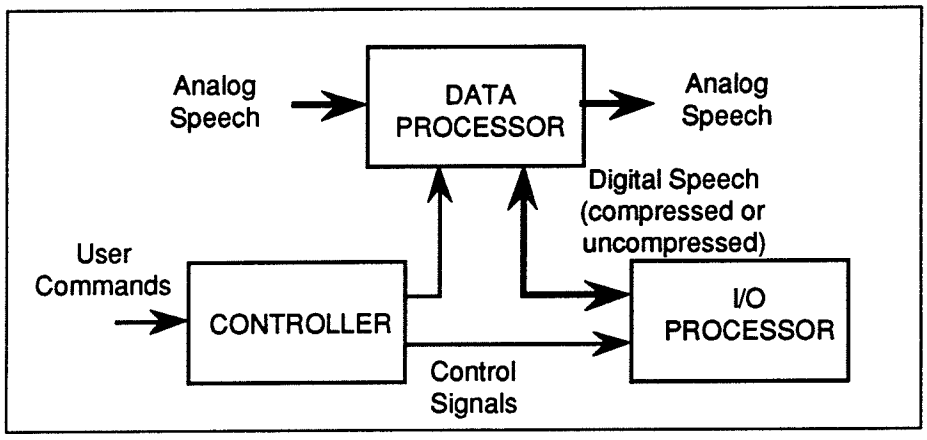


Fig. 4.3. A basic architecture of a CELP system from a signal processor perspective.

The data processor consists of a signal pre-processor, a digital data-processor, and a signal post-processor [Kins90]. The signal pre-processor transforms the analog speech to a PCM form that is suitable for CELP algorithms. The digital data-processor applies CELP algorithms to the input data. The signal post-processor converts the processing results back to the analog form for a listening process.

Finally, the I/O processor manages data exchanges in a digital form. The data exchanges take place between the data processor and a packet radio channel or a storage device.

Since the CELP algorithm is complex, we introduce a CELP engine as a data processor. Here, the data processor has its own commands, controller, local storage,

processing capability, and I/O processor. This engine tremendously reduces the interactions between the main controller and the data processor, thus reducing the time loss due to the interaction process (overhead). It reduces the burden of the controller, the global storage, and global data path, giving more resources and time for user interactions or I/O processing.

The architecture of our system, shown in Fig. 4.4, combines both functional (Fig. 4.2) and signal processor (Fig. 4.3) architectures. Here, the data processor consists of functional blocks, transforming speech data into several required forms for performing CELP speech compression, CELP speech decompression, and error protection processes, as explained before in Chapters II and III.

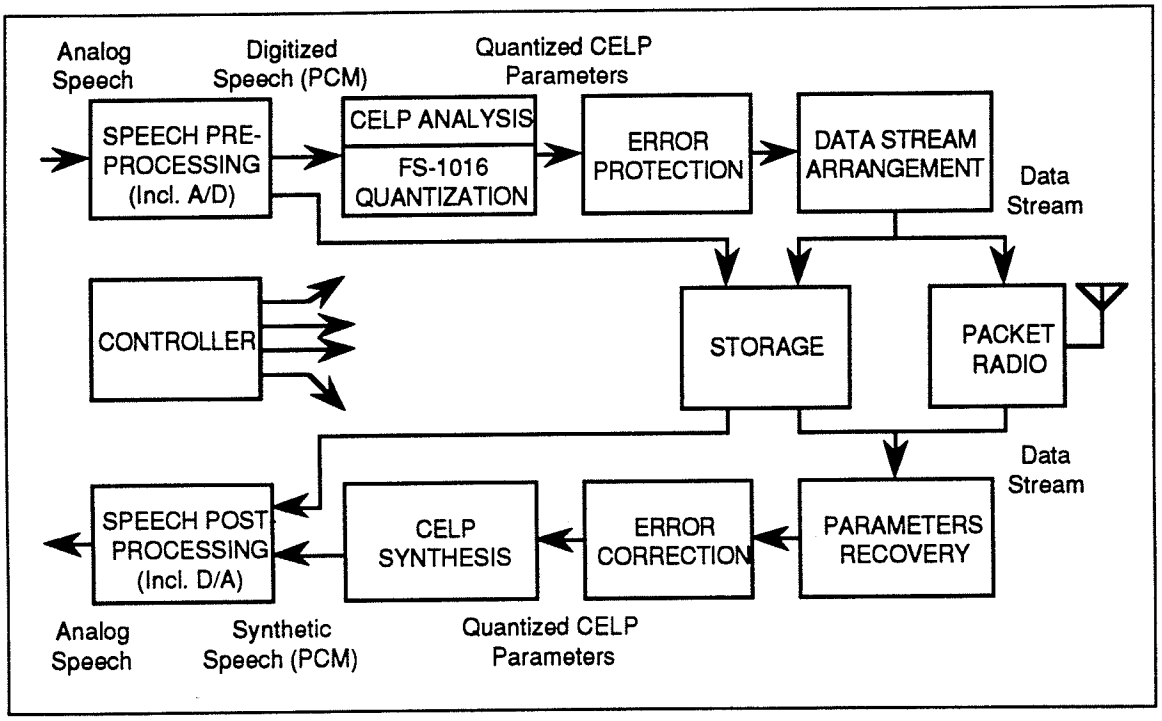


Fig. 4.4. The CELP system architecture.

This architecture can accommodate all the system requirements in both speech communication and storage modes. In the speech communication mode, the controller

instructs the data processor to acquire analog speech and digitally compress the speech. The data processor then protects and arranges the compressed speech, and the controller sends the resulting data stream to the packet radio channel. For speech receiving (listening), the controller sends the received data stream to the data processor. The data processor recovers the compressed speech from the data stream, and corrects any detected error(s). A CELP synthesizer then reconstructs the speech, and the D/A converter converts the resulting PCM data into audible analog signals. In the speech storage mode, the controller activates the same process except it sends or receives the compressed speech to or from the storage, respectively.

With the above categorization of roles, the system becomes manageable, implementable, and flexible. The system blocks can then be developed modularly such that the complexity of CELP algorithms does not appear at the system level. The modularity makes the system development manageable because each functional block can be developed separately. Not only that, a fast processor or hardware can implement the computationally extensive blocks, while a slower processor implements the other blocks. We can then preserve the architecture regardless of the available implementation platform.

We discuss now parts of the architecture starting with the controller, the data processor, and finally the I/O processor.

4.1.1 Controller

In a real-time system, a controller is very important to ensure a process occurs in correct order and with a proper timing. Furthermore, a single process may be needed by more than one function. For example, in both speech communication and storage, the CELP compression is activated. With a controller, the process can be shared, thus avoiding duplication.

Since there are several functions of the system, the controller must understand which function is desired by a user. Thus, for each function we must assign a user

command. We then design the controller to associate a certain user command with a list of scheduled actions taking place in the data processor and the I/O processor.

At the beginning, the controller always waits for a user command. When the command is received, the controller parses the command through a command table. A valid command has a corresponding sequence of control signals that activates a certain process in the data processor and data media. After all processes are completed, the controller waits for a new command. Details of this process are discussed later in system organization.

Since the system is used for speech communication and storage, we group the commands to two categories: (i) commands for speech communication, and (ii) commands for speech storage. In spite of this command categorization, commands from both categories may share several processes, as mentioned earlier.

User Commands for Speech Communication

The system must communicate in either real time or non real time. In real-time mode, the speech data coming from an analog audio system are compressed and directly sent to the packet radio channel. At the same time, the speech data coming from the channel are decompressed and sent to the audio system for speech listening. In non real-time mode, the system sends or receives a file containing compressed speech data to or from the channel. Thus, there are four basic user commands needed to perform speech communication:

(i) *Send CELP Speech*,

(ii) *Receive CELP Speech*,

(iii) *Send CELP File*, and

(iv) *Receive CELP File.*

It should be noted that for the full-duplex mode, another command can be created that executes *Send CELP Speech* immediately after executing *Receive CELP Speech.*

User Commands for Speech Storage

Here, the system must store or retrieve the speech data to or from the storage, respectively. In speech storage, the analog speech is acquired, compressed, and stored in a file. In speech retrieval, the compressed speech is obtained from a file, decompressed, and sent to the analog audio system for listening.

Since PCM is a universal technique of speech representation, a capability of storing PCM speech is also added. In addition, the PCM data can be obtained easily using a low-cost A/D module. This feature is completed by providing file conversions between PCM speech data and CELP speech data.

There are then six user commands needed to perform speech storage:

(i) *Record CELP Speech*,

(ii) *Play CELP Speech*,

(iii) *Record PCM Speech*,

(iv) *Play PCM Speech*,

(v) *Convert PCM to CELP*, and

(vi) *Convert CELP to PCM.*

All user commands, except the format conversion commands, must be executed in real time, otherwise the user may experience losses of data. A fast system may perform all the processing in real time. However, since the most demanding process is the CELP compression, a slower system may have to record speech in the PCM form before compressing it. This scheme allows a non real-time speech recording on a slow, low-cost platform, while maintaining the real time playback capability.

Command Control Phases

For real-time applications, the time allocated for executing a user command is restricted. The proposed FS-1016 defines a set of CELP parameters for every block (frame) of 240 speech samples. Since we use a sampling rate of 8 kHz, all real-time user commands must finish their process on a block of data in less than 30 ms.

For non real-time applications, no restriction is applied for command execution time. However, long execution times may cause inconveniences to the user. To avoid that, the commands should use all suitable fast routines that are used by the real-time commands.

4.1.2 Data Processor

The data processor digitally transforms speech data through its functional blocks. As a CELP engine, it has data buffers and a set of commands. The data buffers hold input data and results of a process executed by a command. The data buffers are also known by the main controller. The buffers are:

- A/D BUFFER, (240 words) which holds PCM data resulting from an A/D converting process. Every 30 ms, this buffer contains 240 new speech samples.
- D/A BUFFER, (240 words) which holds PCM data for a D/A converting process. To have uninterrupted speech playback, this buffer must be updated with new data every 30 ms.
- BUFFER240, (240 words) which (usually) holds PCM data from and for an I/O processor.
- BUFFER9, (9 words) which holds CELP parameters from and for an I/O processor.
- INTERMEDIATE, (240 words) which holds PCM data. This is a general purpose buffer that helps A/D and D/A buffers during real-time recording and playback.

The commands of the data processor shown in Table 4.1 perform all the necessary functions related to CELP compression and decompression as explained in Chapters II and III.

Table 4.1. Commands of the data processor to perform CELP compression and decompression.

Command	Description
Record PCM	Read A/D BUFFER, and send data to I/O
Playback PCM	Receive data from I/O, and write D/A BUFFER
Record CELP	Read A/D BUFFER, convert to CELP in BUFFER9, and send data to I/O
Playback CELP	Receive I/O data, convert to PCM in BUFFER240, and write D/A BUFFER
Receive Host Data	Receive data from I/O, and put to one of the buffers
Send Host Data	Send data to I/O from one of the buffers
Convert PCM to CELP	Convert PCM data from BUFFER240 to CELP data, and put the result in BUFFER9
Convert CELP to PCM	Convert CELP data from BUFFER9 to PCM data, and put in BUFFER9

4.1.3 I/O Processor

There are two data media needed for our system, i.e., (i) a packet radio channel, and (ii) a storage device. For real-time communication, the system needs a packet radio that has at least 4.8 kbit/s throughput. On the other hand, for non-real-time communication, the system has more freedom and the 4.8 kbit/s throughput is not necessary. For real-time speech recording and playback, the system also needs a digital data storage with a net data rate at least 4.8 kbit/s. However, since the system also provides recording and playback in PCM format, it needs a data rate of at least 8 kbyte/s or

16 kbyte/s for 8 bit or 16 bit PCM, respectively.

Although the structure of the I/O processor is much simpler than the data processor, we still provide data buffers and commands that are specific for the I/O processor. The buffers are accessible by the controller and hold data from and for the packet radio or storage. The buffers are:

- CHANNEL IN BUFFER, (9 words, or 144 bits) which holds CELP data from packet radio channel. For real-time communication, this buffer contains 9 new CELP words (144 bits) every 30 ms.
- CHANNEL OUT BUFFER, (9 words, or 144 bits) which holds CELP data for packet radio transmission.
- DISK BUFFER, (96,000 words) which holds both PCM and CELP data from and for storage. This size is enough for 12 s of 16-bit PCM, or 320 s of CELP data.

As shown in Table 4.2, the commands initiate I/O functions that interchange digital data among the data processor, channel, and storage.

Having identified the functional blocks and their corresponding processes, we now want to have an efficient implementation of them. We thus study the process characteristics and the platform capabilities to decide which part of the platform should perform each process, as shown in the following system organization.

4.2 System Organization

We implement the system on a PC-AT equipped with a TMS320C30 EVM. We can have many options of platforms ranging from special VLSI circuits, speech processor chip-based circuits, DSP-based circuits, to general purpose computers. However, there are three factors affecting our choice: (i) computational power (because of the CELP algorithm requirement), (ii) development difficulties, and (iii) system costs. The platform is chosen

because it satisfies the computational power requirement, it has a complete development system, and finally the system cost is low.

Table 4.2. Commands for I/O processor.

Command	Description
Write Channel	Write CELP data to CHANNEL OUT BUFFER
Read Channel	Read CELP data from CHANNEL IN BUFFER
Write PCM File	Write a certain size of PCM data from DISK BUFFER to a file
Read PCM File	Read a certain size of PCM data from the storage to DISK BUFFER
Write CELP File	Write a certain size of CELP data from DISK BUFFER to a file
Read CELP File	Read a certain size of CELP data from the storage to DISK BUFFER

4.2.1 Hardware Organization

The platform allows us to implement all the blocks of the architecture (except the packet radio), as shown in Fig. 4.5. We first look at its capabilities, and then use them to decide the block distribution across the platform.

Capabilities of Hardware Platform

The host computer has an INTEL 80286 processor, 512 Kbytes memory (minimum), a hard disk, and a serial port (RS232C). This computer can run programs written in 8088/80286 assembly or high level languages such as C or Pascal. The host computer is a 100% IBM PC-AT compatible, so that low level access can be done in a standard way.

The EVM is a half-size board that is installed in a PC-AT slot. The EVM contains a

digital signal processor, memory, digital I/Os, and an analog interface. The TMS320C30 is a 32-bit digital signal processor and capable of 60 ns instruction cycle at 33 MHz clock cycle. (However, the processor in the EVM runs on a 30 MHz clock thus increasing the instruction cycle by approximately 10%). The processor has 2K words of internal RAM, while the EVM has 16K words of zero-wait state static RAM for codes and data.

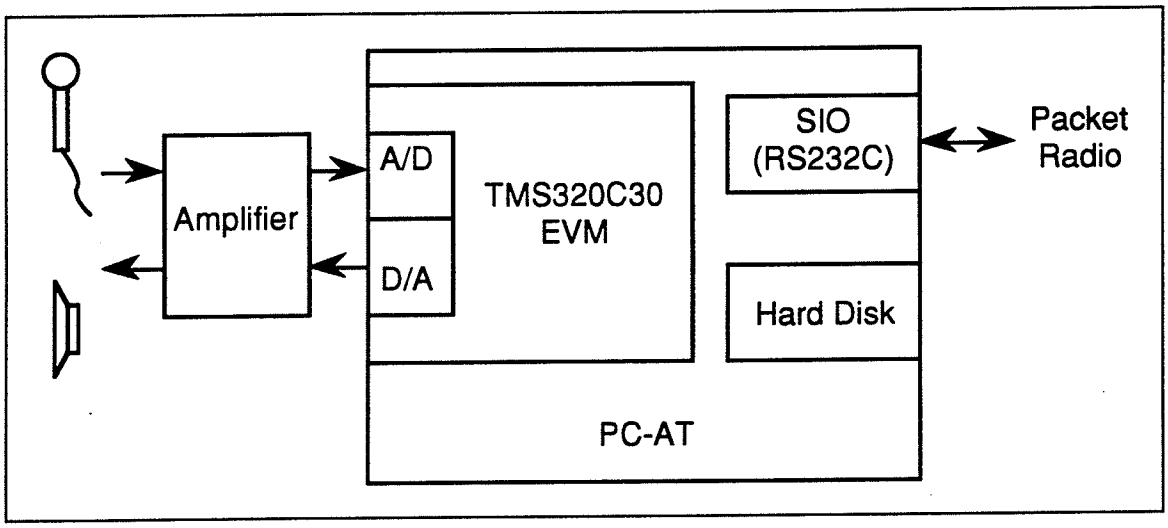


Fig. 4.5. The configuration of the hardware platform.

The EVM is equipped with an analog interface circuit (AIC) based on a TLC32044. The TLC32044 is a programmable 14-bit A/D and D/A converters in a single chip. It has a programmable filter and sampling rate controller. It should be noted that μ-law encoding/decoding is not necessary here because the dynamic range of the digital speech with a resolution of 14 bits (84 dB) surpasses the equivalent range of 72 dB of μ-law companded speech.

To communicate with its host, the EVM has a 16-bit bidirectional port. The port can pass 8 bit or 16 bit data. At the EVM side, the data interchanges can be driven by either interrupts, or status polling, or direct memory access (DMA). Every host writing or reading to or from the port causes an interrupt on the EVM. At the host side, the

interchanges are driven by status polling.

Block Distribution

We implement the system controller on the host PC because there is no extensive computation involved, and the controller has to interact with a user through a keyboard and a display. The PC-AT then has a user interface on which we can type-in user commands to select functions the system has to perform.

The data processor resides in the EVM because it needs high computational power and it has direct access to speech signals. Here, the AIC implements A/D and D/A sections. The TMS320C30 executes CELP analyzer and synthesizer algorithms. In addition, the digital signal processor implements error protection, and data stream arrangement. Thus, the EVM performs a CELP engine.

Finally, we implement the I/O processor on the PC. The storage device is the host hard disk, and the packet radio must connect to the host serial port.

4.2.2 Software Organization

Since some functional blocks, i.e., the A/D converter, D/A converter, pre-processor, and post-processor, have been implemented in hardware, the remaining blocks must be developed in software. The software runs on both the EVM and the host computer. The following is a description of the software blocks.

4.2.2.1 Controller

Figure 4.6 shows a flowchart of the controller. The PC program begins with initialization of the system including the serial port and the EVM. It also sets the system state to a default setting. The program displays a menu, and then waits for a command from a user. The user should press a key on the keyboard corresponding to a user command. If the user command is valid, the controller issues micro-commands (the

commands of data processor and I/O processor) related to the user command.

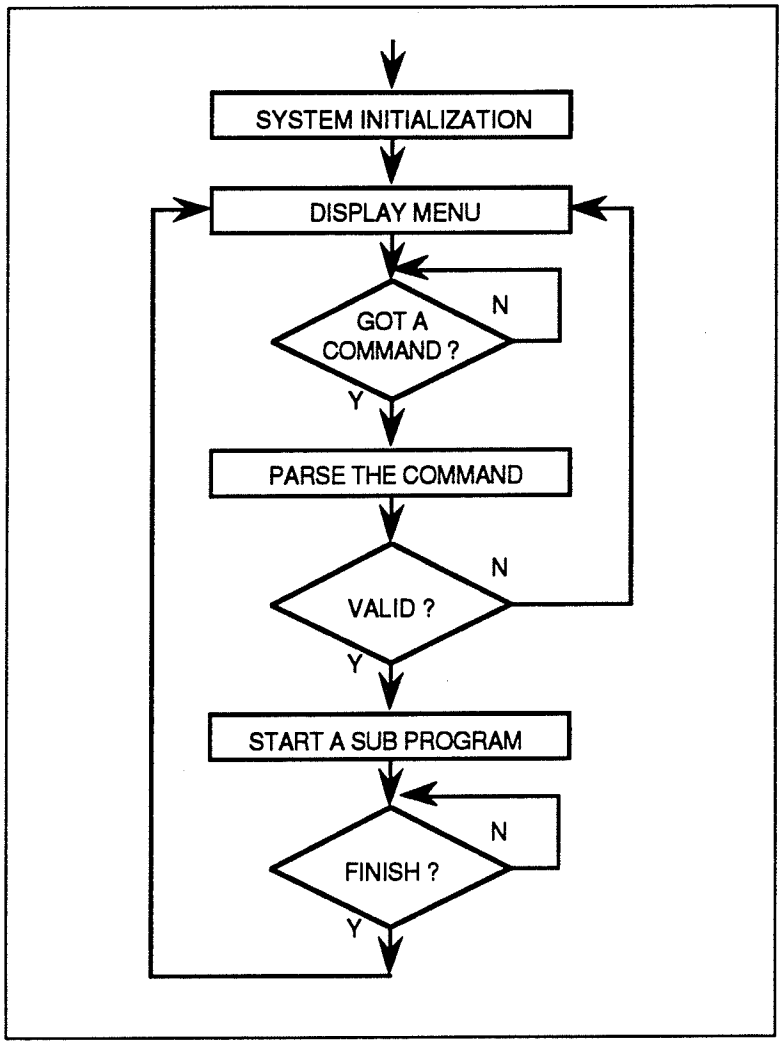


Fig. 4.6. A flowchart of the controller.

Since several user commands require executions of subprograms inside the EVM, a command protocol is needed. Here, the controller sends an EVM command and waits for an echo flag from the EVM acknowledging the acceptance of the command. If the echo does not come in a certain time period, the controller resets the EVM and tells the user that the EVM has failed in doing the required process. If the EVM acknowledges the EVM command, the controller then waits for another flag from the EVM informing that the process is done. Similarly, if the flag does not come in a certain period of time, the

controller resets the EVM and tells the user that the EVM has failed. Otherwise, the controller continues with the next process until all actions necessary to complete a user command have been done. To inform the user of a successful operation, the controller displays the menu and then waits for another user command.

A command protocol for the I/O processor is not necessary because the process is simple and the controller can reach all I/O functions (subroutines) directly. Thus, instead of sending commands, the controller simply calls equivalent subroutines.

The system employs such a controlling scheme because it provides a clear and structural command control. The scheme can control all system states. It simplifies system development because we can add commands modularly. This approach is also recommended by the EVM user's guide.

With the above structure, the user commands are translated into sequences of actions as shown in Table 4.3.

4.2.2.2 Data Processor

As a CELP engine, the data processor also has a local controller, local data processor, and local I/O processor. They are designed to perform the EVM commands. We distribute the data processor into several software sections: EVM controller, data acquisition and distribution, CELP analysis, CELP synthesis, error protection, and host-EVM data exchanges.

EVM Controller

The EVM controller works similarly with the main controller. Here, a command from the host starts a sequence of processes, as shown in Table 4.1. We thus identify subroutines corresponding to the processes, shown in Table 4.4.

There are two processes that are not fully controlled: data acquisition from the A/D converter and data distribution to the D/A converter. They are interrupt driven processes

that are enabled during initialization. Since we reserve two buffers for both processes, (A/D BUFFER and D/A BUFFER), a program can get new data by simply reading the A/D BUFFER. Similarly, for providing data to the D/A converter, a program just writes the data to the D/A BUFFER.

Table 4.3. User commands. The description shows a sequence of micro-commands to be performed from left to right to complete the execution of a command. Notice that all data processor and I/O processor commands have an EVM and I/O prefix, respectively.

Command	Description	
Send CELP Speech	EVM: Record CELP	I/O: Write CHANNEL OUT BUFFER
Receive CELP Speech	I/O: Read CHANNEL IN BUFFER	EVM: Playback CELP
Send CELP File	I/O: Read CELP file	I/O: Write CHANNEL OUT BUFFER
Receive CELP File	I/O: Read CHANNEL IN BUFFER	I/O: Write CELP file
Full Duplex Comm.	I/O: Read CHANNEL IN BUFFER EVM: Record CELP	EVM: Playback CELP I/O: Write CHANNEL OUT BUFFER
Record CELP Speech	I/O: EVM: Record CELP	I/O: Write CELP file
Play CELP Speech	I/O: Read CELP file	EVM: Playback CELP
Record PCM Speech	EVM: Record PCM	I/O: Write PCM file
Play PCM Speech	I/O: Read PCM file	EVM: Playback PCM
Convert PCM to CELP	I/O: Read PCM file EVM: Convert PCM to CELP I/O: Write CELP file	
Convert CELP to PCM	I/O: Read CELP file EVM: Convert CELP to PCM I/O: Write PCM file	

The two processes are not controlled because they must be activated in a constant period. Since the EVM controller must execute many other subprograms and wait for the results, it is impossible to properly control the activation timing of the data conversion and access.

Table 4.4. List of processes performed by the EVM. A sequence of the processes constitutes an EVM command. We organize the process to software and hardware sections.

Process	Software Sections	Hardware Sections
Write A/D BUFFER	Data acquisition	AIC
Read D/A BUFFER	Data distribution	AIC
Send data to host	Host-EVM data exchanges	DMA
Receive data form host	Host-EVM data exchanges	DMA
Convert PCM to CELP	CELP Analyzer Error Protection	DSP
Convert CELP to PCM	Error Protection CELP Synthesizer	DSP

Data Acquisition and Distribution from A/D and to D/A Converters

The data acquisition program, whose flow chart is shown in Fig. 4.7, is responsible for providing speech input data in PCM form. As mentioned earlier, the program is placed as an interrupt service routine that responds to periodic interrupts occurring every 125 μs, due to the 8 kHz sampling rate.

When an interrupt occurs, this program reads the contents of a data register of the AIC and writes it on the A/D BUFFER at the location pointed to by the buffer pointer. The pointer of the buffer is then incremented by one so that it points to the next data location. Since this buffer is a circular buffer, if the pointer is at the end of the buffer, the next location must be the beginning of the buffer.

Since the data compression works on a block of 240 speech samples, we have to wait for the data acquisition process to complete acquiring one block of speech before processing the block. If the pointer is pointing to the beginning of the buffer, we know that one block of data has been acquired, thus the data can be read. It should be noted that this scheme implies two things: (i) the pointer must be readable by the all subroutines that need access to the A/D BUFFER, and (ii) the data access from the buffer can be done faster

than a period of the interrupt.

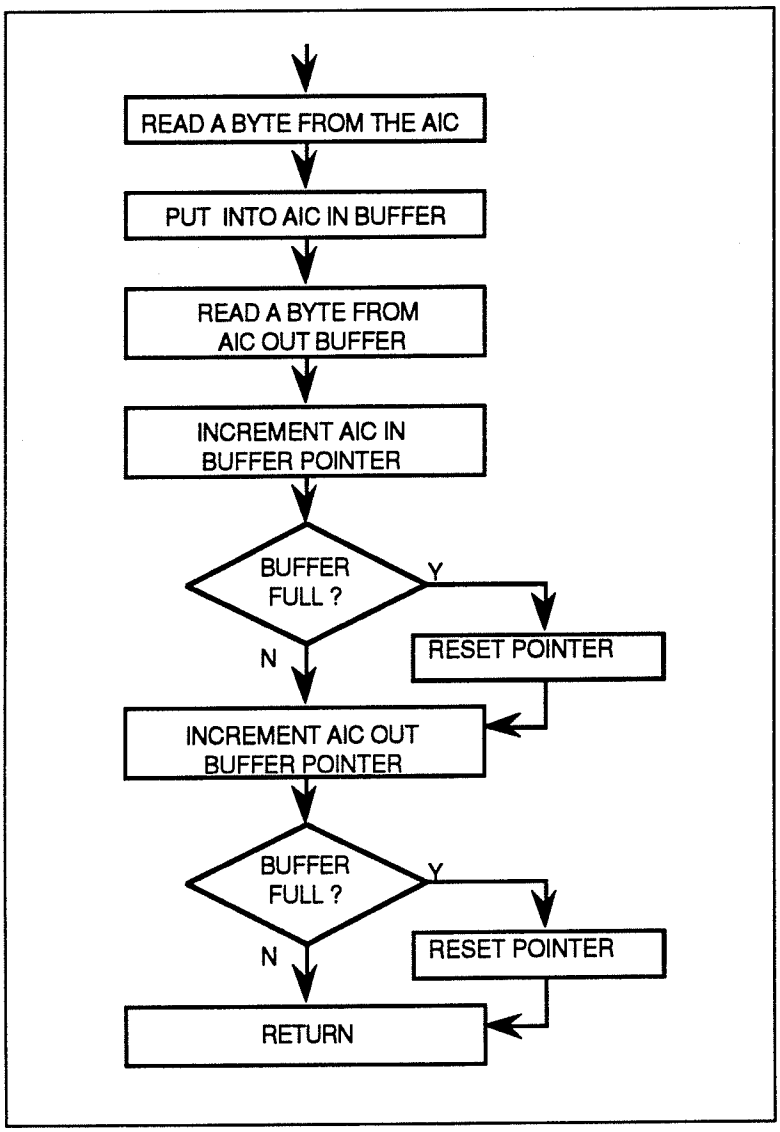


Fig. 4.7. A flowchart of the A/D and D/A interrupt service.

The data distribution program inverses the process. An interrupt causes a process of loading data from the D/A BUFFER into a data register of the AIC. Since the buffer is also a circular buffer, all pointer procedures mentioned above also apply.

Both processes can share the same source of interrupt because the A/D and D/A converters operate at the same conversion rate. Thus, the interrupt service routine contains

a program to do both A/D and D/A transfers.

CELP Analyzer

Figure 4.8 shows a flowchart of a CELP analyzer corresponding to the scheme developed in Chapters II and III (Fig. 2.20 and 3.3). This flowchart is a direct representation of the analysis scheme described in those chapters. The program reads 240 speech samples (one frame of speech) from a working buffer and weights the data using a Hamming window (see Eq. (2.13)). The program then performs LPC analysis on the windowed data to obtain the LP coefficients according to section 2.5.1. Here we utilize the Levinson-Durbin algorithm to solve Eq. (2.12) [Pars86].

The LP coefficients become the LSP parameters using Eq (2.20) and (2.21). Since we have a table of all possible values of x in Eq (2.20) (see Appendix A), we apply the value to that equation and check for a zero crossing. Each LSP frequency is a f_i corresponding to an x which causes a zero crossing.

Having the LP coefficients, we must now find the code book parameters. We first set up the LP filter using LP coefficients corresponding to the LSP. We then find code book parameters for every 60 speech samples according to the scheme of Fig. 3.3. We must repeat the search process three more times until we have code book parameters for all 240 speech samples.

To obtain ACB parameters, we must first provide the ACB searching target. The program computes the zero responses of the new LP filter and subtracts them from the original signal to compensate for the ringing from previous frames. A PW filtering of the compressed speech (see Eq. (2.17)) results in an ACB searching target. A code book search subroutine (containing joint optimization) then finds the ACB parameters.

To obtain SCB parameters, we also must provide the SCB search target. The program creates excitation impulses using ACB parameters, and then applies the impulses to the cascade of LP and PW filter. The resulting signal is subtracted from the ACB search

target mentioned earlier to get the SCB search target. Having determined the searching target, the program calls the joint optimization subroutine to find the SCB parameters.

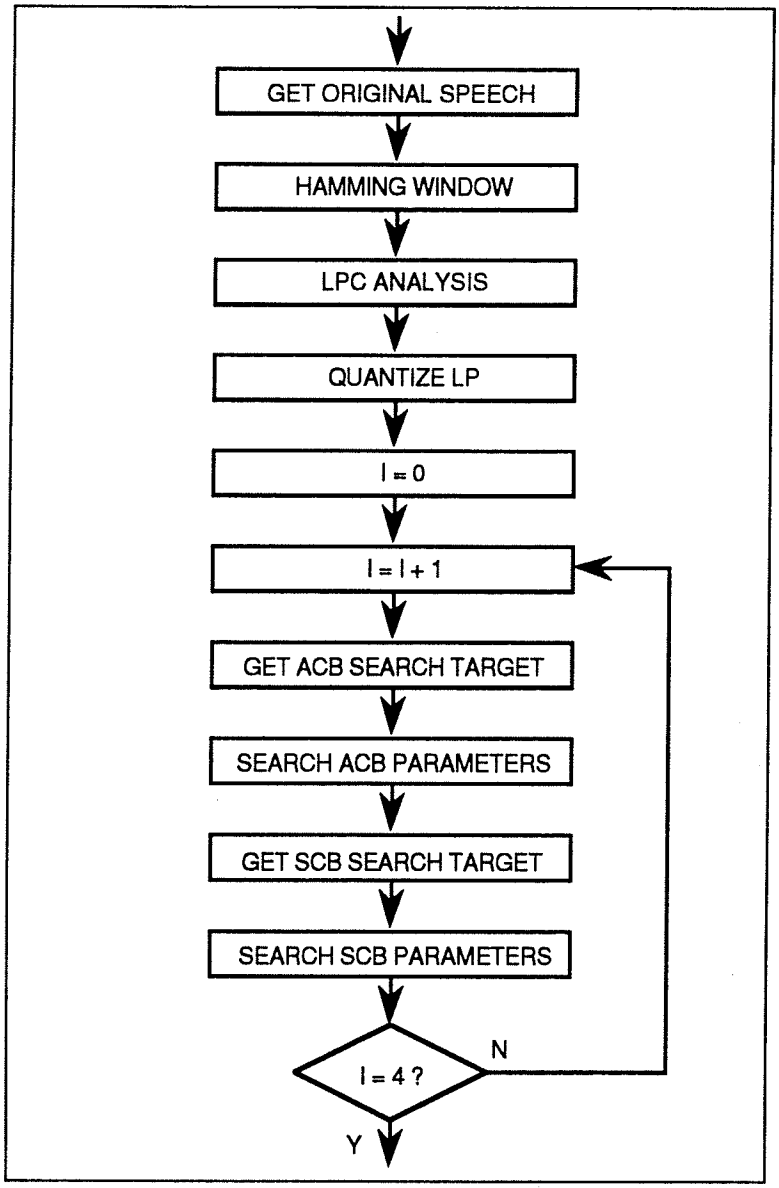


Fig. 4.8. A flowchart of a CELP analyzer.

With this scheme, we can get all the CELP parameters for one frame of speech. The program sends the parameters to error protection and data arrangement blocks before returning to the EVM controller.

CELP Synthesizer

Figure 4.9 shows a flowchart of a CELP synthesizer corresponding to the scheme described in Chapter II. Similarly, this flowchart directly represents the synthesis scheme. The code book parameters available in an input buffer produce 60 excitation impulses, which updates the ACB and excites an LP filter to produce 60 samples of synthetic speech. The speech is stored in another working buffer.

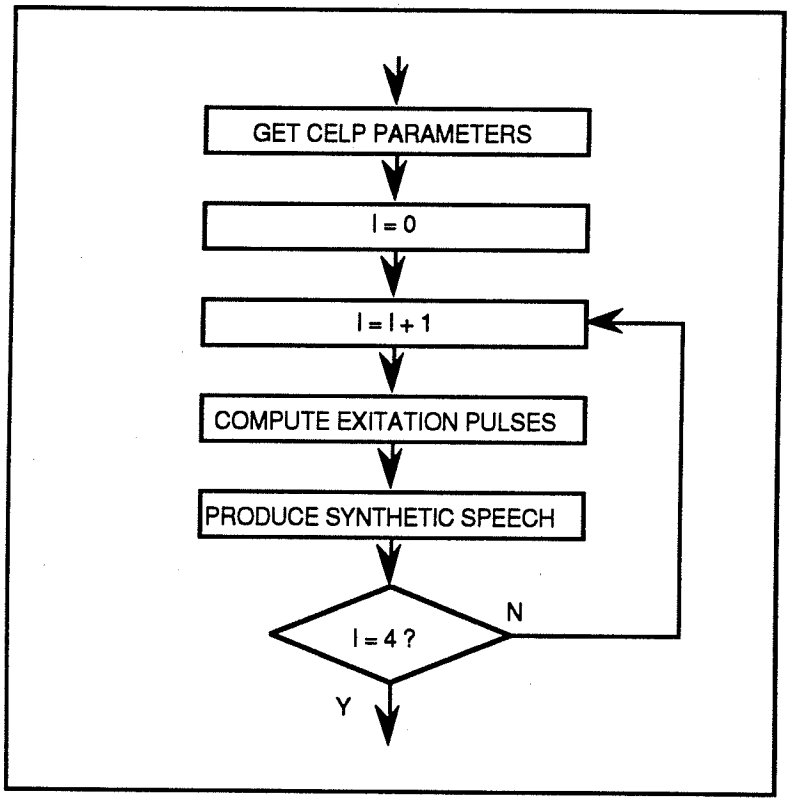


Fig. 4.9. A flowchart of a CELP synthesizer.

The process repeats three more times until 240 synthetic samples are available in the working buffer. The program then returns to the EVM controller, and the controller declares a successful operation.

Error Protection, Data Stream Arrangement, and Data Recovery

The system has three schemes of error protection embedded in the proposed FS-1016, i.e., (i) FEC (Hamming code), (ii) a stability rule, and (iii) data scrambling (see Section 2.7). During the transmission, the Hamming (15,11) code protects several bits of the CELP parameters, as shown in Fig. 4.10. The system then scrambles the CELP parameter bits before transferring them to the channel. The program returns to the EVM controller, which then declares a successful operation.

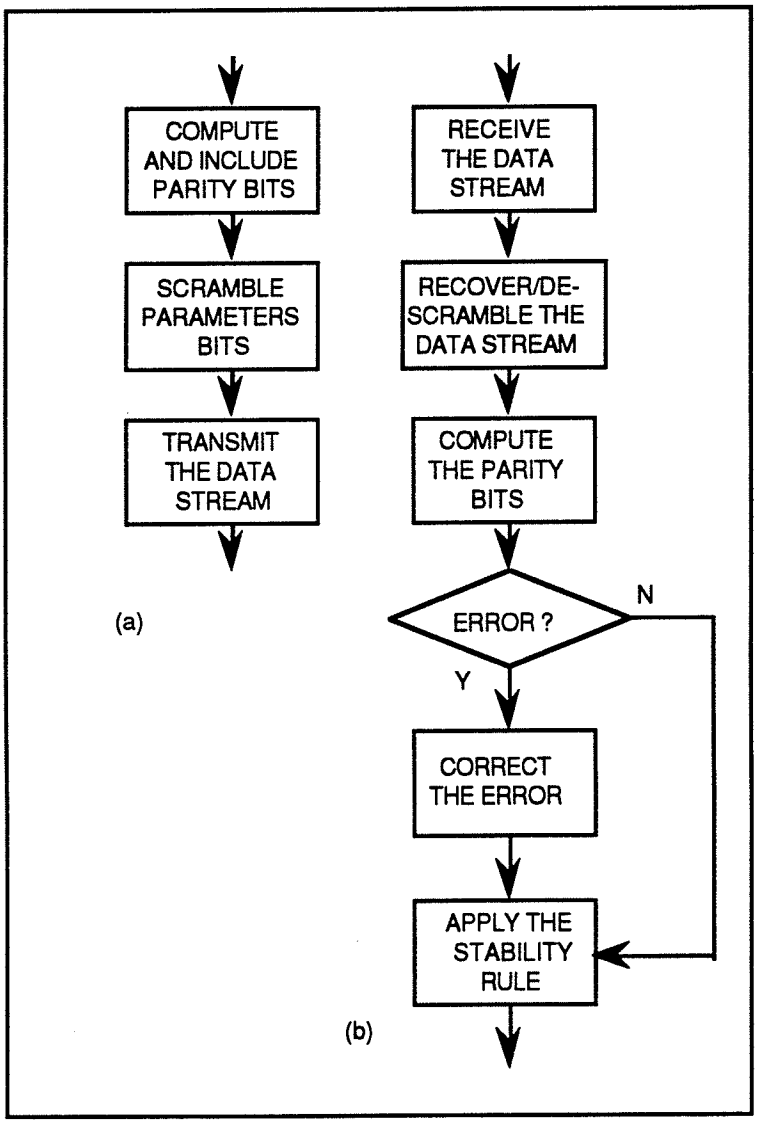


Fig. 4.10. Flowchart of the error protection and data stream schemes during (a) data transmission, and (b) data reception.

During the reception, the system first unscrambles the data stream. The Hamming code then detects and corrects errors within its range. We then apply the smoother algorithm.

At this stage, we only smooth the LSP parameters through the stability rule mentioned in section 2.7, and leave the other parameters as they are. This is because the pre-third draft of the proposed FS-1016 has not standardized the procedure of obtaining the smoothing threshold. Without the standard, an intensive experiment would be required to obtain the threshold before we can proceed with smoothing of the other parameters.

HOST-EVM Data Exchanges

A DMA controller of the TMS320C30 controls the host-EVM data exchanges. The host asks for a data exchange by sending one exchange command. Since the data can be 240 PCM samples or 9 CELP parameters, and the direction can be from or to the EVM, we have four types of data exchanges. The digital signal processor writes the starting address and the number of samples to be transferred to the DMA controller. The digital signal processor then grants the communication control to the DMA controller. The next access by the host interrupts the DMA controller instead of the digital signal processor until all the data have been transferred. Using a special interrupt, the DMA controller communicates the end of transmission to the digital signal processor which, at this time, takes back the communication control.

4.2.2.3 I/O Processor

As mentioned earlier, the I/O processor must perform data exchanges involving two sections, namely, the packet-radio channel and storage. The processor performs four different exchanges of CELP data (read/write to or from a channel/storage) and two different exchanges of PCM data (read/write to or from a storage).

Although the host controls the I/O processes, there are two processes that are not fully controlled: data receiving and sending from and to the RS232 port, with a similar reason with the A/D and D/A cases. They are interrupt driven processes. Since we reserve two buffers for both processes, CHANNEL IN BUFFER and CHANNEL OUT BUFFER, a program can acquire new data by simply reading the CHANNEL IN BUFFER. Similarly, for providing data to be sent, a program just writes the data to the CHANNEL OUT BUFFER.

Packet Radio Channel

The system communicates with the channel through the RS232C port because packet radio is usually connected to the RS232C port. Here, we assume the packet radio consists of a TNC and a radio (transceiver). The RS232C port is connected to the TNC while the TNC is connected to the radio. We also assume that the packet radio operates such that it transmits to a communication channel all the data it receives from the RS2322C port, and provides to the RS232C port all the data it receives from the channel. This implies that the communication protocol and packet radio initialization are performed by the packet radio or another program. From a programming point of view, this assumption simplifies data access to the channel. Writing or reading data to or from the channel is reduced to writing or reading data to or from the RS232C port, respectively.

During data reception, an I/O program acquires CELP parameters in the data stream form. As mentioned earlier, the program is an interrupt service routine that responds to periodic interrupts occurring every 1/600 s. This rate may vary depending on the channel rate. It also varies due to the asynchronous mode of the serial communication.

When an interrupt occurs, this program reads a byte of data from a data register of a serial communication adapter (*universal asynchronous receiver/transmitter*, UART) and puts the byte in the CHANNEL IN BUFFER at the location pointed to by the buffer pointer, as shown in Fig. 4.11. The pointer of the buffer is then incremented by one location. Since this buffer is a circular buffer, if the pointer is at the end of the buffer, the next location

must be the beginning of the buffer.

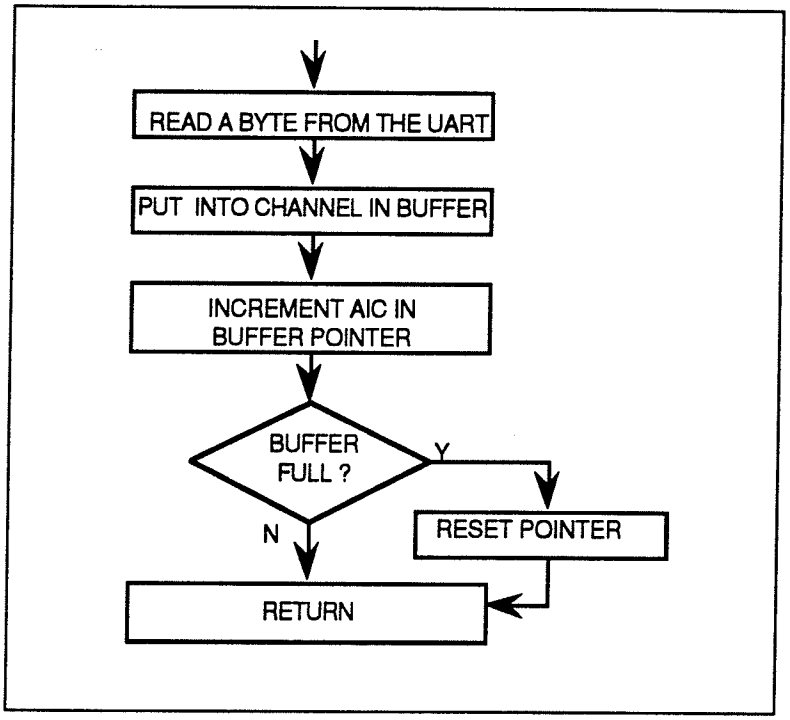


Fig. 4.11. A flowchart of the RS232C interrupt services. Here, the figure shows the data reception service only. The data transmission service is exactly the same except a byte is read from the CHANNEL OUT BUFFER and put on the data register of the UART.

Similar to the data acquisition process previously described, the data reception process must complete acquiring one block of data stream (18 bytes) before the reception reads the data from the CHANNEL IN BUFFER. If the pointer is pointing to the beginning of the buffer, one block of data has been acquired, thus the data can be read. It should be noted that this scheme also implies two things: (i) the pointer must be readable by all subroutines that need access to the CHANNEL IN BUFFER, and (ii) the data access from the buffer can be done faster than a period of an interrupt.

Data transmission inverses the reception process. An interrupt causes a process of loading data from the CHANNEL OUT BUFFER into a data register of the UART. Since the buffer is also a circular buffer, all pointer procedures mentioned above also apply.

We do not put both processes in the same interrupt service routine because the sources of the interrupts are different. They are different because the communication mode is asynchronous, thus the rate of both interrupts, especially for data reception, usually varies. In addition, this separation is very suitable for the half duplex operation, because here the data reception and transmission must be activated separately.

Speech Storage

We want to have a capability of storing or retrieving speech data in either PCM or CELP format. During storing (recording), the system writes the data available in the DISK BUFFER into a disk file. During the retrieval process (playback), the system reads data from a file to the DISK BUFFER. Thus, the data interchange between a buffer in the host memory and the host hard disk is the basic operation for speech storage. However, we distinguish between real-time and non real-time speech storing and retrieving due to the time critical nature of the real-time operation.

In real-time recording, all the speech data reside in the DISK BUFFER, that uses host memory. A program then stores all data into a file using standard *disk operating system* (DOS) procedures. In the real-time playback, all data from a file are put into the DISK BUFFER before being sent to the EVM or the channel. This scheme guarantees that there is no data loss during a real-time operation regardless of the access time of the storage device. However, this scheme introduces a limitation in the length of speech the system can record or playback to or from a file, due to the limitation of the buffer size on a PC-AT. As mentioned earlier, we provide a DISK BUFFER with a maximum size of 192,000 bytes to hold 12-s and 320-s of PCM and CELP data, respectively.

Non-real-time speech-storage only occurs during the conversion process between a CELP file and a PCM file. The system reads a block of speech data from a file and sends it to the EVM. The conversion results are stored in another file. The process is repeated for another block of speech data until the entire file has been converted. Thus, the size of the

memory buffer needed is very small, and the size of the storage device becomes the limit of the speech length the system can store.

The limitation in the real-time mode can be alleviated if we can install more memory on the EVM. The additional memory can then be used to build a swinging buffer [Fere91] that can hold the data during slow disk access. In this scheme, the limit of the speech length we can record or playback is the size of the storage device. However, this scheme may introduce longer coding delay.

4.3 System Implementation

Since the architectural and organizational designs reduce the design complexity, system implementation simply becomes converting all functions to computer code. Now, we are interested in an efficient development of efficient code. We thus describe the development before we briefly explain the code. The code details are provided in Appendix B.

4.3.1 System Development Method

The use of the PC AT and the EVM makes the system development much easier because both host computer and the EVM have powerful development tools. However, applying careful development methodology for realization is still critical due to several problems. First, fast CELP algorithms are not trivial, and involve many computational processes. Second, the EVM has a limited size of installed memory, resulting in a need to compact the program size and to schedule the use of working buffers. Third, the real-time requirement demands efficient code to achieve execution time as short as possible. Finally, since the TMS320C30 itself is a processor with pipelining and parallel execution, a higher complexity of software must be considered.

Unfortunately, solutions for the problems contradict each other. For example, we solve the software complexity addressed in the first and fourth problems by dividing it into

many small routines that are easy to develop and maintain. However, creating small routines always introduces overhead in memory and execution time needed, due to the routine transition and the need of local data. Since there can be many small routines, the overheads may grow unacceptably. The second and third problems limit the number of routines we can make. Furthermore, the solutions for the second and third problems, that pertain to spatial and temporal optimization, require techniques such as memory sharing and parallel type of execution. Those techniques usually increase the software complexity. Thus, solving a problem increases the difficulties of solving the other problem.

The following sections describe the development method used to solve the difficulties. System development begins with hardware setup, followed by software development, and finally system test.

4.3.1.1 Hardware Setup

The final system is a PC-AT equipped with the EVM, a two channel amplifier, a microphone, a speaker, a TNC, and a transceiver. For full-duplex implementation the TNC and the transceiver must be set accordingly. However, a smaller configuration can be used during the development process. A general purpose computer, such as SUN SPARC 2 workstation, can be used to simulate the algorithms. The SUN workstation is prefered because its fast processor can speed up the simulation process. In addition, its sound capabilities (record and playback) are very useful for immediate listening tests. A PC-AT is sufficient to code and test the EVM routines because the EVM development system has a TMS320C30 simulator that runs on a PC-AT. The PC-AT with the EVM installed uses the resulting EVM routines for real-time debugging.

4.3.1.2 Software Development Method

Software developments starts with algorithm simulations. The EVM and the host programs can use the simulation code. Since the programs are built modularly, they must

be integrated during and at the end of the development process. The integration results in a system prototype that can be used in system verification.

Simulation

We use C programs to simulate the algorithms on a SUN workstation. This simulation is essential because the algorithms are complex. It is difficult to develop routines of such algorithms directly in a complicated environment such as the TMS320C30 assembly language environment. The high level language simplifies the development of the logical structure of each routine. Here, the process of the functional blocks are translated into C routines. Thus, the resulting C program has a structure that represents the architecture and organization.

The use of the C programming language is very suitable because the language supports modular programming. The language also has a powerful developer that makes programming and debugging much easier. These features are important because intensive tests and re-programming of the routines take place. For example, translating the algorithms into code requires verification process. Optimizing the size and the execution time of the routines also require intensive tests and re-programming. Finally, the C routines are transportable into many different platforms, including the host computer and the EVM.

Actual Software Coding

After optimizing the C routines, they are re-coded one by one into the TMS320C30 and host codes. Since the simulation codes have been verified, the EVM and host programming just translates the original codes into the EVM and host codes without concerning the CELP algorithms anymore. Although this process may be tedious, the difficulties due to the algorithm complexity have disappeared and the process simply becomes mechanical. It should be noted that the EVM and the host subroutines are

developed separately because they use different tools. However, both development tools run on a PC.

EVM Routine Coding

There are two ways to convert a C module into its TMS320C30 counterpart. First, a C compiler, available from Texas Instruments, automatically transfers a C program into a TMS320C30 assembly language. Second, we can directly write a TMS320C30 assembly language module with the help of the logical structure and data type of the corresponding simulation program. Although we can have full control and very efficient codes using the latter approach, the former approach is more convenient. We utilize both by using the first scheme to obtain a draft of the code, and using the second scheme to compact the draft. However, for simple control and time critical processes, such as interrupt handling, we use the second scheme.

Each EVM routine is tested off line using a TMS320C30 simulator. Here, we can check the number of clocks and registers used. In addition, we can control several CPU operational options, such as pipelining, for easier debugging. The execution results of the routine are compared with its C counterpart for verification. A real-time debugger then tests the program in the actual processor. Here, we can fine tune the software and test the peripheral specific functions that are not fully supported by the simulator. Finally, we load the verified code into the EVM for normal operation or operational tests.

Host Routine Coding

The host program is a combination of C and 8086 assembly language. We use Microsoft Quick C and Quick Assembler as the software developer. Most of the simulation programs developed before can be used directly without major modifications. The additional codes are for hardware specific functions, such as data exchanges with the EVM and the serial port.

Integration

We gradually integrate all the routines into one operational program. The system integration starts with the controllers of both EVM and host as the main program. The command and data transfers between the host and the EVM are established. Here, we utilize generic communication programs provided by the EVM manufacturer. The integration tailors the functional blocks, one by one, into the program in both EVM and host. We verify every tailoring process. The complete prototype is then ready for testing and adjustment in real operation.

4.3.1.3 System Test and Adjustment

Although careful coding and tailoring eliminate many potential problems of realization, problems still occur during operational tests. Some of the operational problems come from functional failures when all functions, especially interrupts, are activated together. However, those are minor problems that can be fixed with given tools.

The more difficult problems relate to the program execution time. Here, all functions work well but the total execution time, including the overhead, goes beyond the allocated time. Although this is not a problem for non real-time applications, it causes unacceptable data loss in the real-time application. To solve this problem, further time optimization is required. Modules that require long execution time are adjusted. The adjustment may have to occur in the simulation program level, thus re-coding must take place. The adjustment must be repeated until the execution time is acceptable.

The development method described above is recommended in many digital signal processor applications. It helps the development process by handling the difficulties separately using appropriate tools. The codes resulting from the development process are described in the following section.

4.3.2 System Realization

Since the host computer, the EVM, and the audio system have implemented all necessary hardware, most of the system realization deals with software coding. The software for the host and the EVM are written in the C and 8086 assembly language, and the TMS320C30 assembly language, respectively. In the following section, we explain the codes starting with the host program and ending with the EVM program.

Host Controller

The controller is adapted from a generic controller program supplied by Texas Instruments (TI). The program starts by calling a subroutine `init_evm()` to reset the communication registers and initialize the size of the data block during the transfer process. A routine `display_menu()` displays a menu of user commands. The controller then enters a loop of accepting user commands and executing them. A routine `kbhit()` checks if the keyboard has been pressed. If a key has been pressed, the controller calls `command_process()` to execute the user command corresponding to the key pressed. The menu is displayed by `display_menu()` signalling that the process is executed successfully and the system is ready to accept the next command. The system then starts another new cycle in the loop.

However, during real-time data exchanges between the host and the EVM, the controller must keep on transferring data until all data have been exchanged. So, the controller checks the flags `play` and `record`. If `play` is set, a block of data is transferred to the EVM by a routine `playing()` even if no key is pressed. Accordingly, if `record` is set, the block of data is read from the EVM by `recording()`. The processes are done outside `command_process()` to allow a user to issue another command during a real-time data exchange process. However, until now the only command that can be issued during a real-time process is `STOP`, that is a command to stop

the exchange process.

It should be noted that the flag `record` is reset as soon as a data buffer used to hold the data is full. The flag `play` is reset by a `STOP` command only. If the buffer is full, the buffer pointer is reset. So, the real-time playback is always repeated until the user wants to stop it.

Routine `command_process()` reads a command using a routine `get_command()`. The received command is then parsed through a command table using a C control structure called `switch`. The controller can then start a corresponding command. If there is no command corresponding to the key pressed, the controller returns without doing anything.

If the command is one of the host-only processes, such as storing CELP parameters from the channel, or sending CELP parameters from a file to the channel, the controller starts the process without sending any command to the EVM. If the command involves processes in the EVM, the controller issues an EVM command using `send_command()`.

The routines `playing()` and `recording()` mentioned above are special cases of issuing EVM commands because the routines need a tight cooperation between the host and the EVM for fast data exchanges.

Host Serial Port Programs

Host serial port routines deal with data exchanges between the data register of the UART and the CHANNEL OUT BUFFER or the CHANNEL IN BUFFER. A routine `transmit_rs232()` puts data from the CHANNEL OUT BUFFER into the UART. Another routine `receive_rs232()` gets data from the UART, and puts the data into the CHANNEL IN BUFFER. These routines are interrupt services of the UART.

File Access Programs

File access routines save or retrieve data during speech storage functions. The C library has provided routines to save and retrieve disk data called `fwrite()` and

fread(). Both routines are used in every process that requires disk access. Since a disk file may contain either CELP or PCM data formats, we recommend the use of extensions .CEL and .PCM for filenames, respectively.

EVM Controller

The controller, that is also adapted from the generic program supplied by TI, has the same structure as the host controller. The controller starts with calling init_evm() to initialize the DMA controller and the serial port connected to the AIC. The controller then calls init_arrays() and init_aic() to prepare and initialize the data buffers and AIC. The controller starts a loop that contains a routine command_process(). This routine processes every command that comes from the host. One cycle of the command_process() execution starts with waiting for a DMA transfer to finish its process. This is intended to avoid a routine to access data in a buffer used by the DMA before the buffer is ready. The controller then reads the port where the host puts its commands. The command is parsed through an EVM command table. A valid command activates the corresponding routines. The controller then sends a message signalling a successful operation and returns to begin a new cycle.

During real-time operations such as data exchanges between the AIC and the host, the controller must wait for a complete block of data to be available in either the A/D BUFFER or the D/A BUFFER before the buffers can be accessed for either recording or playing data. To send a block of data in the buffer to the host, the controller invokes the routine recording(). On the other hand, to receive a block of data from the host, the controller invokes the routine playing(). Those routines are fast because they use DMA transfer. So, had the controller not waited for the completion of the AIC data transfers, the recording() would have sent the same data twice to the host, or the playing() would have overridden the data in the buffer before they would have been used by the AIC.

Analog Data Access

Analog data access consists of data acquisition and data distribution from and to the AIC, respectively. As mentioned in the previous chapter, both processes reside in the same routine `c_int05()` that is an interrupt service for the interrupts from the AIC serial port. The routine starts with the data distribution process. The routine reads a word of data from the D/A BUFFER. The data are left-shifted by two bits before being sent into the serial port connected to the AIC. The data are shifted because the AIC operates on 14 bits while the serial port operates on 16 bits.

The routine then reads data from the serial port that contains results of the A/D conversion and puts the data into the A/D BUFFER. The 14 bit data should be right-shifted by two bits to be aligned in the 16 bit world. However, since the TMS320C30 is a 32-bit processor, the right shifting is accomplished by left shifting by 16 bits and then right shifting by 18 bits. This ensures that the sign of the data is placed properly in the 32-bit world.

CELP Speech Analysis

The controller first puts a block of PCM data into a buffer and prepares another buffer to receive the results before starting the analysis (compression) process. The controller then calls `Analyze()` to begin compressing the PCM data.

The routine `Analyze()` starts with saving the addresses of input and output buffers and converting PCM data into floating point format. It then calls routines `AnalyzeLPC()` to obtain the LP parameters, as mentioned in Section 2.5.1. A routine `ExpandBandwidth()` alters the LP parameters before they are converted into LSP parameters using a routine `ConvertToLSP()`. The resulting LSP parameters are stored in the first ten locations of the output buffer after they are checked by `CheckLSPStability()`. The program calls `CodebookSearching()` to obtain all code book parameters. `ConvertToDataStream()` then protects the parameter bits and

scatters them according to FS-1016 protocol. A routine `ShiftTheOriginalSpeech()` prepares the searching target for the next frame and `UpdatePreviousLSP()` stores the LSP to be used by interpolation and stability checking of the next frame.

The code book searching begins with interpolation of the LSP parameters by `InterpolateLSP()` and conversion of the LSP to LP parameters by `ConvertLSPToLPC()`. To combine the LP filter and the perceptually weighting filter, we expand the LP parameter using `ExpandBandwidth()`. A routine `FindImpulseResponse()` converts the LP filter into a matrix form.

A routine called `FindACBSearchingTarget()` works on the first 60 PCM data from the input buffer to provide the searching target. A routine `ACBSearchingOdd()` uses the target to give the best ACB parameters. The parameters are stored into the output buffer. `FindSCBSearchingTarget()` uses the parameters to compute the SCB parameter searching target. `SCBSearching()` then obtains SCB parameters and stores them in the output buffer.

Next searching requires some preparations. The routine `UpdateACB()` generates excitation impulses and uses them to update the content of ACB. Finally, a routine `GetDelayElements()` tracks the delay elements used later for obtaining zero response in `FindACBSearchingTarget()`.

The program repeats all the code book searching procedures above three more times until all 240 samples have been used. The routines used are exactly the same except for the second and fourth subframes, where the `ACBSearchingEven()` replaces `ACBSearchingOdd()`. Routine `ACBSearchingEven()` works only on a subset of the ACB, thus reducing the computation time. `DeltaEncodingACB()` encodes the resulting ACB parameters using delta coder. At the end of the analysis process, there are 26 data in the output buffer representing all the CELP parameters of one frame, ready to be converted to a data stream.

CELP Speech Synthesizer

A routine `Synthesize()` decompresses a block of data into 240 speech samples. The routine calls `ConvertFromStream()` to get the LSP and the code book parameters from the data stream. `CheckLSPStability()` checks the stability of LSPs. The program then finds the corresponding PCM data for each subframe, beginning with interpolating the LSP and converting them to LP parameters, using `InterpolateLSP()` and `ConvertLSPToLPC()`.

A routine `UpdateACB` computes 60 excitation impulses and uses them to update the ACB. The impulses also excite an LP filter to generate the synthetic speech in routine `GetDelayElements()`. Since the results are in floating point numbers, `ConvertToPCM()` translates then to PCM form. The process is repeated three more times using the other code book parameters until all 240 PCM data are available in the output buffer.

Error Protection and Bit Scattering

Since Hamming (15,11) is a simple code, the implementation of its encoder can be done directly using several modulo-2 additions inside a bit scattering routine `ConvertToDataStream()`. The decoder is also embedded inside a descrambler routine `ConvertFromStream()`. The `ConvertToDataStream ()` converts the 26 CELP parameters into a stream of 18 data bytes. Conversely, the `ConvertFromStream()` converts the stream back into 26 CELP parameters. Another error protection scheme, stability rule, is done by a routine `CheckLSPStability()`. This routine checks the monotonic property of the incoming LSP parameters. If the LSP parameters fail the test, the previous LSP parameters replace them.

Benchmarking of the Codes

We were able to fit all the EVM code into less than 11,000 words of memory. Total execution time of the current version of the system is close to real-time (approximately 3 times), as shown in Table 4.5. Here, we use 512 and 128 sizes of SCB and ACB, respectively. The code is optimized in C, and only a small portion of codes (less than 5 %) is coded using assembly language. The linker informs the memory used is, the simulator and debugger approximate the execution clocks. It should be noted that since the debugger benchmarks the functions, the overhead of interrupt processes is not included because the interrupts are disabled during the simulation and debugging.

Table 4.5. One frame execution time of the system in current version. We use 512 and 128 sizes of SCB and ACB, respectively. Here, the codes are optimized in C.

Functions	Clocks	% of Real Time
`Analyze()`	1,978,998	220
`Synthesize()`	45,717	5
Both	2,024,715	225

Table 4.5 shows that the analysis process dominates the computation. The two inner products inside the joint optimization scheme consume 50.68% of analysis computations, as shown in Table 4.6. This consumption is very high considering that there are approximately 20 other processes in a CELP analysis, and the second most dominant process, that is finding the SCB searching target, needs only 2.56 %. Despite their very simple structure, the inner products need such a high computational power because they are called 384 times by ACB searching (due to 2 even and 2 odd subframes and 128 entries), plus 2058 times by SCB searching (due to 4 subframes and 512 entries).

However, real-time implementation is still possible because we have not coded all functions using assembly language. A function `FindSCBSearchingTarget()` gives a significant reduction in execution clock when it is coded using assembly language, as

shown in Table 4.7. Here the execution time was reduced by a factor of more than 3. However, further substantial reduction must rely on efficient inner-product computation.

Table 4.6. Domination of inner product computation.

Processes	Clocks	% of Analysis Computations	% of Total Computations	% of Real Time Computations
Inner Product	1,003,662	50.68	49.56	111.11
Joint Optimization	1,329,424	67.77	65.66	147.67
Code Book Searching	1,961,301	99.09	96.87	217.88
SCB Searching Target	50,704	2.56	2.50	5.55

Table 4.7. An example of the effect of assembly coding to reduce the execution time. Here, the execution time is reduced by a factor of more than 3.

Functions	Clocks		
	Unoptimized C	Optimized C	Assembler
`FindSCBSearching Target()`	45,264	12,676	3,410

Table 4.5 indicates that CELP coding is an asymmetric technique, i.e., the synthesizer is much simpler than the analyzer [Jaya90]. This type of technique is desirable in many applications such as one-way broadcasting communication and CD-ROM based speech systems.

4.4 Summary

A systematic design enables an implementation of a CELP system on a PC equipped with a TMS320C30 DSP EVM. Since the resources (memory and computational power) are limited, we identify minimum tasks that can be combined to perform all user requirements. This approach leads to a signal processor structure consisting of a controller (with its command set), data processor, and I/O processor. Furthermore, to reduce the

controller burden, we develop a CELP engine as data processor. Having identified tasks of all parts, we organize the task on the implementation platform, based on the characteristics of each tasks and the platform capabilities. The remaining work is then to convert every task into efficient hardware circuits or software codes.

We were able to put all codes without exceeding the memory limit (16 Kword on the EVM and 512 Kbyte on the PC). Our current system can perform all user requirements, except for those that use real-time CELP analysis. However, an observation indicates that further code optimization can result in a real-time CELP analysis.

CHAPTER V
EXPERIMENTAL RESULTS

The CELP system described in the last chapter has high speech-intelligibility and natural speech-quality. A preliminary test shows that the system achieves 95.67 % word intelligibility. A subjective listening gives a 3.21 mean opinion score (MOS), which is comparable with high bit-rate coders such as ADPCM and CVSD. Furthermore, an objective test shows a 10.10 dB segmental signal-to-noise ratio (SEGSNR), which is considered good compared to other published results.

Tests were performed mainly to verify the design. The tests are also useful for comparison with other coders. In addition, they can diagnose system weaknesses which can be used for future improvements.

5.1 Performance Measurements

5.1.1 Features for Measurements

As mentioned in Section 2.1, the features of speech compression include speech intelligibility and quality. The system must preserve the speech message. We want to measure the system's ability to preserve the meaning of speech. Furthermore, normal, public communication needs natural speech-quality. Thus, we are interested in measuring the difference between the sound of the synthetic CELP speech and that of the original, human speech.

Since these features are commonly used as criterion in speech compression, the standard procedures are available. Not only does it simplify the process, it also provides a common ground for coder comparison. In addition, diagnostic tests are also available.

5.1.2 Validity of the Measurements

To achieve test validity, we base our tests on standardized procedures and materials. For the intelligibility test, we use one of the original versions of a *rhyme* test introduced by Fairbanks [Fair58]. Here, we perform a word intelligibility test on a group of listeners. Since the test only covers isolated words, we add a sentence intelligibility test to observe the effect(s) of word context to the intelligibility. This type of speech message is more common in speech communication.

For the quality test, we perform subjective and objective tests. In the subjective test, a group of listeners judge the quality according to their personal preference. We employ a *Category-Judgment* technique described in the IEEE recommended practice on speech quality measurements [IEEE69]. The test materials are Harvard sentences, which are frequently used because they are phonetically balanced.

In the objective test, we employ a metric to the distance between the original and the synthetic waveforms. Here, we use a segmental SNR measurement similar to the technique used in waveform coding [JaNo84], [RoBa90].

Since this is a preliminary test, we relax some requirements of the guidelines. We use only a subset of the Harvard sentences and Fairbank utterances for subjective and intelligibility tests, respectively. From 720 Harvard utterances, we randomly select 20 sentences. Similarly, from 50 sets of Fairbanks utterances, we select 18 sets of utterances. However, the selection is not done randomly, instead the sets represent 18 different consonants that are predominant in English [Fair58].

We also use a limited number of listeners. The IEEE standard recommends 6 to 10 trained listeners or at least 50 untrained listeners. However, the IEEE standard makes clear that there is no standard number of listeners that should participate. The number may vary according to the test purposes. We thus arbitrarily use 10 listeners for our test. Although they are untrained listeners, they are university students to ensure their familiarity with the words used in the test.

Despite this simplification, the test results are still adequate for our purpose at this stage. We are more concerned with a verification of the quality of our CELP algorithms and system prototype than a deep analysis on the system behaviour from a speech quality and intelligibility perspective. Furthermore, a complete and formal test requires a very large utterance data base, a large number of participants, a standard room, and standard equipment. A common practice is to send the recorded samples to professional test laboratories such as Dynastat [KeST89].

5.1.3 Methods of Measurements

We perform four tests: (i) word intelligibility test, (ii) sentence intelligibility test, (iii) subjective quality test, and (iv) objective quality test. The following is a description of the measurements and procedure of the tests. The test materials are in Appendix C.

5.1.3.1 Word Intelligibility Test

In this test, a group of listeners must identify speech utterances produced by the CELP synthesizer. The utterances are the Fairbanks words.

The Measure

The measure of this test is the percentage of correct identifications of test utterances. The percentage is obtained as follows.

- For each utterance, listeners must choose one word of five possible words corresponding to the utterance produced from a tape.
- Since the correct word is known, the number of correct identifications can be computed for all utterances.
- The percentage of correct identification can be obtained by comparing the number of correct identifications and the number of test utterances.

We also measure the level of difficulty to identify the correct word. For every identification, the listeners should mark one of three levels: difficult, fairly easy, and easy (see Table 5.1). We then average the scores associated with the marked levels for all utterances and users.

Table 5.1. Difficulty levels and their scores.

Levels	Difficulties	Scores
E	Easy	3
DE	Fairly Easy	2
D	Difficult	1

Test Procedure

The following is a step-by-step procedure of our intelligibility test.

(a) Build the set of test utterances

From 50 Fairbanks sets of utterances, we select 18 sets of utterances. Each set contains one main word and four other words that are similar to the main word except for the first consonant. The first consonant of the main word represents one particular consonant of the 18 selected consonants. An example of the main word is `sale` representing consonant `/s/`, with its corresponding other words `male`, `pale`, `tale`, and `bale`.

(b) Prepare the questionaire

We set up a questionaire that must be filled by the listeners as shown in Appendix C. An example of one line in the questionaire is:

```
_male  _pale  _tale  _sale  _bale      D  DE  E
```

For each utterance, they have to mark the place corresponding to word they hear, and the difficulty level.

(c) Prepare the experiment tape

An english speaking male is asked to record the main word of each set to a source tape. Each speech utterance from the source tape is then converted into a digital PCM form using the CELP system. The system analyzes and then synthesizes the PCM data to produce a corresponding synthetic utterance. We record all synthetic utterances to another tape, called the test tape in an order according to the questioner.

(d) Conduct the listening test

Before the listening begins, we explain the procedure of the test. We adjust the loudness to a comfortable listening level. We then play the test tape, and the listeners fill the questionaire.

(e) Compile the data

The resulting data from the questionaires are used to compute the intelligibility and difficulty scores.

5.1.3.2 Sentence Intelligibility Test

In this test, a group of listeners must identify sentences produced by the CELP synthesizer. The utterances are the Harvard sentences.

The Measure

The measure of this test is the percentage of correct identifications of words in the sentences. The percentage is obtained as follows.

- For each utterance, listeners must write the all the words of the utterance

produced from a tape.

- Since the correct words are known, the number of correct identifications can be computed for all utterances.
- The percentage of correct identifications can be obtained by comparing the number of correct identifications and the number of all words in the test utterances.

Test Procedure

The following is a step-by-step procedure of our sentence intelligibility test.

(a) Build the set of test utterances

From the Harvard sentences used in the quality tests, explained in the next section, we arbitrarily select 10 utterances. The selected sentences from a male spoken set are sentences no. 4, 9, 10, 11, and 12. The selected sentences from a female spoken set are sentences no. 1, 8, 9, 10, and 12. An example of the sentences is

```
Slide the box into that empty space.
```

(b) Prepare the questionaire

For every utterance, we provide a blank space in the answering sheet.

(c) Prepare the experiment tape

An english speaking male and female are asked to record the 10 sentences (5 each) into the source tape. Each speech utterance from the tape is then converted into a digital PCM form using the CELP system. The system analyzes and then synthesizes the PCM data to produce a corresponding synthetic utterance. We record all synthetic utterances to the test tape according to the questioner.

(d) Conduct the listening test

Before the listening begins, we explain the procedure of the test. We adjust the loudness to a comfortable listening level. We then play the test tape and the listeners fill the questionaire.

(e) Compile the data

The resulting data from the questionaires are used to compute the sentence intelligibility.

5.1.3.3 Subjective Quality Test

In a subjective test, we measure subjective preferences of listeners regarding the speech utterances.

The Measure

The measure of this test is MOS. A higher MOS corresponds to a better quality. The MOS is obtained as follows.

- For each utterance, a listener gives a number from 1 to 5 according to his/her subjective preference of the natural quality of the utterance. The relationship between a preference and a number is tabulated as follows.

Preferences	Scores
EXCELLENT	5
GOOD	4
FAIR	3
POOR	2
UNSATISFACTORY	1

- Scores of all utterances from all listeners are added together and divided by the numbers of utterances and listeners. The result is MOS. We can then rate the system from UNSATISFACTORY to EXCELLENT by applying MOS to the above table.

Test Procedure

The following is a step-by-step procedure of our subjective quality test.

(a) Build the set of test utterances

From 720 Harvard sentences, we randomly select 20 sentences. A computer program is used to generate 20 random integer numbers uniformly distributed from 0 to 72. The first selected sentence corresponds to the first random number. We then skip a number of sentences to get the next sentence. The skip number corresponds to the next random number. This process is repeated until 20 sentences are obtained. The first 10 sentences are grouped into set M, and the rest 10 sentences are grouped to set F. An example of the sentences is

```
The red tape bound the smuggled food.
```

(b) Prepare the questionaire

For each utterance, listeners have to mark the place corresponding to their preference. One example is:

Test number 1

_EXCELLENT _GOOD _FAIR _POOR _UNSATISFACTORY

(c) Prepare the experiment tape

An english speaking male and female are asked to record the utterances in set M and

set F, respectively, to a source tape. Each speech utterance from the tape is then converted into a digital PCM form using the CELP system. The system analyzes and then synthesizes the PCM data to produce a corresponding synthetic utterance. We record all synthetic utterances to the test tape according to the questioner.

(d) Conduct the listening test

Before the listening begins, we explain the procedure of the test. We adjust the loudness to a comfortable listening level. We then play the test tape and the listeners fill the questionaire.

(e) Compile the data

The resulting data from the questionaires are used to compute MOS for (i) set M, (ii) set F, and (iii) both.

5.1.3.3 Objective Quality Test

In an objective test, we measure the Euclidean distance between the original and the synthetic speech. The distance is implied by a ratio of the energy of the original and the synthetic speech, called signal-to-noise ratio (SNR).

The Measure

The measure of this test is SEGSNR of the synthetic speech. A high SEGSNR usually means a high quality. The SEGSNR is obtained as follows.

- For each block of 240 samples, a *block* SNR (in dB) is obtained by dividing the energy of the original signal with the energy of the error between the original speech and the synthetic speech.
- The segmental SNR is obtained by averaging all the block SNRs of all utterances, excluding the SNR from blocks which have energy less than –52 dB

of maximum possible speech energy. (52 dB is the maximum noise of the test equipment).

Test Procedure

The following is a step-by-step procedure of our subjective quality test.

(a) Build the set of test utterances

We use all utterances from set M and F to be the test utterances.

(b) Error measurement

The PCM form of both original and synthetic data are used to compute block SNR.

(c) Compile the data

We calculate SEGSNR by averaging the block SNR of (i) set M, (ii) set F, and (iii) both.

5.2 System Performance

System Intelligibility

The results show that the system has high word intelligibility. Based on results from 11 listeners, the average of correct identifications is **17.2** words (out of 18 words) or **95.67 %**. The difficulty score is **2.41**. Thus, words can be identified quite easily.

Table 5.2 and Table 5.3 indicate that the system is more sensitive to alveolar consonants (consonants with alveolar as the place of articulation) [O'Sha87]. Table 5.3 shows that failures or difficult consonant (difficulty score less than 2.00) are of that type.

One possible cause is the removal of essential information by anti-aliasing filtering of the speech during the digitizing. Indeed, alveolar consonants contain mainly high-frequency energy [O'Sha87]. A listening to the original PCM speech supports this

possibility. Another possible cause is that the SCB employs Gaussian noise as its prototypes. Although this noise is suitable for voiced articulation, it may fail to reconstruct some stops and fricatives, because their residues are not necessarily Gaussian.

Table 5.2 Percentage of correct identifications of every consonant.

Consonants	Words	Correct Identifications		Difficulty Scores
		Average	%	
/l/	law	16	88.89	1.36
/t/	top	16	88.89	2.27
/s/	sale	16	88.89	2.55
/w/	went	17	94.44	2.36
/b/	back	17	94.44	2.72
/d/	day	18	100	1.81
/z/	zeal	18	100	1.91
/j/	yet	18	100	2.09
/n/	not	18	100	2.18
/m/	moon	18	100	2.36
/h/	heel	18	100	2.36
/v/	vile	18	100	2.55
/p/	page	18	100	2.72
/f/	fight	18	100	2.82
/k/	kill	18	100	2.82
/g/	god	18	100	2.82
/r/	rock	18	100	2.82
/dz/	just	18	100	2.91

Sentence intelligibility test confirms the above results. From 693 words tested, the listeners mis-identify 30 words resulting in (surprisingly) **95.67 %** intelligibility. Furthermore, the first consonants of the mis-identified words are also related to the results of word intelligibility (see Table 5.4). However, the labial articulations become more sensitive here.

System Quality

The system has natural quality, and is comparable to ADPCM [Klim87], CVSD [Klim87], and other CELP coders [RoBa90], [KlKK90] (see Table 5.5 and 5.6). It should be noted that MOS values from [Klim87] come from different listeners and different

listening rooms. However, they are still comparable because we use a similar procedure (category-judgement technique) and materials (Harvard sentences).

Table 5.3. Articulation types of 7 most sensitive consonants.

Consonants	Words	Place of Articulation
/l/	law	alveolar
/t/	top	alveolar
/s/	sale	alveolar strident
/b/	back	labial
/w/	went	back roundec
/d/	day	alveolar
/z/	zeal	alveolar strident

Table 5.4. Seven first-consonants of most misclassified words from sentence intelligibility test. The frequency of mis-classifications determines the ranking.

Consonants	Frequencies	Place of Articulation
/s/	5	alveolar strident
/l/	4	alveolar
/d/	4	alveolar
/w/	4	back rounded
/f/	4	labio dental
/b/	3	labial
/m/	3	labial

It is interesting to observe that male spoken speech performs better than female spoken speech. This may be caused by the coarseness of the ACB pitch resolution on high pitch frequency. Using Eq. (2.4), resolution for the highest pitch frequency (ACB address

of 20) is 19 Hz, while resolution for the lowest pitch frequency (ACB address of 147) is 0.37 Hz. Since female voice usually contains higher pitch frequency, the chance that ACB provides incorrect pitch for female speech is higher. This results in rougher synthetic speech, thus degrading the quality.

Table 5.5. Subjective quality of the system.

Coders	MOS
Our CELP	3.21
Our CELP (Male)	3.37
Our CELP (Female)	3.04
32 kbit/s ADPCM [Klim87]	3.73
16 kbit/s CVSD [Klim87]	2.67

Table 5.6. Objective quality of the system.

Coders	SEGSNR (dB)
Our CELP	10.10
Our CELP (Male)	10.27
Our CELP (Female)	9.94
CELP1 [RoBa90]	10.45
CELP2 [KlKK90]	8.57

5.3 Summary

The system is suitable for speech communication and storage application because it has high intelligibility and natural quality. Preliminary results, based on a rhyme and category-judgment tests, show that the system is comparable to other high quality coders.

We notice some difficulties in alveolar and labial types of consonants. Words with

those consonants have less intelligibility scores (88% and up). The telephone-bandwidth filtering may have caused significant loss in the high frequency components of the waveforms of the consonants, removing most of their energy. In addition, the SCB was design for voiced (especially vowel) articulations, thus performing some consonants may be more difficult. The system also performs male voice better than female voice due to coarseness of pitch resolution of the integer address ACB.

CHAPTER VI
CONCLUSIONS AND RECOMMENDATIONS

As a combination os several speech coders, the CELP technique can compress telephone-quality speech from 64 kbit/s down to 4.8 kbit/s (compression ratio of 13:1) while preserving natural speech quality. The quality is preserved because CELP uses waveform similarity as its compression criterion in a perceptually weighted Euclidean sense. To achieve the low bit-rate of 4.8 kbit/s, it uses an entry of a code book (SCB) to represent speech. The compression is possible because an information extraction of the target waveform of the SCB reduces the code book size, thus reducing the number of bits to represent the entry.

The information extracted, which is pitch and vocal tract information, should also be transmitted. The pitch information is implied by ACB parameters, which are quantized using a delta coding. The LP parameters represent the vocal tract information. The LSP scheme efficiently quantizes the LP parameters such that the vocal tract information requires only a 1.133 kbit/s rate. With this efficient representation, all parameters can be transmitted with a rate of 4.8 kbit/s. There are even available several bits for error correction, synchronization, and future enhancements.

An error correction strategy is based on parameter smoothing and a stability rule increases the robustness of the coding because they reduce the sensitivity of the reconstructed speech to perceptually disturbing errors, such as squeaks, blasts, and roughness. A Hamming (15,11) code and data scattering also enhance the robustness.

The high performance of CELP is achieved at the expense of a very high computational demand. In the current analysis scheme, the waveform similarity criterion requires exhaustive, closed-loop searching on two code books, SCB and ACB. This closed-loop searching requires a very high processing power, and consequently, fast

algorithms and processors are essential for a practical implementation.

We have combined several fast algorithms to reduce the computation, especially for code book searching. The algorithms include restructuring the searching scheme, serial searching, a joint optimization scheme and a special SCB with sparse, ternary, and overlapping prototypes. The fast algorithms can reduce the computation of code book searching by a factor of more than 175,000, thus reducing the speed to approximately 14 MIPS for real-time searching.

The CELP coding has been implemented on a low-cost PC-AT, equipped with the TMS320C30 EVM. The system is compatible with the proposed U. S. Federal Standard 1016 CELP system.

Since the fast algorithms use more memory space due to many look-up tables needed during the compression and quantization, and the variability of user commands also increases the system complexity, a signal processor architecture provides a solution to the system complexity and resource limitation. The system consists of a controller for command management, a fast CELP engine for CELP speech processing, and an I/O processor for speech transmission and storage. Based on the characteristics of the subsystems and the capability of the platform, the controller and I/O processor are implemented on the PC-AT, while the CELP engine is implemented on the EVM. The CELP engine is efficiently coded to fit within 16 Kword available memory. With this design approach, it has been shown that a real-time implementation is feasible on such a platform.

Despite the use of the fast algorithms, the code book search still dominates the computation with 99% of analysis computation. We have predicted that the inner products inside the joint optimization would dominate the computation by approximately 75% of total computations in a real-time system (see Chapter III). Indeed, in present system the bottleneck is the inner products with 50% of total computation, comparing to the next most demanding process with only 2.5%.

However a real-time implementation is still possible because most of the EVM codes are not in assembly language. Coding a portion of the code book search in assembly language can reduce the execution time of the portion by a factor of more than 3.

Preliminary experiments using rhyme and category-judgement tests show that the designed CELP system has high intelligibility (95.67% word correct identification), and natural quality. With a MOS of 3.21, the system is comparable to high-quality ADPCM and CVSD coders previously developed. Furthermore, with a SEGSNR of 10.10 dB, the system performs well in comparison with the results obtained in other published work.

In the experiments, it was noticed that some words with alveolar and labial consonants are more sensitive to be misclassified. Since such consonants have most of their energy in the high frequency range, it is suggested that the filtering during PCM digitizing has reduced their intelligibility. Another possibility is the code book may not be very well suited for those consonants.

It was also observed that female voice has less speech reproduction quality. Since the integer entry ACB has coarse resolution on high pitch frequency, while female voices usually have high pitch frequency, the ACB may have produced incorrect pitch, thus reducing the quality.

In summary, this thesis has contributed to technical knowledge through:

- An explanation of CELP coding from a data compression perspective. We explain the CELP coding based on a compression criterion and variability reduction of the code book. Prior to this work, the usual approach has been to explain the coding from a perspective of speech synthesis. The present approach puts CELP coding into a more general framework, thus opening possibilities of applying knowledge and tools that are available in the framework.
- Identification of some most sensitive consonants that affect speech intelligibility of a CELP system. This information is useful for characterizing CELP coders, and future improvement.

- A systematic design of a CELP system on a resource limited platform. This approach is also useful to design complex and computationally extensive DSP system.
- Identification of the most demanding process in current CELP coding, that is the inner product inside the joint optimization scheme. With this identification, future improvements should focus on fast hardware or algorithm of this process.
- Implementation of a compression system for high-quality speech, that is useful for low bit-rate speech communication and efficient storage. The applications include multimedia, speech recorder, voice mail, rural communication system, and speech recognition. The work described here also confirms the feasibility of utilizing a single digital signal processor system to implement a CELP system.

Future work could include the following improvements:

- The assembly coding of the EVM software should continue to achieve real-time CELP analysis.
- A non-integer ACB may be used to improve the pitch resolution for high pitch-frequency voices, as suggested by FS-1016. However, an efficient implementation is needed because the non-integer ACB requires more computation.
- One also needs a fast inner-product processor or algorithm to reduce the computation burden of the DSP. With this improvement, a slower and low-cost DSP may achieve a real-time implementation.

REFERENCES

[ADMD87] J. -P. Adoul, P. Mabilleau, M. Delprat, and S. Morisette, "Fast CELP coding based on algebraic codes," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Dallas, TX), CH2396-0/87, pp. 1957-1960, 1987.

[MiAh87] M. J. Miller and S. V. Ahamed, *Digital Transmission Systems and networks, Vol I.* Rockville, MD: Computer Science Press, 274 pp., 1987

[Alle73] J. Allen, "Speech synthesis from unrestricted text," in [FlRa73].

[Atal82] B. S. Atal, "Predictive coding of speech at low bit rates," *IEEE Trans. Commun.*, vol. COM-30, no. 4, pp.600-614, April 1982.

[Atal86] B. S. Atal, "High quality speech at low bit-rates: Multi-pulse and stochastically excited linear predicitve coders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2243-4/86, pp. 1681-1684, 1986.

[Atal89] B. S. Atal, "A model of LPC excitation in terms of eigenvectors of the autocorrelation matrix of the impulse response of the LPC filter," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2673-2/89, pp. 45-48, 1986.

[AtCK89] B. S. Atal, R. V. Cox, and P. Kroon, "Spectral quantization and interpolation for CELP coders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 69-72, 1989.

[AtHa71] B. S. Atal and S. L. Hanauer, "Speech analysis and synthesis by linear prediction of speech wave," *J. Acoust. Soc. Amer.*, vol. 50, pp. 637-655, 1971.

[AtRe82] B. S. Atal and J. R. Remde, "A new model of LPC excitation for producing natural-sounding speech at low bit-rates," in *Proc. IEEE Int. Conf.*

Acoust., Speech, Signal Processing, (Paris, France), IEEE CH1746-7/82, pp. 614-617, 1982.

[AtSc84] B. S. Atal and M. R. Schroeder, "Stochastic coding of speech signals at vey low bit-rates," *Proc. Int. Conf. Commun.*, part 2, pp. 1610-1613, 1982.

[BGGM80] A. Buzo, A. H. Gray, Jr., R. M. Gray, and J. D. Markel, "Speech coding based upon vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-28, pp. 562-574, October 1980.

[BrBH89] K. Bryden, A. Brind'Amour, and H. Hassanein, "Implementation of an 8000 BPS CELP coder on a single TMS320C25 digital signal processing," *Proc. Pacific Rim Conf. on Commun., Computer and Signal Proc.*, IEEE CH2691-4/89, pp. 384-387, 1989.

[CaHG90] F. J. Casajus-Quiros, L. A. Hernandez-Gomes, and C. Garcia-Mateo, "Analysis and quantization procedures for a real-time implementation of a kb/s CELP coder," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Albuquerque, NM), IEEE CH2847-2/90, pp. 189-192, 1990.

[CaWT89] J. P. Campbell, Jr., V. C. Welch, and T. E. Tremain, "An expandable error-protected 4800 bps CELP coder (U.S. Federal Standard 4800 bps voice coder)," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 735-738, 1989.

[CaTW90] J. P. Campbell, Jr., T. E. Tremain, and V. C. Welch, "The proposed Federal Standard 1016 4800 bps voice coder: CELP," *Speech Technology*, pp. 56-64, Apr./May 1990.

[ChGe87] J. H. Chen and A. Gersho, "Real-time vector APC speech coding at 4800 b/s with adaptive postfiltering," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Dallas, TX), IEEE CH2396-0/87, pp. 2185-2188, 1987.

[CoKKT89] R. V. Cox, Jr., W. B. Kleijn, and P. Kroon, "Robust CELP coders for noisy backgrounds and noisy channels," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 739-742, 1989.

[CoSe86] M. Copperi, and D. Sereno, "CELP coding for high-quality speech at 8 kbit/s," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2243-4/86, pp. 1685-1688, 1986.

[Croc81] R. E. Chrociere, "Sub-band coding," *Bell Syst. Tech. J.*, pp. 1633-1654, September, 1981.

[DaGe86] G. Davidson, and A. Gersho, "Complexity reduction methods for vector excitation coding," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2243-4/86, pp. 3055-3058, 1986.

[DaGe88] G. Davidson, and A. Gersho, "Multiple-stage vector excitation coding of speech waveforms," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (New York, NY), IEEE CH2561-9/88, pp. 3055-3058, 1988.

[DeLH88] D. J. DeFatta, J. G. Lucas, and W. S. Hodgkiss, *Digital signal processing: A system design approach.* New York: John Wiley & Sons, 661 pp., 1988.

[Dudl39] H. Dudley, "The vocoder," *Bell Lab. Rec.*, vol. 17, pp. 122-126, 1939.

[DuRW39] H. Dudley, R. R. Riez, and S. S. A. Watkins, "A synthetic speaker," *Journ. Franklin Inst.*, no. 227, pp. 739-764, 1939.

[EsGa77] D. Esteban and C. Galand, "Application of quadrature mirror filters to split band voice coding scheme," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 191-195, 1977.

[EsRa91] M. B. Esa and S. N. S. A. Rahman, "Technological development in malaysian rural environment," *Proc. IEEE Conf. Computer, Power, and Communication system*, Regina, SK; May 29-30, 1991), IEEE

91CH2927-2, pp. 252-259, 1991.

[Fair58] G. Fairbanks, "Test of phonemic differentiation: the rhyme test," *J. Acoust. Soc. Amer.*, vol. 30, no. 7, pp. 596-600, July 1958.

[Fere91] K. Ferens, "A speech splicing system for vocal shaping," M.Sc. thesis, The University of Manitoba, Winnipeg, MB, Canada, 172 pp, 1991.

[FlGo66] J. L. Flanagan and R. M. Golden, "Phase vocoder," *Bell Syst. Tech. J.*, vol 45, pp. 1493-1509, November 1966.

[Flan72] J. L. Flanagan, "Voices of men and machines," *Journal of the Acoustical Society of America*, Vol. 51, pp.1375-1387, 1972.

[FlRa73] J. L. Flanagan and L. R. Rabiner, *Speech Synthesis*. Stroudsburg: Dowden, Hutchinson & Ross, 511 pp., 1973.

[Fran69] L. E. Franks, *Signal Theory*. New Jersey: Prentice-Hall Inc., 317 pp., 1969.

[GoBM81] B. Gold, P. E. Blankenship, and R. J. McAuley, "New applications of channel vocoders," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-29, pp. 13-23, February 1981.

[GrMa76] A. H. Gray, Jr. and J. D. Markel, "Quantization and bit allocation in speech processing," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-24, pp. 459-473, December 1976.

[Hahe90] R. Hagen and P. Hedelin, "Low bit-rate spectral coding in CELP, a new LSP method," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Albuquerque, NM), IEEE CH2847-2/90, pp. 189-192, 1990.

[HiPe87] F. J. Hill and G. R. Peterson, *Digital System: Hardware Organization and Design*. New York, NY: John Wiley & Sons, 601 pp., 1987.

[IEEE69] "IEEE Sandard Publication No. 297: IEEE recommended practice for speech quality measurements," *IEEE Trans. Audio and Electroacous.*, pp.225-246, September 1969.

[InLK91] A. Indrayanto, A. Langi and W. Kinsner, "A neural network mapper for stochastic code book parameter encoding in cod-excited linear predictive speech processing," *Proc. IEEE Conf. Computer, Power, and Communication system*, IEEE 91CH2927-2, pp. 221-224, 1991.

[ItSa68] F. Itakura and S. Saito, "Analysis synthesis telephony based on the maximum likelihood," *Proc. 6th Int. Congr. Acoustics*, paper C-5-5, pp. C17-20, 1968.

[ItSa70] F. Itakura and S. Saito, "A statistical method for estimation of speech spectral density and formant frequency," *Elect. Commun. Japan*, vol 53-A, pp.36-43, 1970.

[JaNo84] N. S. Jayant and P. Noll, *Digital coding of waveforms: Principles and applications to speech and video*. New Jersey: Prentice-Hall, 688 pp., 1984.

[JaRa72] N. S. Jayant and L. R. Rabiner, "The application of dither to the quabtization of speech signals," *Bell Syst. Tech. J.*, pp.1293-1304, July-Aug. 1972.

[Jaya73] N. S. Jayant, "Adaptive quantization with a one word memory," *Bell System Tech. Journal*, pp.1119-1144, September 1973.

[Jaya74] N. S. Jayant, "Digital coding os speech waveforms: PCM, DPCM, and DM quantizers," *Proc. IEEE*, vol. 62, no. 5, pp.611-632, May 1974.

[Jaya90] N. S. Jayant, "High-quality coding of telephone speech and wideband audio," *IEEE Comm. Magazine*, pp.10-19, January 1990.

[JoMc90] P. Karn, "Txas packet radio society projects: An update," *Proc. 9th ARRL/CRRL Computer and Networking Conf.*, London, ON, pp. 122-

126, 1990.

[KaRa86] P. Kabal, and R. P. Ramachandran, "The computation of line spectral frequencies using Chebyshev polynomials," *IEEE Trans. ASSP.*, vol. ASSP-34, no. 6, pp. 1419-1426, December 1986.

[Karn89] P. Karn, "Amateur TCP/IP in 1989," *Proc. 8th ARRL/CRRL Computer and Networking Conf.*, Colorado Spring, CO, pp. 114-118, 1989.

[KeST89] D. P. Kemp, R. A. Sueda, and T. E. Tremain, "An evaluation of 4800 bps voice coders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 200-203, 1989.

[Kins89a] W. Kinsner, "Speech recording and synthesis using digital signal processing," *CRRL Convention*, Winnipeg, MB, Canada, 21 pp., August 1989.

[Kins89b] W. Kinsner, "Digital speech over packet radio," *CRRL Convention*, Winnipeg, MB, Canada, 16 pp., August 1989.

[Kins90] W. Kinsner, "Digital system organization," *Lecture notes*, University of Manitoba, 1990.

[Kins91] W. Kinsner, "Review of data compression methods, including Shannon-Fano, Huffman, aritmetic, Storer, Lempel-Ziv-Welch, fractal, neural network, and wavelet algorithms," *Technical Report*, DEL91-1, University of Manitoba, 157 pp., Jan. 1991.

[Klim87] G. A. Klimenko, "A study of ADPCM, CVSD, and phoneme speech coding techniques," M.Sc. thesis, The University of Manitoba, Winnipeg, MB, Canada, 228 pp, 1987.

[KlKi87] G. A. Klimenko and W. Kinsner, "A study of CVSD, ADPCM, and PSS speech coding techniques," *Proc. 9th. Conf. on Engin. in Medicine and Biology Society*, IEEE CH2513-0/87, pp. 1797-1798, 1987.

[KlKK88] W. B. Kleijn, D. J. Krasinski, and R. H. Ketchum, "Improved speech quality and efficient vector quantization in SELP," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (New York, NY), IEEE CH2561-9/88, pp. 155-158, 1988.

[KlKK90] W. B. Kleijn, D. J. Krasinski, and R. H. Ketchum, "Fast methods for the CELP speech coding algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*,vol. 38, no. 8., pp.1330-1342, August 1990.

[KrAt87] P. Kroon and B. S. Atal, "Quantization procedures for the excitation in CELP coders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Dallas, TX), CH2396-0/87, pp. 1649-1652, 1987.

[Krey78] E. Kreyzig, *Introductory Functional Analysis with Applications.* New York, NY: John Willey & Sons, 690 pp., 1978.

[KuFT91] S. Kumar, L. E. Fisher, and T. Truman, "A thin route digital radio," *Proc. IEEE Conf. Computer, Power, and Communication system*, Regina, SK; May 29-30, 1991), IEEE 91CH2927-2, pp. 316-320, 1991.

[LaIK91] A. Langi, A. Indrayanto, and W. Kinsner, "Stochastic code book searching using neural network for code-excited linear predictive coding," *Proc. IASTED Int. Conf. Computer, Elec., Control, and Communication*, Calgary, 1991.

[LaKi90a] A. Langi and W. Kinsner, "CELP high-quality speech processing system for packet radio transmission and networking," *Proc. 9th ARRL/CRRL Computer and Networking Conf.*, London, ON, pp. 164-169, 1990.

[LaKi90b] A. Langi and W. Kinsner, "A hardware description and logic cell array approach in development of a synchronous timing logic unit for a TMS3210-based real-time speech processing system," *Proc. 16th Canadian Medical and Biological Eng. Soc. Conf.*, Winnipeg, MB, pp. 95-96, 1990.

[LaKi91a] A. Langi and W. Kinsner, "Code-excited linear predictive speech processing for digital transmission and storage," *Proc. IEEE Conf. Computer, Power, and Communication system*, IEEE 91CH2927-2, pp. 205-209, 1991.

[LaKi91b] A. Langi and W. Kinsner, "Design and implementation of CELP speech processing system using TMS320C30," *Proc. 10th ARRL/CRRL Computer and Networking Conf.*, San Jose, CA, pp. 87-93, 1990.

[Lawr53] W. Lawrence, "The synthesis of speech from signals which have a low information rate," *Communication Theory*, Butterworth & Co, pp. 460-469, 1953 (reprinted in [FlRa73]).

[LiBG80] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for quantizer design," *IEEE Trans. Commun.*, Vol. COM-28, pp. 84-95, January 1980.

[LiRo89] Y. J. Lin and J. Rothweiler, "A high quality speech coder at 400 BPS," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 204-206, 1989.

[Mall89] S. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. 11, no 7, pp. 84-95, July 1989.

[Mari89] J. Mariani, "Recent advances in speech processing," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 429-440, 1989.

[Mark72] J. D. Markel, "Digital inverse filtering- a new tool for formant trajectory estimation," *IEEE Trans. Audio Electroacoust.*,vol. AU-20, pp.129-137, 1972.

[MoHo87] T. Moriya, and M. Honda, "Transform coding of speech with weighted vector quantization," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Dallas, TX), IEEE CH2396-0/87, pp. 1629-1640, 1987.

[O'Sha87] D. O'Shaughnessy, *Speech Communication: Human and Machine.* Reading: Addison-Wesley, 568 pp., 1987.

[Oyam85] G. Oyama, "A stochastic model of excitation source for linear prediction speech analysis-synthesis," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tampa, FL), IEEE CH2118-8/85, pp. 941-944, 1985.

[Pars86] T. W. Parsons, *Voice and Speech Processing.* New York: McGraw-Hill, 402 pp., 1986.

[PiDo89] J. Picone and G. R. Doddingson, "A phonetic vocoder," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 580-583, 1989.

[Proa83] J. G. Proakis, *Digital Communication.* New York: McGraw-Hill, 608 pp., 1983.

[Rabi68] L. R. Rabiner, "Digital formant synthesizer for speech-synthesis studies," *J. Acoust. Soc. Amer.*, vol. 43, pp. 822-828, 1968.

[RaSc78] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals.* Englewood Cliffs, NJ: Prentice-Hall, 512 pp., 1978.

[RJSC71] L. B. Rabiner, L. B. Jackson, R. W. Schafer, and C. H. Cober, "A hardware realization of a digital formant speech synthesizer," *IEEE Trans. Commun. Technol.*,vol. COM-19, pp.1016-1020, 1971.

[RoBa90] R. C. Rose and T. P. Barnwell III, "Design and performance of an analysis-by-synthesis class of predictive speech coders," *IEEE Trans. Acoust., Speech, Signal Processing*,vol. ASSP-38, no. 9., pp.1489-1503, September 1990.

[RTWC89] D. J. Rahikka, T. E. Tremain, V. C. Welch, and J. P. Campbell, Jr., "CELP coding for land mobile radio applications," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Glasgow, Scotland), IEEE CH2673-2/89, pp. 465-468, 1989.

[Smit57] B. Smith, "Instantaneous companding of quantized signals," *Bell Syst. Tech. J.*, pp.653-709, May 1957.

[ScAt85] M. R. Schroeder and B. S. Atal, "Code-Excited Linear Prediction (CELP): high quality speech at very low bit rates," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tampa, FL), IEEE CH2118-8/85, vol. 1, pp. 937-940, 1985.

[Schr66] M. R. Schroeder, "Vocoders: analysis and synthesis of speech," in *Proc. IEEE*, vo. 54, pp. 720-734, 1984.

[ScRa73] R. W. Schafer and L. R. Rabiner, "Design and simulation of a speech analysis-synthesis system based on short-time Fourier analysis," *IEEE Trans. Audio Electroacoust.*,vol. AU-21, no. 3., pp.165-174, June 1973.

[Stew22] J. Q. Steward, "An electrical analog of the vocal organs," *Nature (Macmillan Journal)*, vol. 110, pp. 311-312, 1922.

[SoJu84] F. K. Soong and B.-H. Juang, "Line Spectrum pair (LSP) and speech data compression," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (San Diego, CA), IEEE CH1945-5/84, pp. 1.10.1-1.10.4, 1984.

[Swan87] C. Swanson, "A study and implementation of real-time linear predictive coding of speech," M.Sc. thesis, The University of Manitoba, Winnipeg, MB, Canada, 228 pp, 1987.

[TrAt86] I. M. Trancoso and B. S. Atal, "Efficient procedures for finding the optimum innovation in stochastic coders," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Tokyo, Japan), IEEE CH2243-4/86, pp. 2375-2378, 1986.

[VeHe90] M. Viterli and C. Herley, "Wavelets and filter banks: Relationships and new results," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Albuquerque, NM), IEEE CH2847-2/90, pp. 1723-1726, 1990.

[ViMa75] R. Viswanathan, and J. Makhoul, "Quantization properties of transmission parameters in linear predictive systems," *IEEE Trans. Acoust., Speech, Signal Processing*,vol. ASSP-23, pp.309-321, June 1973.

[Wein91] C. J. Weinstein, "Opportunities for advanced speech processing in military computer-based systems," in *Proc. IEEE,* vo. 79, IEEE 0018-9219/91, pp. 1626-1641, November 1991.

[WeTC89] V. C. Welch, T. E. Tremain, and J. P. Campbell, Jr., "A comparison of U.S. Government standard voice coder," *Proc. Int. Conf. on Commun.*, IEEE CH2681-5/89, pp. 269-273, 1989.

[WoJC83] D. Y. Wong, B. H. Juang, and D. Y. Cheng. "Very low data rate speech compression with LPC vector and matrix quantization," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, (Boston, MA), IEEE CH1841-6/83, pp. 65-68, 1983.

APPENDIX A

PROPOSED FEDERAL STANDARD 1016

PROPOSED FEDERAL STANDARD 1016

TELECOMMUNICATIONS: ANALOG TO DIGITAL CONVERSION OF RADIO VOICE BY 4,800 BIT/SECOND CODE EXCITED LINEAR PREDICTION (CELP)

Pre-Third Draft

July 6, 1990

This proposed Federal standard has not yet been approved and is subject to modification.

1. SCOPE

1.1 Description. This standard specifies interoperability-related requirements for the conversion of analog voice to a 4,800 bit/s digitized form for digital radio transmission by a method known as Code Excited Linear Prediction (CELP) and the reverse process, the synthesis of analog voice from 4,800 bit/s CELP-digitized voice. In addition to digital radio applications, CELP is also very suitable for encrypted telephone use and other applications wherein voice must be digitized prior to encryption.

1.2 Purpose. This standard is to facilitate interoperability between radio telecommunication facilities and systems of the Federal Government.

1.3 Application. This standard shall be used by all Federal departments and agencies in the design and procurement of all radio equipment employing 4,800 bit/s digitized voice.

2. CONVENTIONS AND DEFINITIONS

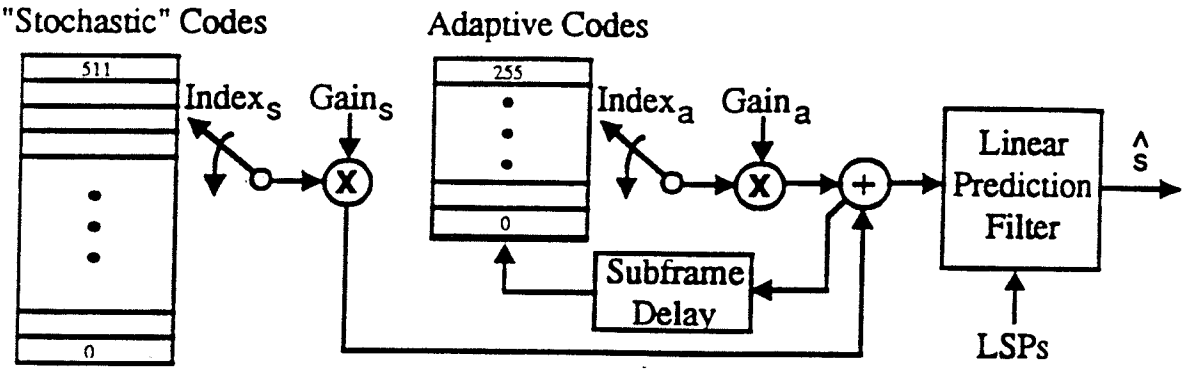
a. Frame. A CELP frame is 144 bits in length. A frame interval is 30 ms $\pm$.01 percent in duration and contains 240 voice samples (8,000 samples/s).

b. Subframe. A CELP subframe is 1/4 the length of a frame. Thus, a subframe is 7.5 ms $\pm$.01 percent in duration and contains 60 voice samples.

3. REQUIREMENTS

3.1 Voice Synthesis

3.1.1 General Description. Since Code Excited Linear Prediction (CELP) is an analysis-by-synthesis type of technique, voice synthesis is described first. As shown in the diagram below, CELP synthesis involves the excitation of a parameter-adjusted Linear Prediction Filter by the sum of gain-scaled codes selected from a fixed, stochastically-derived "stochastic code book" and an adaptive "code book", utilizing parameters transmitted in a 144-bit frame structure.



The fixed, stochastically-derived "stochastic" codes are described in section 3.3. "Stochastic" code gain is described in section 3.4. Adaptive codes are described in section 3.5 and adaptive code gain is described in section 3.6. Section 3.7 describes the Line Spectral Parameters (LSPs) that adjust the Linear Prediction Filter. Section 3.8 shows transmission format, including bit assignments for the transmitted information. Single bit error correction on some of the most significant bits is described in section 3.9.

3.1.2 Postfiltering. Postfiltering may be used to enhance the synthesized voice coming out of the CELP Linear Prediction Filter.

3.1.3 Lowpass Filtering. Lowpass (i.e., reconstruction) filtering shall be employed. A lowpass filter having characteristics as described in section 3.2.1 for frequencies above 200 Hz, with the addition of any digital-to-analog compensation needed for an essentially flat frequency response, is recommended.

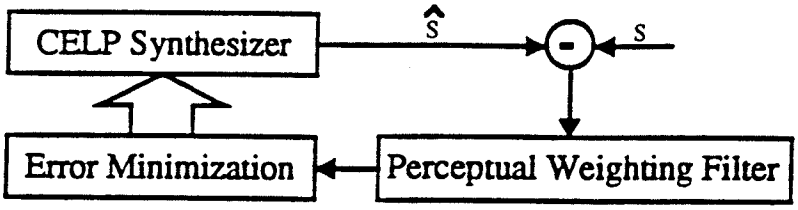
3.2 Voice Analysis

3.2.1 Voice Input Filtering. The analog voice input bandpass should be essentially flat from 200 to 3,400 Hz. A typical input filter has 3 dB attenuation points at 100 and 3,800 Hz; less than 1 dB of inband ripple; and minimum attentuations of 18 dB at 50 Hz, 18 dB at 4,000 Hz, and 46 dB above 4,400 Hz.

3.2.2 Analog-to-Digital Conversion. Analog-to-digital conversion shall use an 8 kHz $\pm$ 0.1 percent sampling frequency and have a dynamic range of at least 12 bits.

3.2.3 Amplitude Scaling. To maintain proper receiver voice levels, analysis shall be based upon use of input digitized voice whose peak values are -32,768.0 and +32,767.0.

3.2.4 Analysis. As reflected in the following diagram, CELP is an analysis-by-synthesis type of technique.



The objective of CELP analysis is to minimize the perceptual difference (i.e., find the best match) between the actual digitized voice and the synthesized voice resulting from use of the parameters to be transmitted. As shown in the diagram, 8 kHz-sampled analog voice is subtracted from the CELP synthesizer approximation and passed through a perceptual weighting filter (see section 3.2.5). The synthesizer parameters to be transmitted are adjusted for minimum perceptual error with respect to the actual input voice signal.

3.2.5 Perceptual Weighting Filter. It is recommended that a perceptual weighting filter be the cascade of a linear predictive whitening filter and a bandwidth expanded linear predictive synthesis filter. The bandwidth expanded linear predictive synthesis filter's poles are moved radially toward the origin by a weighting factor, typically 0.8.

3.3 "Stochastic" Codes. There are 512 fixed, stochastically-derived codes (i.e., vectors). During voice analysis, a code may be selected from a set smaller than 512 in order to reduce computational complexity (at the expense of voice reproduction quality). All 512 fixed, stochastically-derived codes shall be available for voice synthesis. Each of the fixed, stochastically-derived codes contains 60 ternary (i.e., either -1, 0, or 1) elements, representing information used to form the excitation for the Linear Prediction Filter over a subframe period (i.e., 8,000 elements/s for 7.5 ms). The fixed, stochastically-derived codes, 512 overlapped codes of 60 elements each, are created from a 1,082 element "stochastic code book" as described below.

3.3.1 "Stochastic Code Book". The 1,082 element "stochastic code book" is defined by the following FORTRAN computer program.

```
program codebook
implicit none
integer M, L, MAXCODE, WIDTH, j, k
parameter (M=512, L=60)
parameter (MAXCODE=2*(M-1)+L)
parameter (WIDTH=20)
real x(0:MAXCODE+1), THRESH
parameter (THRESH=1.2)
open (unit=10, file='codebook.h', status='new')
```

```
      do 50 k=0, MAXCODE, 2
        call noise(x(k), x(k+1))
        do 20 j=0, 1
           if(abs(x(k+j)) .lt. THRESH) then
              x(k+j) = 0.0
           else
              x(k+j) = sign(1.0, x(k+j))
           end if
   20   continue
   50 continue
      do 80 k=1, MAXCODE-WIDTH, WIDTH
         write (10,90) (int(x(j)), j=k, k+WIDTH-1)
   80 continue
      write (10,91) (int(x(j)), j=MAXCODE/WIDTH*WIDTH+1, MAXCODE)
      stop 'codebook.h generated'
   90 format(1x, 20(i3, ','))
   91 format(1x, 20(i3:, ','))
      end
      subroutine noise(x1, x2)
      implicit none
      real x1, x2
      integer random, i, j
      real f(2), f1, f2, s
   10 do 30 i=1, 2
         do 20 j=1, 4
            f(i) = (float(random() + 32768))/65535.
   20    continue
   30 continue
      f1=2.*f(1)-1.
      f2=2.*f(2)-1.
      s=f1*f1+f2*f2
      if(s .ge. 1.) goto 10
      s=sqrt(-2.*alog(s)/s)
      x1=f1*s
      x2=f2*s
      return
      end
      function random ()
      implicit none
      integer random
      integer MIDTAP, MAXTAP
      parameter (MIDTAP=2, MAXTAP=5)
      integer y(MAXTAP), j, k, temp
      save y, j, k
      data y/-21161, -8478, 30892, -10216, 16950/
      data j/MIDTAP/, k/MAXTAP/
      temp=iand(y(k)+y(j), 65535)
      if (temp .gt. 32767) temp=temp-65536
      y(k)=temp
      random=temp
      k=k-1
      if (k .le. 0) k=MAXTAP
      j=j-1
      if (j .le. 0) j=MAXTAP
      return
      end
```

The first and last 200 elements generated by the above FORTRAN program are shown below. The left-most and highest elements are lowest numbered (e.g., the element in the first row and third column is numbered 2).

```
 0,  1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  1,  0,  1,  0,  0,  0,  0, -1,
 0,  0, -1,  0, -1,  0,  1,  0,  0,  0,  0,  0,  0,  0, -1,  0, -1, -1,  0,  0,
-1,  0, -1,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  1,  0,
 0,  0,  0, -1,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  1, -1,  0,
 1,  0,  1,  0, -1,  1,  0,  0,  0,  0,  0, -1,  0,  0,  1,  0,  0,  0, -1,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0, -1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,
 0,  1, -1,  0,  0, -1, -1,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0, -1,  0, -1, -1,  0,  0, -1,  0,  1,  0,  0,  0,  1,  0,  0,  0,  0,
 0, -1,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0, -1,  0,  1,  0,  0,  0,  0,  0,

 .....................................................................

      0,  1,  0,  0,  0,  0,  0,  0,  1, -1,  0,  0,  1,  0,  0,  0,  0, -1,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  1,  0,  0,  1,  0,  0,
 0,  0,  1,  0,  0,  1,  0,  0, -1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,
 1,  0,  0, -1,  0,  0, -1,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0.  0,  1,  0,  0,  0,  0,  0,  1,  0,  0, -1,  1, -1,  0,  1,  0,  0,  0,
 0,  0,  0,  0,  1, -1,  0,  0,  0,  0,  0,  0,  0,  0, -1,  1,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0, -1,  0,
 0,  0, -1,  0, -1,  0,  0,  0,  0,  0,  0, -1,  0,  0, -1,  0,  0,  0,  0,  0,
 0,  0,  0,  1, -1,  0,  0,  0,  0, -1,  0,  1,  0,  0,  1,  0,  0,  0,  0, -1,
 0,  0,  0,  0,  1,  0,  0,  0,  0, -1,  0,  0, -1,  0,  0,  0,  0,  0,  0,  1,
 0,  0
```

3.3.2 Creation of Stochastically-derived Codes. The 512 fixed codes of 60 elements each are assembled from the 1,082 stocastically-derived "stochastic code book" elements as shown below. The left-most code elements are first-most in time.

Index	Elements In Codes	
511	0, 1, 2,...	..., 58, 59
510	2, 3, 4,...	..., 60, 61
.		
.		
n	2(511-n), 2(511-n)+1,..	..., 2(511-n)+59
.		
.		
1	1,020, 1,021, 1,022,...	..., 1,078, 1,079
0	1,022, 1,023, 1,024,...	..., 1,080, 1,081

3.4 "Stochastic" Code Gain. The relative amplitude (to the nearest table value) to be applied to the stochastically-derived code elements of each subframe is determined during analysis and coded into 5 bits according to the following table. The decimal index number is then transmitted in binary form. At the receiver, in voice synthesis, the gain index number is used to determine the relative amplitude of the stochastically-derived code elements during the subframe.

Index	Gain	Index	Gain	Index	Gain	Index	Gain
0	-1,330	8	-178	16	1	24	224
1	-870	9	-136	17	3	25	278
2	-660	10	-98	18	13	26	340
3	-520	11	-64	19	35	27	418
4	-418	12	-35	20	64	28	520
5	-340	13	-13	21	98	29	660
6	-278	14	-3	22	136	30	870
7	-224	15	-1	23	178	31	1,330

3.5 Adaptive Codes

3.5.1 Adaptive "Code Book" and Integer-delay Codes. The adaptive "code book" is a 147-element, shifting storage register that is updated at the start of every subframe with the previous 60 elements (subframe interval) of Linear Prediction Filter excitation. Element ordering is such that the first excitation elements into the Linear Prediction Filter are the first into the adaptive "code book". Elements already in the storage register are shifted up during the update process. The 128 integer-delay overlapped adaptive codes, of 60 elements each, are generated from the information in the adaptive "code book" as follows. Adaptive codes 60 through 147 are composed of elements -1 to -60, -2 to -61, ... through -88 to -147, respectively (where element -1 was the last element into the adaptive "code book"). Adaptive code "n", where n ranges from 20 to 59, repeats the adaptive "code book" elements sequentially from -1 to -n to form a 60-element code. As examples, adaptive code 20 repeats adaptive "code book" elements -1 ... -20 three times and adaptive code 59's elements run -1 ... -59, -1. Regarding order of utilization, element -1 of a particular adaptive code is used to compute the first element in time of Linear Prediction Filter excitation for a particular subframe period.

3.5.2 Noninteger-delay Codes. An additional 128 implicit, noninteger-delay codes, defined in section 3.5.4, can optionally be made available through interpolation. Forty point interpolation, or the equivalent, shall be used in both voice analysis and voice synthesis. (However, this does not preclude interpolating with fewer points, perhaps as few as 8 points, during preliminary code search computations). The process of interpolation is described in section 3.5.3.

3.5.3 Interpolation. A noninteger-delay adaptive code is interpolated from the nearest lower-valued integer-delay code (e.g., noninteger-delay code 49.67 is determined by interpolating from integer-delay code 49). The 60 elements of the integer-delay code are first renumbered as R0.....R59 for purposes of description, where R0 is normally the most negatively numbered (i.e., most delayed) element under the previous numbering system (i.e., -1 to -147), R1 is the next most negatively numbered element, etc,. For example, for integer-delay code 66, R0=element -66 and R1=element -65. For integer-delay code 21, because of the repetition within lower numbered integer-delay codes, R0=element -18 and R1=element -17. Likewise, for integer-delay code 58, R0=element -2, R1=element -1, and R3=element -58.

The next step in interpolation is to compute the weighting factors. The following FORTRAN program gives interpolation weighting factors for the interpolation of even numbers of points (N) equal to and less than 40. Note: interpolation with less than 8 points is not recommended and interpolation with other than 40 points is only for preliminary computations during voice analysis.

```
      program weights
      implicit none
      integer N, nf, i, j, k
      parameter (N=40, nf=5)
      real h(-6*N:6*N), w(-N/2:N/2-1,nf), f(nf), pi
      data f/0.25, 0.33333333, 0.5, 0.66666666, 0.75/
      pi = 4.0*atan(1.0)
      open(unit=10, file='weights', status='new'
      do 10 k = -6*N, 6*N
         h(k) = 0.54 + 0.46*cos(pi*k/(6*N))
   10 continue
      do 30 i = 1, nf
         do 20 j = -N/2, N/2-1
            w(j,i) = h(nint(12*(j+f(i))))*sin(pi*(j+f(i)))/(pi*(j+f(i)))
   20 continue
   30 continue
      write(10,69) f
   69 format(7x, 5f14.5)
      do 40 j = -N/2, N/2-1
         write(10,70) j, (w(j, k), k = 1, nf)
   40 continue
   70 format(3x, i4, 5e14.5)
      end
```

For N-point interpolation, the above program yields N weighting factors, numbered from -N/2 to N/2-1. As the final step in interpolation, these weighting factors are multiplied by the values of the adaptive code elements within a "window" around the element being interpolated (R). The sum of these multiplications is the interpolated value of that element (R'). For example, for 40 point interpolation, to find noninteger-delay code 66.67, element -66 becomes R0. Weighting factor -20 (i.e., -.0011766) is multiplied by the value of element -86, weighting factor -19 is multiplied by the value of element -85, ... , up to weighting factor 19 being multiplied by the value of element -47. The sum of these becomes the interpolated value of element -66 (i.e., R'0). The same takes place in a "window" around the remaining elements (R1 - R59). However, in the example given, the "window" runs out beyond R46 (i.e., for R46 weighting factor 19 corresponds to element -1). In such cases, the previously interpolated values for elements R0, R1, R2,.... (i.e., R'0, R'1, R'2, ...) are used to complete interpolation of the remaining elements. Thus, in the example given, when interpolating R58 (i.e., element -8), weighting factor -20 is multiplied by the value of element -28 and weighting factor 19 is multiplied by the value of already interpolated element R'11 (i.e., interpolated element -56). The sum of the 40 values found by multiplying the 40 weighting factors by the 40 element values (i.e., R38, R39, R40, ... R59, R'0, R'1, ... R'10, R'11) becomes the value of R'58 (i.e., interpolated element -8).

3.5.4 Coding Within 1st and 3rd Subframes for Transmission. The following table identifies the 8-bit values transmitted to represent each of the 256 (128 integer-delay and 128 noninteger-delay (implicit)) adaptive codes. The notation is such that, for example, hexadecimal ($) AF is expressed as 10101111 in binary form, where bit 7 is on the left and bit 0 is on the right.

Adaptive Code	Hex	Adaptive Code	Hex	Adaptive Code	Hex	Adaptive Code	Hex	Adaptive Code	Hex
20.00	$42	34.67	$C0	51.67	$98	68.67	$C5	97.00	$A1
20.33	$46	35.00	$C3	52.00	$90	69.00	$C9	98.00	$97
20.67	$47	35.33	$C2	52.33	$80	69.33	$C8	99.00	$87
21.00	$57	35.67	$D2	52.67	$9A	69.67	$C7	100.0	$9F
21.33	$56	36.00	$D3	53.00	$8A	70.00	$CB	101.0	$8F
21.67	$59	36.33	$D1	53.33	$82	70.33	$C6	102.0	$81
22.00	$58	36.67	$D0	53.67	$92	70.67	$CA	103.0	$91
22.33	$AE	37.00	$30	54.00	$1A	71.00	$D6	104.0	$9B
22.67	$BE	37.33	$32	54.33	$12	71.33	$DA	105.0	$8B
23.00	$BA	37.67	$3A	54.67	$00	71.67	$DB	106.0	$83
23.33	$B8	38.00	$31	55.00	$08	72.00	$D7	107.0	$93
23.67	$BC	38.33	$33	55.33	$06	72.33	$D9	108.0	$18
24.00	$AC	38.67	$3B	55.67	$0E	72.67	$D5	109.0	$10
24.33	$A8	39.00	$3F	56.00	$0F	73.00	$D8	110.0	$04
24.67	$94	39.33	$37	56.33	$07	73.33	$D4	111.0	$0C
25.00	$84	39.67	$3E	56.67	$17	73.67	$20	112.0	$16
25.33	$8C	40.00	$36	57.00	$1F	74.00	$28	113.0	$1E
25.67	$9C	40.33	$34	57.33	$0D	74.33	$38	114.0	$14
26.00	$9E	40.67	$4A	57.67	$05	74.67	$22	115.0	$1C
26.25	$8E	41.00	$4B	58.00	$1D	75.00	$2A	116.0	$F9
26.50	$86	41.33	$4E	58.33	$15	75.33	$39	117.0	$FA
26.75	$96	41.67	$4F	58.67	$FB	75.67	$29	118.0	$FD
27.00	$0A	42.00	$5F	59.00	$FF	76.00	$21	119.0	$E9
27.25	$02	42.33	$5E	59.33	$EB	76.33	$23	120.0	$FE
27.50	$0B	42.67	$5C	59.67	$EF	76.67	$2B	121.0	$E8
27.75	$03	43.00	$5D	60.00	$ED	77.00	$27	122.0	$FC
28.00	$1B	43.33	$54	60.33	$EA	77.33	$2F	123.0	$43
28.25	$13	43.67	$55	60.67	$EE	77.67	$25	124.0	$F2
28.50	$09	44.00	$50	61.00	$EC	78.00	$2D	125.0	$F6
28.75	$01	44.33	$51	61.33	$E6	78.33	$3D	126.0	$F8
29.00	$19	44.67	$AA	61.67	$E2	78.67	$35	127.0	$5B
29.25	$11	45.00	$A6	62.00	$E4	79.00	$3C	128.0	$5A
29.50	$F3	45.33	$A2	62.33	$E0	79.33	$2E	129.0	$63
29.75	$F7	45.67	$B6	62.67	$F4	79.67	$2C	130.0	$62
30.00	$E7	46.00	$B2	63.00	$F0	80.00	$26	131.0	$77
30.25	$E3	46.33	$BB	63.33	$60	81.00	$24	132.0	$76
30.50	$E5	46.67	$B0	63.67	$64	82.00	$49	133.0	$52
30.75	$E1	47.00	$B9	64.00	$74	83.00	$48	134.0	$53
31.00	$F1	47.33	$B4	64.33	$70	84.00	$4C	135.0	$66
31.25	$F5	47.67	$BD	64.67	$73	85.00	$4D	136.0	$67
31.50	$61	48.00	$A4	65.00	$72	86.00	$44	137.0	$CC
31.75	$65	48.33	$A0	65.33	$6C	87.00	$45	138.0	$CD
32.00	$75	48.67	$A9	65.67	$7C	88.00	$40	139.0	$AB
32.25	$71	49.00	$AD	66.00	$68	89.00	$41	140.0	$CF
32.50	$6D	49.33	$95	66.33	$78	90.00	$A7	141.0	$CE
32.75	$7D	49.67	$85	66.67	$7A	91.00	$A3	142.0	$DE
33.00	$69	50.00	$9D	67.00	$7E	92.00	$B7	143.0	$BF
33.25	$79	50.33	$8D	67.33	$6A	93.00	$B3	144.0	$DF
33.50	$7B	50.67	$89	67.67	$6E	94.00	$B1	145.0	$DD
33.75	$7F	51.00	$99	68.00	$6F	95.00	$B5	146.0	$DC
34.00	$6B	51.33	$88	68.33	$C4	96.00	$A5	147.0	$AF
34.33	$C1								

Note: Before an adaptive code is selected for transmission, consideration should be given to the possibility that a multiple of actual "pitch" is being selected. Submultiples of a selected code's pitch equivalent should be examined and may be utilized instead if results are found to be within approximately 0.5 dB of a selected code's squared prediction error.

3.5.5 Coding Within 2nd and 4th Subframes for Transmission. The adaptive code selected for transmission is represented in 6 bits, based upon the adaptive code selected in the previous subframe. Utilizing the table of adaptive codes in section 3.5.4, if a previous subframe's adaptive code was in the range of 20.00 - 29.25, the adaptive code to be transmitted could run from 20.00 to 38.33. If the previous subframe's adaptive code was in the range of 115.0 - 147.0, the adaptive code could run from 84.00 to 147.0. Otherwise, the 6 bits will code the range from 31 codes lower to 32 codes higher than the previous subframe's adaptive code (considering both integer-delay and noninteger-delay adaptive codes). This is true even if an implementation uses only integer adaptive codes. Binary coding is such that the lowest numbered adaptive code is coded as 000000 and the highest numbered adaptive code is coded as 111111. For example, for previous subframe adaptive codes between 29.50 and 114.0, binary 011111 indicates no difference from the adaptive code of the previous subframe.

3.6 Adaptive Code Gain. The relative amplitude (to the nearest table value) to be applied to the adaptive code elements of each subframe is determined during analysis and coded into 5 bits according to the following table. The decimal index number is then transmitted in binary form. At the receiver, in voice synthesis, the adaptive code gain index number is used to determine the relative amplitude of the adaptive code elements during the subframe.

Index	Gain	Index	Gain	Index	Gain	Index	Gain
0	-.993	8	0.255	16	0.780	24	1.062
1	-.831	9	0.368	17	0.816	25	1.117
2	-.693	10	0.457	18	0.850	26	1.193
3	-.555	11	0.531	19	0.881	27	1.289
4	-.414	12	0.601	20	0.915	28	1.394
5	-.229	13	0.653	21	0.948	29	1.540
6	0.000	14	0.702	22	0.983	30	1.765
7	0.193	15	0.745	23	1.020	31	1.991

3.7 Linear Prediction

3.7.1 Linear Prediction Filter. A 10th order Linear Prediction Filter is excited by the sum of the gain-adjusted "stochastic" and adaptive codes to synthesize voice.

3.7.2 Line Spectral Parameters. During voice analysis, the parameters of the Linear Prediction Filter are determined and transmitted on a 30 ms frame basis as 10 Line Spectral Parameters (LSPs), on the basis of 10th order linear prediction. It is recommended that LSPs be determined with no preemphasis, 15 Hz bandwidth expansion (i.e., 0.994 weighting factor), and a 30 ms nonoverlapped Hamming window spanning the last two subframes of the present frame and first two subframes of the next frame. Shown below is the coding to be employed for the 10 LSPs. Note: LSPs are expressed in terms of frequency (Hz) and must be monotonic (i.e., higher numbered LSPs must not contain lower frequency LSPs than lower numbered LSPs).

Index	LSP1 (Hz)	LSP2 (Hz)	LSP3 (Hz)	LSP4 (Hz)	LSP5 (Hz)	LSP6 (Hz)	LSP7 (Hz)	LSP8 (Hz)	LSP9 (Hz)	LSP10 (Hz)
0	100	210	420	620	1,000	1,470	1,800	2,225	2,760	3,190
1	170	235	460	660	1,050	1,570	1,880	2,400	2,880	3,270
2	225	265	500	720	1,130	1,690	1,960	2,525	3,000	3,350
3	250	295	540	795	1,210	1,830	2,100	2,650	3,100	3,420
4	280	325	585	880	1,285	2,000	2,300	2,800	3,200	3,490
5	340	360	640	970	1,350	2,200	2,480	2,950	3,310	3,590
6	420	400	705	1,080	1,430	2,400	2,700	3,150	3,430	3,710
7	500	440	775	1,170	1,510	2,600	2,900	3,350	3,550	3,830
8		480	850	1,270	1,590					
9		520	950	1,370	1,670					
10		560	1,050	1,470	1,750					
11		610	1,150	1,570	1,850					
12		670	1,250	1,670	1,950					
13		740	1,350	1,770	2,050					
14		810	1,450	1,870	2,150					
15		880	1,550	1,970	2,250					

3.7.3 LSP Weighted Averaging. To determine the LSP values to be used for each subframe, weighted averaging is used between the LSPs transmitted with the previous frame and the LSPs transmitted with the present frame. For each of the subframes of a given frame, the following weighting is given to the LSPs transmitted with the previous and present frames before they are utilized.

Subframe	Previous LSPs	Present LSPs
1	7/8	1/8
2	5/8	3/8
3	3/8	5/8
4	1/8	7/8

Thus, for example, if LSP 1 of a frame is 250 Hz and LSP 1 of the previous frame was 340 Hz, the value used for subframe 1 of the present frame is: 7/8(340 Hz) + 1/8(250 Hz) = 328.75 Hz. For each subframe, this method of weighted averaging is employed for all 10 LSPs.

3.8 Transmission Format

3.8.1 Transmission Rate. The transmission rate for CELP shall be 4,800 bits/s $\pm$ .01 percent.

3.8.2 Bit Assignment. The table below gives the assignment of the 144 bits in each CELP frame (30 ms at 4,800 bits/s). Order of transmission is from bit 1 to bit 144. Abbreviations used in the following table are:

i = bit number, with 0 being the least significant bit
n = subframe number
CG(n)-i = Fixed, Stochastically-derived Code Gain
CI(n)-i = Fixed, Stochastically-derived Code Index
HP-i = Parity
LSP j-i = Line Spectral Parameter (LSP), where j = LSP number
PD(n)-i = Adaptive Code Index
PG(n)-i = Adaptive Code Gain
SP = Expansion (Bishnu) Bit
SY = Synchronization Bit

Bit	Description	Bit	Description	Bit	Description	Bit	Description
1	PG(4)-4	37	PD(2)-0	73	PD(1)-4	109	CI(1)-2
2	PD(3)-4	38	CI(4)-1	74	CG(3)-2	110	PG(2)-1
3	LSP 1-1	39	LSP 9-0	75	LSP 7-1	111	CI(3)-7
4	CG(2)-4	40	CI(3)-8	76	CI(2)-7	112	LSP 4-0
5	CI(3)-3	41	PG(1)-4	77	CI(3)-0	113	CI(2)-5
6	CI(1)-8	42	CG(2)-2	78	PD(2)-5	114	PD(1)-7
7	PD(4)-0	43	PD(1)-3	79	LSP 4-1	115	PG(1)-0
8	LSP 8-0	44	LSP 6-1	80	CG(1)-0	116	CG(4)-4
9	PG(2)-3	45	CI(3)-4	81	PG(4)-3	117	LSP 5-0
10	CG(3)-0	46	CI(2)-2	82	LSP 9-1	118	PD(4)-2
11	PD(1)-5	47	CG(1)-4	83	PD(3)-6	119	CI(1)-3
12	LSP 3-3	48	PD(2)-3	84	CI(1)-4	120	CI(3)-1
13	CI(2)-3	49	LSP 1-2	85	CG(2)-1	121	LSP 7-2
14	CI(4)-4	50	PG(3)-2	86	LSP 6-2	122	CI(4)-2
15	PD(2)-1	51	HP-1	87	CI(4)-3	123	PD(1)-1
16	LSP 10-0	52	PD(3)-1	88	PG(2)-2	124	PG(2)-4
17	PG(1)-3	53	CG(4)-3	89	PD(4)-3	125	CG(3)-3
18	CG(4)-0	54	LSP 8-1	90	LSP 1-0	126	LSP 3-1
19	LSP 5-2	55	PG(3)-0	91	CG(4)-2	127	CI(1)-7
20	PD(3)-0	56	CI(2)-8	92	LSP 8-2	128	PD(3)-2
21	HP-0	57	PD(4)-1	93	CI(2)-4	129	CI(2)-6
22	CI(1)-1	58	CI(4)-0	94	HP-2	130	LSP 9-2
23	CI(4)-8	59	LSP 3-2	95	PD(2)-2	131	PG(4)-1
24	LSP 2-2	60	PG(2)-0	96	LSP 3-0	132	CG(1)-1
25	PG(3)-1	61	PD(1)-6	97	PG(1)-2	133	PD(2)-4
26	PD(4)-5	62	CG(2)-0	98	CG(3)-4	134	HP-3
27	CG(1)-3	63	CI(3)-6	99	LSP 10-2	135	LSP 6-0
28	CI(3)-5	64	LSP 10-1	100	CI(4)-5	136	PG(3)-3
29	LSP 7-0	65	PG(1)-1	101	CI(2)-0	137	CI(4)-6
30	CI(2)-1	66	CI(4)-7	102	PD(1)-2	138	PD(1)-0
31	PD(3)-7	67	PD(3)-3	103	LSP 5-1	139	LSP 2-3
32	CI(1)-0	68	CG(1)-2	104	SP-0	140	CG(4)-1
33	PG(4)-0	69	LSP 5-3	105	PG(4)-2	141	CI(3)-2
34	LSP 4-3	70	CI(1)-6	106	CG(2)-3	142	LSP 4-2
35	CG(3)-1	71	LSP 2-0	107	LSP 2-1	143	PD(3)-5
36	CI(1)-5	72	PG(3)-4	108	PD(4)-4	144	SY

3.8.3 Synchronization Bit. The single synchronization bit per frame shall alternate between binary 0 and 1 from frame to frame. It should be coded as binary 0 for the first frame transmitted.

3.8.4 Expansion (Bishnu) Bit. When following this standard, the "Bishnu" bit shall be set to binary 0 in each frame. The purpose of this bit is to allow for future transition to as yet undefined coding techniques.

3.9 Error Protection

3.9.1 Encoding. Forward error correction of 11 bits with a (15,11) Hamming parity code is provided. The 11 bits protected from a single bit error are: PD(1)-5, PD(1)-6, PD(1)-7, PG(1)-4, PG(2)-4, PD(3)-5, PD(3)-6, PD(3)-7, PG(3)-4, PG(4)-4, and SP. (See section 3.8.2 for an explanation of these abbreviations). The 4 parity bits are encoded as follows. Parity bits HP-0 through HP-3 each represent an even parity computation on 7 of the 11 protected bits as shown below (i.e., a parity bit is 0 if the sum of the 7 protected bits is even).

HP-0 -- PD(1)-5, PD(1)-6, PG(1)-4, PG(2)-4, PD(3)-6, PG(3)-4, and SP.
HP-1 -- PD(1)-5, PD(1)-7, PG(1)-4, PD(3)-5, PD(3)-6, PG(4)-4, and SP
HP-2 -- PD(1)-6, PD(1)-7, PG(1)-4, PD(3)-7, PG(3)-4, PG(4)-4, and SP
HP-3 -- PG(2)-4, PD(3)-5, PD(3)-6, PD(3)-7, PG(3)-4, PG(4)-4, and SP

3.9.2 Decoding. The 4 parity bits (HP-0,1,2, and 3) shall be used to correct single errors in the 11 protected bits as follows. Parity bits are calculated on the received 11 protected bits as described in section 3.9.1. Then, these calculated parity bits are Exclusive ORed with the received parity bits. The following table shows how to correct for a single bit error by reversing the binary value of one of the protected bits based upon the resulting binary value of the Exclusive ORed parity bits.

Parity Result	Invert	Parity Result	Invert
3	PD(1)-5	11	PD(3)-6
5	PD(1)-6	12	PD(3)-7
6	PD(1)-7	13	PG(3)-4
7	PG(1)-4	14	PG(4)-4
9	PG(2)-4	15	SP
10	PD(3)-5		

3.10 General Application Notes

3.10.1 Amplitude Scaling. "Stochastic" elements and gain values contained in this standard are based upon use of input digitized voice that can assume any integer value in the range -32,768.0 to +32,767.0 and an impulse of 1.0 to calculate the impulse response of the perceptual weighting filter.

3.10.2 Filter Structure. Various filter structures may be used in the implementation of this standard. All implementations must be interoperable with implementations based upon Direct Form (i.e., block-wise) filtering.

3.10.3 Smoothing. It is desirable to employ smoothing to prevent loud clipped voice (i.e., blasts) and squeaks. Smoothing based upon estimates of the channel error rate (provided by the error correction mechanism described in section 3.9) is recommended.

4 EFFECTIVE DATE. The use of this standard by U.S. Government departments and agencies is mandatory effective 180 days after the date of this standard.

5. CHANGES: When a Government department or agency considers that this standard does not provide for its essential needs, a statement citing specific requirements shall be sent in duplicate to the General Services Administration (K), Washington, DC 20405, in accordance with the provisions of Federal Information Resources Management Regulation 41 CFR 201-8.101-3. The General Services Administration will determine the appropriate action to be taken and will notify the agency.

TECHNICAL DEVELOPMENT ACTIVITY:

National Security Agency
Information Security Organization
Fort George G. Meade, MD 20755

PREPARING ACTIVITY:

National Communications System
Office of Technology and Standards
Washington, DC 20305-2010

APPENDIX B

SOFTWARE LISTINGS

PROGRAM RUNNING ON PC HOST:

(1) CELP_PC.C CONTAINING HOST CONTROLLER AND I/O PROCESSOR
INCLUDE: PC_1.H (SHARE WITH PC.C)
CELP_PC.H
CELP_CMD.H (USED ALSO BY EVM PROGRAM)
(2) PC.C CONTAINING HOST-EVM COMMUNICATION
INCLUDE: PC_1.H
PC_2.H

BOTH PROGRAM MUST BE LINKED TOGETHER.

COMPILATION FILE: CELP_PC.MAK

```
/*****************************************************************************************/
/*  CELP_PC.C                                                                            */
/*                                                                                       */
/*  CELP CODING: PC HOST CODE                                                            */
/*                                                                                       */
/*  COMPILED WITH QUICKC (MICROSOFT)                                                     */
/*       FOR 80286                                                                       */
/*       WITH HUGE MEMORY MODEL                                                          */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                  */
/*  MODIFIED BY ARMEIN LANGI, UNIVERSITY OF MANITOBA, 1991                               */
/*****************************************************************************************/
/*  PC COMMAND TRANSFER/USER INTERFACE PROGRAM                                           */
/*=======================================================================*/
#include <stdlib.h>  /* INCLUDE HEADERS FOR STANDARD LIBRARY FILES USED   */
#include <stdio.h>
#include <conio.h>   /* TURBO C SPECIFIC CONSOLE I/O HEADER FILE          */
#include "pc_1.h"          /* EVM SUPPORT PROTOTYPES, MACROS, STRUCTS*/
#include "celp_pc.h"       /* GLOBALS, LOCAL STRUCTURES, MACROS         */
#include "celp_cmd.h"      /* SHARED HOST/EVM COMMAND WORDS             */

/*=======================================================================*/
/* MAIN() MAIN PROGRAM LOOP                                              */
/*=======================================================================*/
void main(void)
{
        RESET_C30;
    init_evm();                 /* CLEAR ANY DATA TO EVM                 */
    buffer = 240;               /* SET EVM BLOCK TRANSFER SIZE           */
    display_menu(main_menu);    /* DISPLAY MENU                          */

    for(;;)
    {
        if (kbhit())            /* WAIT FOR KEYSTROKE                    */
        {
            command_process();      /* PROCESS USER KEYSTROKES           */
        display_menu(main_menu);    /* REDISPLAY MENU                    */
        }
        if    (play)  playing(PLAYING); /* PLAY  IF IN PLAY MODE         */
        else if (record)
        {
            recording(RECORDING);       /* RECORD IF IN RECORD MODE      */
            /*-------------------------------------------*/
            /* IF END OF DISK STORAGE BUFFER TERMINATE RECORD             */
            /*-------------------------------------------*/
        if (index >= MAX_FILE)
            {
                record = OFF;       /* TURN OFF RECORD FLAG              */
                write_disk();       /* WRITE DATA TO DISK                */
                index = 0;          /* RESET DISK STORAGE BUFFER INDEX*/
                display_menu(main_menu); /* REDISPLAY MENU               */
            }
        }
    }
}
/*=======================================================================*/
/* DISPLAY_MENU() DISPLAY AN ARRAY OF STRINGS ON THE SCREEN              */
```

```
/*      menu: AN ARRAY OF STRINGS TERMINATED BY A NULL                          */
/*==============================================================*/
void display_menu(char **menu)
{
    CLRSCR();                   /* CLEAR SCREEN                                 */
    while(*menu) puts(*menu++);         /* PRINT ALL MENU ITEMS                 */
    if    (record) puts("Recording..."); /* NOTE ANY RECORD                     */
    else if (play)  puts("Playing..."); /* OTHERWISE NOTE ANY PLAY              */
}
/*==============================================================*/
/* COMMAND_PROCESS(): PROCESS A COMMAND BASED ON USER KEYSTROKE
    */
/*==============================================================*/
void command_process(void)
{
    int command;                /* COMMAND VARIABLE                             */
    command = get_command();        /* DECODE KEYSTROKE                         */
    switch(command)
    {
        /*-------------------------------------------------------------------*/
        /* THE FOLLOWING COMMANDS REQUIRE SIMPLE COMMAND PASSING
WITH    */
        /*   NO DATA EXCHANGE                                                */
        /*-------------------------------------------------------------------*/
        case REC_RS232:             /* GET CELP FROM SERIAL PORT            */
         printf("Sorry, not available yet because has not been debbuged...!\n");
         command=NONE; /* Wait for the next version that includes RS232 program */
         break;
    case SEND_RS232:            /* SEND CELP TO SERIAL PORT                 */
         printf("Sorry, not available yet because has not been debbuged...!\n");
         command=NONE; /* Wait for the next version that includes RS232 program */
         break;
    case REC_CELP:              /* GET CELP FROM A/D                        */
         RecordCELP();
         break;
    case PLAY_CELP:             /* SEND CELP TO D/A                         */
         PlayCELP();
         break;
    case PCM2CELP:              /* CONVERT PCM TO CELP                      */
        PCM_2_CELP();
        RESET_C30;
        init_evm();             /* CLEAR ANY DATA TO EVM                    */
        break;
    case CELP2PCM:              /* CONVERT CELP TO PCM                      */
        CELP_2_PCM();
        RESET_C30;
        init_evm();             /* CLEAR ANY DATA TO EVM                    */
        break;
        /*-------------------------------------------------------------------*/
        /*THE FOLLOWING COMMANDS SIMPLY SET THE MODE OF THE HOST
PROGRAM*/
        /*-------------------------------------------------------------------*/
        case REC_PCM:           /* RECORD ENCODED DATA TO DISK              */
            command = NONE;     /* NO EVM COMMUNICATIONS NECESSARY*/
            /*-----------------------------------------------*/
            /* ABORT IF ALREADY IN SOME DATA TRANSFER MODE                   */
            /*-----------------------------------------------*/
```

```
        if (record || play) break;
        record = ON;          /* SET RECORD MODE                                  */
        open_file("wb");      /* OPEN DATA FILE TO RECORD                         */
        break;
    case PLAY_PCM:            /* PLAY RECORDED DATA FROM DISK                     */
        command = NONE;       /* NO EVM COMMUNICATIONS NECESSARY*/
        /*-------------------------------------------*/
        /* ABORT IF ALREADY IN SOME DATA TRANSFER MODE                            */
        /*-------------------------------------------*/
        if (record || play) break;
        play = ON;            /* SET PLAYBACK MODE                                */
        open_file("rb");      /* OPEN DATA FILE FOR CODED DATA                    */
    fread(&endd, sizeof(long int), 1,  file); /* READ DATA LENGTH                 */
    if(endd <= BUFFSIZE)
    {
     fread(disk, sizeof(int),(int) endd, file); /* READ DATA                     */
    }
    else
    {
      if(endd <= BUFFSIZE * 2)
      {
     fread(disk, sizeof(int),BUFFSIZE, file); /* READ DATA                      */
     fread(disk1, sizeof(int),(int) (endd-BUFFSIZE), file); /* READ DATA        */
      }
       else
      {
     fread(disk, sizeof(int),BUFFSIZE, file); /* READ DATA                      */
     fread(disk1, sizeof(int),BUFFSIZE, file); /* READ DATA                     */
     fread(disk2, sizeof(int),(int) (endd-BUFFSIZE*2), file); /* READ DATA       */


      }
    }
    fclose(file);             /* CLOSE FILE                                       */
        break;
    case STOP:
        command = NONE;       /* NO EVM COMMUNICATIONS NECESSARY*/
        if (record) write_disk(); /* WRITE RECORDED DATA TO DISK                  */
        record = OFF;         /* TURN OFF DATA TRANSFER MODES                     */
        play   = OFF;
        index  = 0;           /* RESET DISK STORAGE BUFFER INDEX*/
        RESET_C30;
        init_evm();           /* CLEAR ANY DATA TO EVM                            */
        break;
    /*-------------------------------------------------------------------------*/
    /* EXIT PC HOST PROGRAM                                                    */
    /*-------------------------------------------------------------------------*/
    case QUIT:
    CLRSCR();             /* CLEAR THE SCREEN                                   */
    RESET_C30;
    exit(EXIT_SUCCESS);   /* EXIT                                               */
        break;
    /*-------------------------------------------------------------------------*/
    /* FOR UNRECOGNIZED COMMANDS EXIT WITHOUT EVM COMNUNICATION
*/
    /*-------------------------------------------------------------------------*/
    default:
        command = NONE;
```

```
            break;
        }
        if (command) send_command(command);
}
/*================================================================*/
/* OPEN_FILE() ATTEMPTS TO OPEN A FILE UNTIL GIVEN A VALID FILENAME */
/*    mode: STRING CONTAINING THE MODE TO OPEN THE FILE            */
/*================================================================*/
void open_file(char *mode)
{
    CLRSCR();                   /* CLEAR THE SCREEN                 */
    do
    {
        printf("Enter filename: ");      /* PROMPT FOR FILENAME     */
        scanf("%13s",filename);          /* GET USER INPUT          */
        file = fopen(filename, mode);    /* ATTEMPT TO OPEN FILE    */
        if (!file) puts("Bad file name."); /* INFORM OF ERROR       */
    }
    while(!file);
}

void PlayCELP(void)
{
unsigned i,k;
int command;

    printf("For the source CELP file\n");
    open_infile();
    fread(&endd, sizeof(long int), 1, infile); /* READ DATA LENGTH   */
    k = (unsigned int) (endd/9);
    printf("Number of frame = %d\n",k);
    if(endd <= BUFFSIZE)
    {
        fread(disk, sizeof(int),(int) endd, file); /* READ DATA      */
    }
    else
    {
        if(endd <= BUFFSIZE*2)
        {
            fread(disk, sizeof(int),BUFFSIZE, file); /* READ DATA    */
            fread(disk1, sizeof(int),(int) (endd-BUFFSIZE), file); /* READ DATA */
        }
        else
        {
            fread(disk, sizeof(int),BUFFSIZE, file); /* READ DATA    */
            fread(disk1, sizeof(int),BUFFSIZE, file); /* READ DATA   */
            fread(disk2, sizeof(int),(int) (endd-BUFFSIZE*2), file); /* READ DATA */


        }
    }
    fclose(infile);
    command = 0;
    while(!command)
    {
        k = (unsigned) (endd/9);
        send_command(ERASE_BUFFERS);
```

```
        index=0;
        while(k>0)
        {
            buffer=9;
            playing(INTRCELP);
            send_command(PLAY_CELP);
            k=k-1;
        }
        if(kbhit())
        {
            command = get_command();
            if(command != STOP)
            {
                command = 0;
            }
            else
            {
                command = 1;
            }
        }
    }
}

void RecordCELP(void)
{
unsigned long file_size=0;

unsigned i,k;

    printf("For the destination CELP file\n");
    open_ofile();
    send_command(ERASE_BUFFERS);
    index=0;
    k = 1;
    while(k>0)
    {
      send_command(REC_CELP);
      buffer=9;
      recording(OUTTRCELP);
        if (index >= MAX_FILE)
        {
        record = OFF;          /* TURN OFF RECORD FLAG                    */
        write_disk();          /* WRITE DATA TO DISK                      */
        index = 0;             /* RESET DISK STORAGE BUFFER INDEX*/
        k = 0;
        }
    }
}

void PCM_2_CELP(void)
{
unsigned long file_size=0;
unsigned i,k;

    printf("For the source PCM file\n");
    open_infile();
    printf("For the destination CELP file\n");
```

```
    open_ofile();
    fread(&file_size, sizeof(long int), 1, infile);  /* READ DATA LENGTH          */
    k = (unsigned int) (file_size/240);
    printf("Number of frame = %d\n",k);
    file_size = 9 * (unsigned long) k;
    fwrite(&file_size,sizeof(long int),1, outfile); /* WRITE DATA LENGTH          */
    send_command(ERASE_BUFFERS);
    while(k>0)
    {
      fread(disk, sizeof(int),240, infile);  /* READ DATA                         */
      buffer=240;
      index=0;
      playing(OSPEECH);
      send_command(PCM2CELP);

      buffer=9;
      index=0;
          recording(OUTTRCELP);

      fwrite(disk,sizeof(int),9, outfile); /* WRITE CODED DATA                    */
      k=k-1;
    }
    fclose(infile);
    fclose(outfile);
}

void CELP_2_PCM(void)
{
unsigned long file_size=0;
unsigned *un_disk=disk;
unsigned i,k;

    open_infile();
    open_ofile();
    fread(&file_size, sizeof(long int), 1, infile);  /* READ DATA LENGTH          */
    k = (unsigned int) (file_size/9);
    printf("Number of frame = %d\n",k);
    file_size = 240 * (unsigned long) k;
    fwrite(&file_size,sizeof(long int),1, outfile); /* WRITE DATA LENGTH          */
    send_command(ERASE_BUFFERS);
    while(k>0)
    {
      fread(disk, sizeof(int),9, infile);  /* READ DATA                          */
      buffer=9;
      index=0;
      playing(INTRCELP);

      send_command(CELP2PCM);

      buffer=240;
      index=0;
          recording(SYNPEECH);

      fwrite(disk,sizeof(int),240, outfile); /* WRITE CODED DATA                 */
      k=k-1;
    }
    fclose(infile);
    fclose(outfile);
```

```
}

void open_infile(void)
{
    do
    {
    printf("Enter source filename: ");     /* PROMPT FOR FILENAME                */
    scanf("%13s",filename);              /* GET USER INPUT                     */
    infile = fopen(filename, "rb");       /* ATTEMPT TO OPEN FILE               */
    if (!infile) puts("Bad file name.");  /* INFORM OF ERROR                    */
    }
    while(!infile);

}

void open_ofile(void)
{
    do
    {
    printf("Enter destination filename: "); /* PROMPT FOR FILENAME              */
    scanf("%13s",filename);              /* GET USER INPUT                     */
    outfile = fopen(filename, "wb");      /* ATTEMPT TO OPEN FILE               */
    if (!outfile) puts("Bad file name."); /* INFORM OF ERROR                    */
    }
    while(!outfile);

}
```

```
/*****************************************************************************/
/*  CELP_PC.H                                                                */
/*                                                                           */
/*  CELP CODING: PC HOST STRUCTURES, MACROS, GLOBALS                         */
/*                                                                           */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                      */
/*  MODIFIED BY ARMEIN LANGI, UNIVERSITY OF MANITOBA, 1991                   */
/*****************************************************************************/
/* FUNCTION PROTOTYPES                                                       */
/*===========================================================*/
void main(void);
void display_menu(char **menu);
void command_process(void);
void open_file(char *mode);
void CELP_2_PCM(void);
void PCM_2_CELP(void);
void open_infile(void);
void open_ofile(void);
void PlayCELP(void);
void RecordCELP(void);

/*===========================================================*/
/* MACROS                                                                    */
/*===========================================================*/
#define MAX_FILE 96000  /* SET FOR 24000 BUFFERS IN DISK STORAGE BUFFER  */
/*===========================================================*/
/* GLOBAL VARIABLES                                                          */
/*===========================================================*/
extern FILE *file;              /* GLOBAL FILE POINTER                       */
extern FILE *infile;
extern FILE *outfile;
extern unsigned int disk[];             /* BUFFER FOR DISK STORAGE           */
extern unsigned int disk1[];
extern unsigned int disk2[];
extern unsigned long int index;     /* INDEX INTO DISK STORAGE ARRAY         */
extern unsigned long int endd;      /* INDEX TO END OF PLAYBACK DATA         */
extern int record;              /* RECORD FLAG                               */
extern int play;                /* PLAYBACK FLAG                             */
extern int buffer;              /* BLOCK TRANSFER SIZE FROM EVM              */
extern int iobase;
char filename[13];              /* STRING FOR HOLDING FILENAME               */
/*-----------------------------------------------------------*/
/* ARRAY OF STRINGS FOR MENU                                                 */
/*-----------------------------------------------------------*/
char *main_menu[] =
{
    "                    CELP SPEECH SYSTEM\n\r",
    "                  4800 Bit-per-second Speech Coder\n\r",
    "                      Armein Langi",
    "                   University of Manitoba, 1991",
    "",
    "F1  Record PCM",
    "F2  Play PCM",
    "F3  Receive CELP from RS232",
    "F4  Send CELP to RS232",
    "F5  Record CELP",
    "F6  Play CELP",
```

```
    "F7  Convert from PCM to CELP",
    "F8  Convert from CELP to PCM",
    "F9  Stop (Playback or Record)",
    "END Quit",
    "",
    NULL
};
```

```
/*****************************************************************************************/
/*  CELP_CMD.H                                                                           */
/*                                                                                       */
/*  CELP CODING: SHARED PC HOST AND TMS320C30 CODE                                       */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                 */
/*  MODIFIED BY ARMEIN LANGI, UNIVERSITY OF MANITOBA, 1991                               */
/*****************************************************************************************/
/*  PC EXTENDED KEYBOARD VARIABLES                                                       */
/*=======================================================================*/
#define DEL         83
#define HOME        71
#define END         79
#define F1          59
#define F2          60
#define F3          61
#define F4          62
#define F5          63
#define F6          64
#define F7          65
#define F8          66
#define F9          67
#define F10         68
/*-----------------------------------------------------------------------------------------*/
/*    COMMANDS FROM THE PC TO THE EVM                                                    */
/*-----------------------------------------------------------------------------------------*/
#define REC_PCM       F1      /* START RECORDING PCM                                    */
#define PLAY_PCM      F2      /* START PLAYING PCM                                      */
#define REC_RS232     F3      /* RECEICE CELP FROM RS232                                */
#define SEND_RS232    F4      /* SEND CELP TO RS232                                     */
#define REC_CELP      F5      /* START RECORDING CELP                                   */
#define PLAY_CELP     F6      /* START PLAYING CELP                                     */
#define PCM2CELP      F7      /* CONVERT PCM TO CELP                                    */
#define CELP2PCM      F8      /* CONVERT CELP2PCM                                       */
#define STOP        F9      /* STOP PLAY OR RECORD                                    */
#define QUIT        END     /* EXIT FROM PC CONTROL PROGRAM                           */
#define RECORDING     128     /* DMA IN READY CONDITION FOR RECORD                      */
#define PLAYING       129     /* DMA IN READY CONDITION FOR PLAYBACK                    */
#define INTRCELP      130
#define SYNPEECH      131
#define OSPEECH       132
#define OUTTRCELP     133
#define ERASE_BUFFERS   134
#define BLOCK_SIZE    240     /* BLOCK SIZE FOR TRANSFER TO HOST                        */
                              /*    THIS VALUE IS STORED IN buffer                      */
```

```
/*****************************************************************************/
/*  PC.C                                                                     */
/*                                                                           */
/*  TMS320C30 EVALUATION MODULE HOST SUPPORT CODE                            */
/*                                                                           */
/*  COMPILED WITH QUICK C (MICROSOFT)                                        */
/*       FOR 80286                                                           */
/*       WITH HUGE MEMORY MODEL                                              */
/*                                                                           */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                      */
/*****************************************************************************/
/*  PC COMMAND TRANSFER/USER INTERFACE PROGRAM                               */
/*===========================================================================*/
#include <stdlib.h>  /* INCLUDE HEADERS FOR STANDARD LIBRARY FILES USED     */
#include <stdio.h>
#include <ctype.h>
#include <bios.h>   /* INCLUDE TURBO C SPECIFIC HEADER FILES                */
#include <dos.h>
#include <conio.h>
#include "pc_1.h"          /* EVM SUPPORT STRUCTURES AND MACROS             */
#include "pc_2.h"          /* EVM SUPPORT GLOBAL VARIABLES                  */

/*===========================================================================*/
/* INIT_EVM(): INITIALIZE EVM                                                */
/*===========================================================================*/
void init_evm(void)
{
    get_iobase();          /* GET I/O BASE ADDRESS OF EVM                    */
    RESET_EVM;
    WRITE_DATA(NONE);      /* CLEAR HOST REGISTER VALUE TO EVM               */
    RUN_C30;
}
/*===========================================================================*/
/* GET_IOBASE(): GET IOBASE ADDRESS OF TMS320C30 EVM                         */
/*===========================================================================*/
void get_iobase(void)
{
    char *d_options  = getenv("D_OPTIONS"); /* GET D_OPTION ENV VARIABLE     */
    int found_base  = OFF;             /* IOBASE (-p) OPTION FOUND           */
    if (!d_options) return;            /* IF NO ENVVAR RETURN DEFAULT*/
    while (*++d_options)               /* SEARCH ENTIRE STRING               */
    {
        /*-------------------------------------------------------------------*/
        /* SEARCH FOR '-P' IOBASE OPTION                                     */
        /*-------------------------------------------------------------------*/
        if (d_options[-1] == '-' && toupper(d_options[0]) == 'P')
        {
            found_base = ON;
            break;
        }
    }
    if (!found_base) return;       /* IF -p NOT FOUND RETURN DEFAULT*/
    sscanf(++d_options, "%x", &iobase);/* OTHERWISE READ IN IOBASE           */
    switch(iobase)
    {
        /*-------------------------------------------------------------------*/
        /* RETURN ONLY VALID VALUES FOR THE IOBASE ADDRESS                   */
```

```
        /*-------------------------------------------------------------------------*/
        case 0x240:
        case 0x280:
        case 0x320:
        case 0x340:
            break;
        /*-------------------------------------------------------------------------*/
        /* IF INVALID ENTRY IN IOBASE LOCATION RETURN ERROR AND EXIT               */
        /*-------------------------------------------------------------------------*/
        default:
        puts("Invalid I/O base");
        exit(EXIT_FAILURE);
    }
}
/*=========================================================================*/
/* GET_COMMAND(): DECODE KEYSTROKE FROM IBM EXTENDED KEYBOARD              */
/*=========================================================================*/
int get_command(void)
{
    unsigned int key;
    key = _bios_keybrd(_KEYBRD_READ); /* 16-BIT CODE FROM KEYBOARD          */
    fflush(stdin);              /* FLUSH LINGERING KEYSTROKES              */
    if (key & 0x0FF) return NONE;  /* IF REGULAR ASCII CHARACTER RETURN    */
    else      return (key >> 8); /* ELSE RETURN EXTENDED KEYBOARD CODE      */
}
/*=========================================================================*/
/* WRITE_DISK(): WRITE RECORDED DATA TO DISK                               */
/*=========================================================================*/
void write_disk(void)
{
    fwrite(&index, sizeof(long int), 1,file); /* WRITE LENGTH OF FILE      */
    if(index<=BUFFSIZE)
    {
     fwrite(disk,   sizeof(int),(int) index, file); /* WRITE CODED DATA    */
    }
    else
    {
    if(index<=BUFFSIZE*2)
    {
     fwrite(disk,   sizeof(int), BUFFSIZE, file); /* WRITE CODED DATA      */
     fwrite(disk1,  sizeof(int),(int) (index-BUFFSIZE), file); /* WRITE CODED DATA */
    }
    else
    {
     fwrite(disk,   sizeof(int), BUFFSIZE, file); /* WRITE CODED DATA      */
     fwrite(disk1,  sizeof(int), BUFFSIZE, file); /* WRITE CODED DATA      */
     fwrite(disk2, sizeof(int),(int) (index-BUFFSIZE*2), file); /* WRITE CODED DATA */

    }
    }
    fclose(file);                   /* CLOSE FILE                          */

}
/*-------------------------------------------------------------------------*/
/*  waits for EVM to be free to accept command                             */
/*-------------------------------------------------------------------------*/
#define RECORDING 128
```

```
#define PLAYING 129
#define TIMEOUT 325000

void wait_free(int command)
{
    long int count;

    count=0;
    while (READ_CMD & count++ < TIMEOUT);
    if (count >= TIMEOUT)
        {
        RESET_C30;
        switch (command)
            {
            case PLAYING : printf("EVM not free to accept PLAY");
            break;
            case RECORDING : printf("EVM not free to RECORD");
            break;
            default : printf("EVM not free to accept command");
            break;
            }
        exit(1);
        }
}
/*-------------------------------------------------------------------------------*/
/*  WAIT FOR ACKNOWLEDGE OF COMMAND  FROM EVM                                    */
/*-------------------------------------------------------------------------------*/
void wait_ack(int command)
{   long int count;

    count = 0;
    while (READ_CMD != command & count++ <TIMEOUT);
    if (count >= TIMEOUT)
        {
        RESET_C30;
        switch (command)
            {
            case PLAYING : printf("EVM not acking PLAY");
            break;
            case RECORDING : printf("EVM not acking RECORD");
            break;
            default : printf("EVM not acking command no %d",command);
            break;
            }
        exit(1);
        }
}
/*-------------------------------------------------------------------------------*/
/*  SEND_COMMAND(): SEND A COMMAND WORD TO THE EVM AND VERIFY
RECEPTION                                                                      */
/*-------------------------------------------------------------------------------*/
void send_command(int i)
{
    wait_free(i);
    WRITE_CMD(i);
    wait_ack(i);
    WRITE_CMD(NONE);
}
```

```
/*=============================================================*/
/* PLAYING(): SEND CODED DATA TO EVM                           */
/*=============================================================*/
void playing(int play_cmd)
{
    int i;              /* LOOP COUNTER                        */
    int *disk_addr;

    wait_free(play_cmd);    /* WAIT FOR EVM TO BE FREE         */
    WRITE_CMD(play_cmd);    /* WRITE COMMAND FOR EVM TO RECEIVE DATA */
    wait_ack(play_cmd);     /* WAIT FOR ACKNOWLEDGE*/
    if(index <BUFFSIZE)
    {
        disk_addr=&disk[index];
    }
    else
    {
         if (index <BUFFSIZE * 2)
         {
         disk_addr=&disk1[index-BUFFSIZE];
         }
         else
         {
        disk_addr=&disk2[index-BUFFSIZE*2];
         }
    }

    CLR_WRITE_ACK;          /* CLEAR ANY LATENT WRITE ACKNOWLEDGMENT */
    for(i = 0; i < buffer; i++) /* SEND DATA                   */
    {

        /*-------------------------------------------------------*/
        /* WRITE DATA FROM DISK STORAGE BUFFER TO EVM             */
        /*-------------------------------------------------------*/
    WRITE_DATA(disk_addr[i]);
    index++;
        /*-------------------------------------------------------*/
        /* WAIT FOR EVM TO READ DATA                              */
        /*-------------------------------------------------------*/
        do
        {
            UPDATE_STATUS0;
        }
        while(!IS_WRITE_ACK);
        CLR_WRITE_ACK;          /* CLEAR DATA WRITE ACKNOWLEDGE */
    }
    WRITE_CMD(NONE);            /* REMOVE COMMAND              */
    if (index == endd) index = 0;   /* PLAY IN A LOOP          */
}
/*=============================================================*/
/* RECORDING(): READ CODED DATA FROM EVM                       */
/*=============================================================*/
void recording(int record_cmd)
{
    int i,j;            /* LOOP COUNTER                        */
    int *disk_addr;
    wait_free(record_cmd);
```

```
    WRITE_CMD(record_cmd);      /* WRITE COMMAND FOR EVM TO SEND DATA     */
    wait_ack(record_cmd);
    if(index <BUFFSIZE)
    {
        disk_addr=&disk[index];
    }
    else
    {
         if (index <BUFFSIZE*2)
         {
         disk_addr=&disk1[index-BUFFSIZE];
         }
         else
         {
        disk_addr=&disk2[index-BUFFSIZE*2];
         }
    }

    CLR_READ_ACK;            /* CLEAR ANY READ ACKNOWLEDGEMENT              */
    READ_DATA;               /* DUMMY READ TO START TMS320C30 DMA           */
    for(i = 0; i < buffer; i++)
    {

        /*-------------------------------------------------------------------*/
        /* WAIT FOR EVM TO WRITE NEW DATA                                    */
        /*-------------------------------------------------------------------*/
        do
        {
            UPDATE_STATUS0;
        }
        while(!IS_READ_ACK);
    CLR_READ_ACK;          /* CLEAR DATA READ ACKNOWLEDGE                   */
    disk_addr[i] = READ_DATA;/* WRITE DATA TO DISK STORAGE BUFFER           */
    index++;
    }
    WRITE_CMD(NONE);         /* REMOVE COMMAND                              */
}

void clrscr(void)
{

}
```

```
/*****************************************************************************************/
/* PC_1.H                                                                                */
/*                                                                                       */
/*  TMS320C30 EVALUATION MODULE PC HOST SUPPORT FILE                                     */
/*        :MACROS, STRUCTURES, FUNCTION PROTOTYPES                                       */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                  */
/*****************************************************************************************/
#define OFF    0x00
#define ON     0x01
#define TOGGLE  0x01       /* XOR BITFIELD TO TOGGLE ON <-> OFF                          */
#define NONE    0x00       /* WORD REPRESENTING ABSENCE OF COMMAND                       */
#define BUFFSIZE   31920
/*****************************************************************************************/
/* FUNCTION PROTOTYPES                                                                   */
/*****************************************************************************************/
void init_evm(void);
void get_iobase(void);
int  get_command(void);
void write_disk(void);
void send_command(int i);
void playing(int play_cmd);
void recording(int record_cmd);
void clrscr(void);

/*****************************************************************************************/
/* COMMUNICATIONS MACROS                                                                 */
/*****************************************************************************************/
/*---------------------------------------------------------------------------------------*/
/* PC I/O SPACE BASE ADDRESS FOR EVM CONTROL REGISTERS                                   */
/*---------------------------------------------------------------------------------------*/
#define IOBASE_DEFAULT  0x0240

#define CONTROL5      iobase + 0x000A    /* CONTROL REGISTER                             */
#define STATUS0       iobase + 0x0400    /* STATUS0 REGISTER                             */
#define COM_CMD       iobase + 0x0800    /* COMMAND REGISTER                             */
#define COM_DATA      iobase + 0x0808    /* DATA   REGISTER                              */
#define SOFT_RESET    iobase + 0x0818    /* SOFT RESET LOCATION                          */
#define MINOR_CMD     iobase + 0x0014    /* MINOR COMMAND REGISTER                       */

/*---------------------------------------------------------------------------------------*/
/* BIT MASKS FOR READ AND WRITE ACKNOWLEDGE BITS IN STATUS REGISTER  */
/*---------------------------------------------------------------------------------------*/
#define MREAD_ACK      0x0002
#define MWRITE_ACK     0x0004
#define UPDATE_REQ     0x6044

#define WRITE_CMD(x)     outp(COM_CMD,x)    /* WRITE  8 BIT COMMAND                     */
#define READ_CMD         inp(COM_CMD)       /* READ   8 BIT COMMAND                     */
#define WRITE_DATA(x)    outpw(COM_DATA, x)  /* WRITE 16 BIT DATA                       */
#define READ_DATA        inpw(COM_DATA)     /* READ  16 BIT DATA                        */
#define RESET_EVM        outpw(SOFT_RESET, 0) /* RESET EVM                              */
#define RESET_C30        outpw(CONTROL5,0x808)/* PLACE 'C30 IN RESET                    */
#define RUN_C30          outpw(CONTROL5,0x800)/* PULL 'C30 OUT OF RESET*/
#define READ_STATUS0     inpw(STATUS0)      /* READ  STATUS WORD                        */
#define WRITE_MINOR_CMD(x) outpw(MINOR_CMD,x)  /* WRITE STATUS WORD                    */
```

```
/*-------------------------------------------------------------------------------------------*/
/* CHECKS FOR READ OR WRITE ACKNOWLEDGMENT                                                   */
/*-------------------------------------------------------------------------------------------*/
#define IS_READ_ACK     (READ_STATUS0 & MREAD_ACK)
#define IS_WRITE_ACK    (READ_STATUS0 & MWRITE_ACK)
/*-------------------------------------------------------------------------------------------*/
/* CLEAR STATUS0 BITS                                                                        */
/*-------------------------------------------------------------------------------------------*/
#define CLR_READ_ACK    WRITE_MINOR_CMD(MREAD_ACK)
#define CLR_WRITE_ACK   WRITE_MINOR_CMD(MWRITE_ACK)
#define UPDATE_STATUS0  WRITE_MINOR_CMD(UPDATE_REQ)

#define CLRSCR()  clrscr()
```

```
/*****************************************************************************************************/
/*  PC_2.H                                                                                           */
/*                                                                                                   */
/*  TMS320C30 EVALUATION MODULE PC HOST SUPPORT FILE                                                 */
/*        :GLOBAL VARIABLES                                                                          */
/*                                                                                                   */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                              */
/*****************************************************************************************************/
/* GLOBAL VARIABLES                                                                                  */
/*****************************************************************************************************/
int iobase   = IOBASE_DEFAULT;      /* EVM I/O BASE ADDRESS                                          */
unsigned long int index    = 0;     /* INDEX INTO DISK STORAGE ARRAY                                 */
unsigned long int endd     = 0;     /* INDEX TO END OF PLAYBACK DATA                                 */
int record  = OFF;                  /* RECORD FLAG                                                   */
int play    = OFF;                  /* PLAYBACK FLAG                                                 */
int buffer;                         /* BLOCK TRANSFER SIZE FROM EVM                                  */

unsigned int disk[BUFFSIZE];                /* DISK STORAGE ARRAY                                    */
unsigned int disk1[BUFFSIZE];
unsigned int disk2[BUFFSIZE];

FILE *file;                         /* FILE POINTER TO CURRENT FILE                                  */
FILE *infile;
FILE *outfile;
```

```
/*****************************************************************************************/
/* CELP_PC.MAK                                                                           */
/*****************************************************************************************/
PROJ   =CELP_PC
DEBUG  =1
CC  =qcl
AS  =qcl
CFLAGS_G   = /AL /W3 /Ze
CFLAGS_D   = /Zi /Zr /Od
CFLAGS_R   = /O /Ol /DNDEBUG
CFLAGS  =$(CFLAGS_G) $(CFLAGS_D)
AFLAGS_G   = /Cx /W2 /P1
AFLAGS_D   = /Zi
AFLAGS_R   = /DNDEBUG
AFLAGS  =$(AFLAGS_G) $(AFLAGS_D)
LFLAGS_G   = /CP:0xfff /NOI /NOE /SE:0x80 /ST:0x1000
LFLAGS_D   = /CO
LFLAGS_R   =
LFLAGS  =$(LFLAGS_G) $(LFLAGS_D)
RUNFLAGS   =
OBJS_EXT =
LIBS_EXT =

.asm.obj: ; $(AS) $(AFLAGS) -c $*.asm

all:   $(PROJ).EXE

celp_pc.obj:   celp_pc.c

pc.obj: pc.c

$(PROJ).EXE:   celp_pc.obj pc.obj $(OBJS_EXT)
    echo >NUL @<<$(PROJ).crf
celp_pc.obj +
pc.obj +
$(OBJS_EXT)
$(PROJ).EXE

$(LIBS_EXT);
<<
    ilink -a -e "link $(LFLAGS) @$(PROJ).crf" $(PROJ)

run: $(PROJ).EXE
    $(PROJ) $(RUNFLAGS)
```

PROGRAM RUNNING ON EVM:

(1) CELP.C CONTAINING EVM CONTROLLER
INCLUDE: C30_1.H (SHARE WITH C30.C)
CELP.H
CELP_CMD.H (SHARE WITH PC)

(2) C30.C CONTAINING HOST-EVM COMMUNICATION AND AIC CONTROLLER
INCLUDE: C30_1.H
C30_2.H

(3) INIT.ASM DEFINING AND INITALIZATION OF BUFFERS AND TABLES
(4) ANALYZE.C CELP ANALYSIS
INCLUDE: ANALYZE1.H
ANALYZE2.H
(5) LPC.C LPC ANALYSIS
(6) FIND_LSP.C CONSVERSIONS BETWEEN LPC AND LSP
(7) FIND_ACB.C ACB PARAMETER SEARCHING
(8) FIND_SCB.C SCB PARAMETER SEARCHING
(9) DO_TRANS.C UPDATING ACB AND TRACKING THE DELAY ELEMENTS
(10) STREAM.C CONVERTIONS BETWEEN CELP PARAMETERS AND DATA STREAM
(11) SYNTH.C CELP SYNTHESIS
INCLUDE: SYNTH1.H
SYNTH2.H

ALL PROGRAMS MUST BE LINKED TOGETHER.
COMPILATION FILE: MAKEFILE
LINKER FILE: CELP.CMD
MAP FILE: CELP.MAP

```
/*****************************************************************************/
/*   CELP.C                                                                  */
/*                                                                           */
/*   CODE ECXITED LINEAR PREDICTIVE CODING (CELP)                            */
/*   TMS320C30 EVALUATION MODULE APPLICATION                                 */
/*    :TMS320C30 CODE                                                        */
/*                                                                           */
/*  PROGRAMMED BY ARMEIN LANGI                                               */
/*  UNIVERSITY OF MANITOBA, CANADA                                           */
/*  @ 1990, 1991, 1992                                                       */
/*  TEMPLATES FOR COMMAND CONTROL AND EVM INTERFACING ARE                    */
/*   PROVIDED BY                                                             */
/*   TEXAS INSTRUMENTS                                                       */
/*    (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                    */
/*****************************************************************************/
#include "c30_1.h"    /* GENERIC SUPPORT MACROS & STRUCTURES                 */
#include "celp.h"     /* APPLICATION MACROS, GLOBALS, STRUCTURES,
                                 PROTOTYPES                                  */
#include "celp_cmd.h"  /* SHARED PC/TMS320C30 COMMAND PASSING MACROS        */
/*===========================================================*/
/* MAIN()                                                                    */
/*===========================================================*/
void main(void)
{
    init_evm();           /* INITIALIZE EVALUATION MODULE PARAMETERS         */
    init_arrays();        /* INITIALIZE SYSTEM DATA ARRAYS                   */
    init_aic();           /* INITIALIZE TLC32044 ANALOG INTERFACE CHIP       */
    asm("   STI SP,@_StackHolder");
    for(;;)
    {
        asm("   LDI @_StackHolder,SP");
        command_process();    /* DETECT AND PROCESS ANY HOST COMMANDS
    */
    }
}
/*===========================================================*/
/* C_INT05()                                                                 */
/*      SERIAL PORT 0 TRANSMIT INTERRUPT SERVICE ROUTINE                     */
/*      1. IF SECONDARY TRANSMISSION SEND AIC COMMAND WORD                   */
/*      2. OTHERWISE IF COMMAND SEND REQUESTED SETUP FOR SECONDARY           */
/*       TRANSMISSION ON NEXT INTERRUPT                                      */
/*      3. OTHERWISE WRITE OUT OUTPUT DATA AND READ IN INPUT DATA            */
/*      4. RESET SAMPLE INDEX IF FRAME IS FULL                               */
/*===========================================================*/
void c_int05(void)
{
/*
int *p;
*/
    if (secondary_transmit)
    {
        serial_port[0][X_DATA] = aic_secondary;
        secondary_transmit    = OFF;
    }
    else if (send_command)
    {
        serial_port[0][X_DATA] = 3;
```

```
        secondary_transmit    = ON;
        send_command          = OFF;
    }
    else
    {
        serial_port[0][X_DATA] = output[index] >> 2 << 2 ; /* << 2;                    */
        input[index] = serial_port[0][R_DATA] << 16 >> 16;

        if (++index >= 240 || index < 0)
        {
        index = 0;
/*    p = input;
        output = input;
        input = intermediate;
        intermediate = p;
*/
    }
    }
}
/*=================================================================*/
/* INIT_ARRAYS(): INITIALIZE DATA ARRAY PARAMETERS                 */
/*=================================================================*/
void init_arrays(void)
{
    /*-------------------------------------------------------------*/
    /* SET BLOCK SIZE FOR DATA TRANSFER TO THE PC                  */
    /*-------------------------------------------------------------*/
    buffer     = 240;
    /*-------------------------------------------------------------*/
    /* INITIALIZE AND ZERO FILL ARRAYS                             */
    /*-------------------------------------------------------------*/
}
/*=================================================================*/
/* WAIT_BUFFER(): WAIT FOR A NEW BUFFER OF DATA                    */
/*=================================================================*/
void wait_buffer(void)
{
int *p;
    /*-------------------------------------------------------------*/
    /* WAIT FOR ARRAY INDEX TO BE RESET TO ZERO BY C_INT05 ISR      */
    /*-------------------------------------------------------------*/
    while(index);
    /*-------------------------------------------------------------*/
    /* ROTATE DATA ARRAYS                                          */
    /*    INPUT ARRAY BECOMES DMA ACCESS ARRAY                     */
    /*    DMA ACCESS ARRAY BECOMES OUTPUT ARRAY                    */
    /*    OUTPUT ARRAY BECOMES INPUT ARRAY                         */
    /*-------------------------------------------------------------*/
 /* Moved to c_int05
*/
    p       = input;
    input     = output;
    output    = intermediate;
    intermediate = p;

    /*-------------------------------------------------------------*/
    /* WAIT FOR C_INT05 TO LOAD FIRST DATA POINT TO ASSURE THAT THE */
```

```
    /*    WAIT BUFFER ROUTINE TRIGGERS ONLY ONCE FOR EACH NEW BUFFER
    */
    /*-------------------------------------------------------------------------------------*/
    while(!index);
}
/*=====================================================================*/
/* COMMAND_PROCESS(): DETECT AND PROCESS A COMMAND FROM HOST          */
/*=====================================================================*/
void command_process(void)
{
    int i,j;          /* TEMPORARY VARIABLES                                    */
    while(dma[TRANSFER]);  /* WAIT FOR ANY DMA TRANSFER TO COMPLETE          */
    i = *host & 0xFF;     /* READ IN AND MASK HOST DATA REGISTER CONTENTS     */
    /*-------------------------------------------------------------------------------------*/
    /* IF THERE IS A COMMAND NOT RELATED TO DATA TRANSFER PARSE
COMMAND WORD */
    /*-------------------------------------------------------------------------------------*/
    if ((i && i <RECORDING)|| i > PLAYING)
    {
        switch(i)       /* PARSE COMMAND                                       */
    {
        case REC_CELP:
          wait_buffer();
          ReadAD(buffer240);
           Analyze(buffer240,buffer9);
          *host = REC_CELP;
          while(*host & 0xFF);        /* WAIT FOR HOST TO WITHDRAW REQUEST     */
          *host = NONE;             /* WITHDRAW ACKNOWLEDGE                    */
          break;
        case PLAY_CELP:
           Synthesize(buffer9,buffer240);
          WriteDA(buffer240);
          wait_buffer();
          *host= PLAY_CELP;
          while(*host & 0xFF);        /* WAIT FOR HOST TO WITHDRAW REQUEST     */
          *host = NONE;             /* WITHDRAW ACKNOWLEDGE                    */
          break;
        case ERASE_BUFFERS:
          EraseBuffers();
          *host=ERASE_BUFFERS;
          while(*host & 0xFF);        /* WAIT FOR HOST TO WITHDRAW REQUEST     */
          *host = NONE;             /* WITHDRAW ACKNOWLEDGE                    */
          break;
        case PCM2CELP:
           Analyze(buffer240,buffer9);
          *host=PCM2CELP;
          while(*host & 0xFF);        /* WAIT FOR HOST TO WITHDRAW REQUEST     */
          *host = NONE;             /* WITHDRAW ACKNOWLEDGE                    */
          break;
        case CELP2PCM:
           Synthesize(buffer9,buffer240);
          *host=CELP2PCM;
          while(*host & 0xFF);        /* WAIT FOR HOST TO WITHDRAW REQUEST     */
          *host = NONE;             /* WITHDRAW ACKNOWLEDGE                    */
          break;
        case INTRCELP:
          playing(9,buffer9,INTRCELP);
          break;
```

```
            case SYNPEECH:
              recording(240,buffer240,SYNPEECH);
              break;
            case OSPEECH:
              playing(240,buffer240,OSPEECH);
              SignAdjust(240,buffer240,buffer240);
              break;
            case OUTTRCELP:
              recording(9,buffer9,OUTTRCELP);
              break;
            default:
              while(*host & 0xFF);          /* WAIT FOR HOST TO WITHDRAW REQUEST      */
              *host = NONE;                 /* WITHDRAW ACKNOWLEDGE                   */
              break;

        }

        }
        /*-------------------------------------------------------------------------------*/
        /* RECORD OR PLAY IF COMMAND FROM HOST SO INDICATES                              */
        /*-------------------------------------------------------------------------------*/
        else
        {

        if (i == RECORDING)
        {
            wait_buffer();        /* WAIT FOR A NEW BUFFER OF DATA                       */
            recording(240,intermediate, RECORDING);
        }
        else if (i == PLAYING)
        {
            playing  (240,intermediate, PLAYING);
/*      SignAdjust(240,intermediate, intermediate);                                     */
            wait_buffer();
        }
        }
}

void SignAdjust(unsigned Size,int *InputBuffer, int *OutputBuffer)
{
unsigned i;
    for(i = 0; i < Size; i++)
    {
        OutputBuffer[i] = (InputBuffer[i] && 0xFFFF) << 16 >> 16 ;
    }
}
void ReadAD(int *buf)
{
unsigned i;
    for(i = 0; i < 240; i++)
    {
        buf[i] = intermediate[i];
    }
}
void WriteDA(int *buf)
{
unsigned i;
```

```
    for(i = 0; i < 240; i++)
    {
        intermediate[i] = buf[i] && 0xffff;
    }
}
```

```
/*****************************************************************************************/
/*  CELP.H                                                                               */
/*                                                                                       */
/*  TMS320C30 EVALUATION MODULE DATA ACQUISITION PROGRAM                                 */
/*     :TMS320C30 EXTERNALS, GLOBALS, MACROS, PROTOTYPES DEFINES                         */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                  */
/*****************************************************************************************/
/* FUNCTION PROTOTYPES                                                                   */
/*=======================================================================*/
void main(void);
void c_int05(void);
void init_arrays(void);
void wait_buffer(void);
void command_process(void);
void SignAdjust(unsigned k, int *b1, int *b2);
void ReadAD(int *buf);
void WriteDA(int *buf);
extern void synthesizer(int *buffer9, int *buffer240);
extern void analyze(int *buffer240, int *buffer9);
extern void erase_buffers(void);


/*=======================================================================*/
/* GLOBAL VARIABLES                                                      */
/*=======================================================================*/
extern unsigned StackHolder;
extern int *output;              /* POINTER TO OUTPUT DATA ARRAY         */
extern int *input;               /* POINTER TO INPUT DATA ARRAY          */
extern int *intermediate;        /* POINTER TO DATA ARRAY FOR DMA ACCESS */
int buffer9[9];
int buffer240[240];
volatile int index = 0;      /* INDEX INTO INPUT AND OUTPUT DATA ARRAYS  */
/*=======================================================================*/
/* GLOBAL VARIABLES                                                      */
/*=======================================================================*/
extern int buffer;               /* block size for PC data transfer      */
/*-----------------------------------------------------------------------*/
/* AIC CONTROL VARIABLES                                                 */
/*-----------------------------------------------------------------------*/
extern AIC_COMMAND_0  aic_command_0;          /* AIC COMMAND WORD 0      */
extern AIC_COMMAND_1  aic_command_1;          /* AIC COMMAND WORD 1      */
extern AIC_COMMAND_2  aic_command_2;          /* AIC COMMAND WORD 2      */
extern AIC_COMMAND_3  aic_command_3;          /* AIC COMMAND WORD 3      */
extern volatile int send_command ;     /* FLAG TO SEND AIC COMMAND WORD  */
extern volatile int secondary_transmit; /* FLAG TO SEND SECONDARY TRANSMIT*/
extern int aic_secondary;       /* COMMAND TO SEND ON SECONDARY TRANSMIT*/
/*****************************************************************************************/
/*      TMS320C30 CONTROL LOCATIONS                                                      */
/*****************************************************************************************/
/*-----------------------------------------------------------------------*/
/* SERIAL PORT BASE LOCATION                                             */
/*-----------------------------------------------------------------------*/
extern volatile int (*serial_port)[16];
/*-----------------------------------------------------------------------*/
/* DMA BASE LOCATION                                                     */
/*-----------------------------------------------------------------------*/
```

```
extern volatile int *dma;
/*-------------------------------------------------------------------------------------------*/
/* HOST COMMUNICATION REGISTER LOCATION                                                      */
/*-------------------------------------------------------------------------------------------*/
extern volatile int *host;
```

```
/*****************************************************************************/
/*  C30.C                                                                    */
/*                                                                           */
/*  TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT PROGRAMS                   */
/*                                                                           */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                      */
/*****************************************************************************/
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "c30_1.h"          /* FUNCTION PROTOTYPES, MACROS, STRUCTURES       */
#include "c30_2.h"          /* GLOBAL VARIABLES                              */
/*===========================================================================*/
/* C_INT11(): DMA INTERRUPT SERVICE ROUTINE                                  */
/*===========================================================================*/
void c_int11(void)
{
    while(*host & 0xFF);          /* WAIT FOR HOST TO WITHDRAW REQUEST       */
    *host     = NONE;             /* ACKNOWLEDGE WITHDRAWL OF REQUEST        */
    dma[GLOBAL] = 0;              /* TURN OFF DMA                            */

    asm(" AND 0FFFFh,IE");        /* TURN OFF DMA INTERRUPTS                 */
}
/*===========================================================================*/
/* C_INT99(): ERRONEOUS INTERRUPT SERVICE ROUTINE                            */
/*        THIS ROUTINE IDLES AFTER RECEIVING AN UNEXPECTED INTERRUPT         */
/*===========================================================================*/
void c_int99(void)
{

}
/*===========================================================================*/
/* INIT_EVM(): INITIALIZE TMS320C30 EVALUATION MODULE                        */
/*===========================================================================*/
void init_evm(void)
{
    bus[EXPANSION] = 0x0;         /* ZERO WAIT STATES ON EXPANSION BUS       */
    bus[PRIMARY]   = 0x0;         /* ZERO WAIT STATES ON PRIMARY BUS         */
    *host        = NONE;          /* CLEAR ANY ACKNOWLEDGEMENT TO HOST       */

    asm(" OR 800h,ST");           /* TURN ON CACHE                           */
}
/*===========================================================================*/
/* INIT_AIC(): INITIALIZE COMMUNICATIONS TO AIC                              */
/*          NOTE: i IS A VOLATILE TO FORCE TIME DELAYS AND TO FORCE          */
/*             READS OF SERIAL PORT DATA RECEIVE REGISTER TO CLEAR           */
/*           THE RECEIVE INTERRUPT FLAG                                      */
/*===========================================================================*/
void init_aic(void)
{
    volatile int i;
    /*-----------------------------------------------------------------------*/
    /* SET AIC CONFIGURATION CHIP                                            */
    /*  1. ALLOW 8 KHZ SAMPLING RATE AND 3.6 KHZ ANTIALIASING FILTER         */
    /*     GIVEN A 7.5 MHZ MCLK TO THE AIC FROM A 30 MHZ TMS320C30           */
    /*  2. ENABLE A/D HIGHPASS FILTER                                        */
    /*  3. SET SYNCHRONOUS TRANSMIT AND RECEIVE                              */
```

```
/* 4. ENABLE SINX/X D/A CORRECTION FILTER                                    */
/* 5. SET AIC FOR +/- 1.5 V INPUT                                            */
/*-------------------------------------------------------------------------*/
aic_command_0.command = 0;     /* SETUP AIC COMMAND WORD ZERO                */
aic_command_0.ra      = 13;    /* ADJUST SAMPLING RATE TO 8 kHz              */
aic_command_0.ta      = 13;    /*   AND 3.6 kHz ANTIALIAS FILTER             */
aic_command_1.command = 1;     /* SETUP DEFAULT AIC COMMAND WORD 1           */
aic_command_1.ra_prime = 1;
aic_command_1.ta_prime = 1;
aic_command_1.d_f     = 0;
aic_command_2.command = 2;     /* SETUP DEFAULT AIC COMMAND WORD 2           */
aic_command_2.rb      = 36;
aic_command_2.tb      = 36;
aic_command_3.command = 3;
aic_command_3.highpass = ON;   /* TURN ON INPUT HIGHPASS FILTER              */
aic_command_3.loopback = OFF;  /* DISABLE AIC LOOPBACK                       */
aic_command_3.aux     = OFF;   /* DISABLE AUX INPUT                          */
aic_command_3.sync    = ON;    /* ENABLE SYNCHRONOUS A/D AND D/A             */
aic_command_3.gain    = LINE_V; /* SET FOR LINE-LEVEL INPUT                  */
aic_command_3.sinx    = ON;    /* ENABLE SIN x/x CORRECTION FILTER           */
/*-------------------------------------------------------------------------*/
/* CONFIGURE TIMER 0 TO ACT AS AIC MCLK                                      */
/* THE TIMER IS CONFIGURED IN THE FOLLOWING WAY                              */
/*      1. THE TIMER'S VALUE DRIVES AN ACTIVE-HIGH TCLK 0 PIN                */
/*      2. THE TIMER IS RESET AND ENABLED                                    */
/*      3. THE TIMER'S IS IN PULSE MODE                                      */
/*      4. THE TIMER HAS A PERIOD OF TWO INSTRUCTION CYCLES                  */
/*-------------------------------------------------------------------------*/
timer[0][PERIOD] = 0x1;
timer[0][GLOBAL] = 0x2C1;
/*-------------------------------------------------------------------------*/
/* CONFIGURE SERIAL PORT 0                                                   */
/*       1. EXTERNAL FSX, FSR, CLKX, CLKR                                    */
/*       2. VARIABLE DATA RATE TRANSMIT AND RECEIVE                          */
/*       3. HANDSHAKE DISABLED                                               */
/*       4. ACTIVE HIGH DATA AND CLK                                         */
/*       5. ACTIVE LOW FSX,FSR                                               */
/*       6. 16 BIT TRANSMIT AND RECEIVE WORD                                 */
/*       7. TRANSMIT INTERRUPT                                               */
/*       8. RECEIVE INTERRUPT ENABLED/RECEIVE                                */
/*       9. FSX, FSR, CLKX, CLKR, DX, DR CONFIGURED AS SERIAL                */
/*        PORT PINS                                                          */
/*-------------------------------------------------------------------------*/
serial_port[0][X_PORT] = 0x111;
serial_port[0][R_PORT] = 0x111;

asm(" LDI 2,IOF");       /* RESET AIC BY PULLING XF0 ACTIVE-LOW             */

for(i = 0; i < 50; i++);  /* KEEP RESET LOW FOR SOME PERIOD OF TIME         */
serial_port[0][GLOBAL] = 0x0e970300; /* WRITE SERIAL PORT CONTROL           */
serial_port[0][X_DATA] = 0x0;      /* CLEAR SERIAL TRANSMIT DATA            */

asm(" LDI 6,IOF");            /* PULL AIC OUT OF RESET                     */

asm(" LDI   0,IF"); /* CLEAR ANY INTERRUPT FLAGS                           */
asm(" LDI  410h,IE"); /* ENABLE DMA & SERIAL PORT 0 TRANSMIT INTERRUPTS*/
asm(" OR  2000h,ST"); /* SET GLOBAL INTERRUPT ENABLE BIT                     */
```

```
    /*-------------------------------------------------------------------------*/
    /* MODIFY AIC CONFIGURATION                                                */
    /*-------------------------------------------------------------------------*/
    configure_aic(*((int *) &aic_command_0));
    configure_aic(*((int *) &aic_command_3));
}
/*=========================================================================*/
/* CONFIGURE_AIC(): INITIATE AIC CONFIGURATION WORD TRANSMISSION ON NEXT
   */
/*          INTERRUPT AFTER ALL PREVIOUS COMMANDS ARE SENT                 */
/*=========================================================================*/
void configure_aic(int i)
{
    while(send_command || secondary_transmit);
    aic_secondary = i;
    send_command = ON;
}
/*=========================================================================*/
/* RECORDING(): SEND DATA FOR STORAGE ON PC                                */
/*=========================================================================*/
void recording(unsigned buffsize,int *source,int rec_cmd)
{
    asm(" AND   0FFF8h,IF");  /* CLEAR ANY PENDING HOST INTERRUPTS*/
    asm(" OR  @_dma_int2,IE");  /* SYNCHRONIZE DMA WITH PC READ INT        */

    dma[SOURCE]  = (int) source;    /* READ FROM CODEWORD                  */
    dma[DEST]    = (int) host;      /* WRITE TO HOST                       */
    dma[TRANSFER] = buffsize;       /* TRANSFER ENTIRE STRUCTURE           */
    /*-------------------------------------------------------*/
    /* SETUP DMA FOR RECORDING                                             */
    /*    1. INCREMENT SOURCE ADDRESS ONLY ON EACH TRANSFER                */
    /*    2. SYNCHRONIZE WRITES WITH DATA READ INTERRUPT FROM PC           */
    /*     FROM PC                                                         */
    /*    3. TRANSFER COUNTER STOPS UPON REACHING ZERO                     */
    /*    4. THE DMA INTERRUPT SOURCE IS ENABLED                           */
    /*-------------------------------------------------------*/
    dma[GLOBAL] = 0x0E13;
    *host     = rec_cmd;        /* ACKNOWLEDGE REQUEST                     */
}
/*=========================================================================*/
/* PLAYING(): RECEIVE RECORDED DATA FROM PC                                */
/*=========================================================================*/
void playing(unsigned buffsize,int *dest,int play_cmd)
{
    asm(" AND   0FFF8h,IF");  /* CLEAR ANY PENDING HOST INTERRUPTS*/
    asm(" OR  @_dma_int1,IE");  /* SYNCHRONIZE DMA WITH PC WRITE INT*/

    dma[SOURCE]  = (int) host;      /* READ FROM HOST                      */
    dma[DEST]    = (int) dest;      /* WRITE TO CODEWORD                   */
    dma[TRANSFER] = buffsize;       /* TRANSFER ENTIRE STRUCTURE           */
    /*-------------------------------------------------------*/
    /* SETUP DMA FOR PLAYING                                               */
    /*   1. INCREMENT DESTINATION ADDRESS ONLY ON EACH TRANSFER            */
    /*   2. SYNCHRONIZE WRITES WITH DATA WRITE INTERRUPT FROM PC           */
    /*   3. TRANSFER COUNTER STOPS UPON REACHING ZERO                      */
    /*   4. THE DMA INTERRUPT SOURCE IS ENABLED                            */
    /*-------------------------------------------------------*/
    dma[GLOBAL] = 0x0D43;
```

```
    *host     = play_cmd;          /* ACKNOWLEDGE REQUEST                    */
}
```

```
/*****************************************************************************************/
/*  C30_1.H                                                                              */
/*                                                                                       */
/*  TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT FILE 1                                 */
/*    : FUNCTION PROTOTYPES, MACROS, STRUCTURES                                          */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                  */
/*****************************************************************************************/
/* FUNCTION PROTOTYPES*=================================================*/
/*=====================================================================*/
void c_int11(void);
void c_int99(void);
void init_evm(void);
void init_aic(void);
void configure_aic(int i);
void recording(unsigned buffsize,int *source, int rec_cmd);
void playing(unsigned buffsize,int *dest, int play_cmd);

/*=====================================================================*/
/* MACROS
*=====================================================================*/
/*=====================================================================*/
#define OFF    0x00
#define ON     0x01
#define TOGGLE 0x01          /* BIT-MASK TO TOGGLE BETWEEN ON AND OFF        */
#define NONE   0x00
/*=====================================================================*/
/*       TMS320C30 MEMORY-MAPPED CONTROL REGISTER INDICES                */
/*=====================================================================*/
/*---------------------------------------------------------------------------*/
/* BASE GLOBAL CONTROL REGISTER                                              */
/*---------------------------------------------------------------------------*/
#define GLOBAL 0
/*---------------------------------------------------------------------------*/
/* EXTERNAL BUS CONTROL REGISTERS                                            */
/*---------------------------------------------------------------------------*/
#define EXPANSION 0                  /* EXPANSION BUS                        */
#define PRIMARY  4                   /* PRIMARY BUS                          */
/*---------------------------------------------------------------------------*/
/* SERIAL PORT CONTROL REGISTERS                                             */
/*---------------------------------------------------------------------------*/
#define X_PORT  2                    /* TRANSMIT CONTROL                     */
#define R_PORT  3                    /* RECEIVE  CONTROL                     */
#define X_DATA  8                    /* TRANSMIT DATA                        */
#define R_DATA 12                    /* RECEIVE  DATA                        */
/*---------------------------------------------------------------------------*/
/* TIMER CONTROL REGISTERS                                                   */
/*---------------------------------------------------------------------------*/
#define PERIOD  8                    /* PERIOD REGISTER                      */
/*---------------------------------------------------------------------------*/
/* DMA CONTROL REGISTERS                                                     */
/*---------------------------------------------------------------------------*/
#define SOURCE  4                    /* SOURCE ADDRESS REGISTER              */
#define DEST    6                    /* DESTINATION ADDRESS REGISTER         */
#define TRANSFER 8                   /* TRANSFER COUNTER REGISTER            */
/*---------------------------------------------------------------------------*/
/* AIC VOLTAGE INPUT CONTROL MACROS                                          */
```

```
/*-------------------------------------------------------------------------------------*/
#define THREE_V   1
#define LINE_V    2

/*=============================================================
======*/
/*
STRUCTURES*=================================================
====*/
/*=============================================================
======*/
/*  AIC COMMAND WORD BITFIELD ENCODING STRUCTURES                    */
/*=============================================================
======*/
typedef struct
{
    unsigned int command :2;    /* COMMAND BITS --- SHOULD BE SET TO 00*/
    unsigned int ra      :5;    /* RECEIVE COUNTER A LOAD VALUE        */
    unsigned int d_78    :2;    /* UNUSED - DON'T CARES                */
    unsigned int ta      :5;    /* TRANSMIT COUNTER A LOAD VALUE       */
    unsigned int d_ef    :2;    /* UNUSED - DON'T CARES                */
} AIC_COMMAND_0;

typedef struct
{
    unsigned int command  :2;   /* COMMAND BITS --- SHOULD BE SET TO 01*/
      signed int ra_prime :6;   /* RECEIVE COUNTER DELTA A' LOAD VALUE */
    unsigned int d_8      :1;   /* UNUSED - DON'T CARES                */
      signed int ta_prime :6;   /* RECEIVE COUNTER DELTA A' LOAD VALUE */
    unsigned int d_f      :1;   /* UNUSED - DON'T CARES                */
} AIC_COMMAND_1;

typedef struct
{
    unsigned int command :2;    /* COMMAND BITS --- SHOULD BE SET TO 10*/
    unsigned int rb      :6;    /* RECEIVE COUNTER B LOAD VALUE        */
    unsigned int d_8     :1;    /* UNUSED - DON'T CARES                */
    unsigned int tb      :6;    /* TRANSMIT COUNTER B LOAD VALUE       */
    unsigned int d_f     :1;    /* UNUSED - DON'T CARES                */
} AIC_COMMAND_2;

typedef struct
{
    unsigned int command  :2;   /* COMMAND BITS --- SHOULD BE SET TO 11*/
    unsigned int highpass :1;   /* HIGHPASS FILTER ENABLE              */
    unsigned int loopback :1;   /* LOOPBACK TEST ENABLE                */
    unsigned int aux      :1;   /* AUX INPUT ENABLE                    */
    unsigned int sync     :1;   /* SYNCHRONOUS TRANSMIT/RECEIVE ENABLE */
    unsigned int gain     :2;   /* GAIN SELECTION BITS                 */
    unsigned int d_8      :1;   /* UNUSED - DON'T CARES                */
    unsigned int sinx     :1;   /* SINX/X CORRECTION FILTER ENABLE     */
    unsigned int d_abcdef :1;   /* UNUSED - DON'T CARES                */
} AIC_COMMAND_3;
```

```
/*****************************************************************************************/
/*                                                                                       */
/*  C30_2.H                                                                              */
/*                                                                                       */
/*  TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT FILE 2:                                */
/*    : GLOBAL VARIABLES                                                                 */
/*                                                                                       */
/*                                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                                  */
/*                                                                                       */
/*****************************************************************************************/
/* GLOBAL VARIABLES                                                                      */
/*=======================================================================================*/
int buffer;                     /* block size for PC data transfer                       */
/*---------------------------------------------------------------------------------------*/
/* 32-BIT DMA INTERRUPT ENABLE MASKS                                                     */
/*---------------------------------------------------------------------------------------*/
int dma_int0 = 0x00010000;                  /* INT0                                      */
int dma_int1 = 0x00020000;                  /* INT1                                      */
int dma_int2 = 0x00040000;                  /* INT2                                      */
/*---------------------------------------------------------------------------------------*/
/* AIC CONTROL VARIABLES                                                                 */
/*---------------------------------------------------------------------------------------*/
AIC_COMMAND_0 aic_command_0;            /* AIC COMMAND WORD 0                            */
AIC_COMMAND_1 aic_command_1;            /* AIC COMMAND WORD 1                            */
AIC_COMMAND_2 aic_command_2;            /* AIC COMMAND WORD 2                            */
AIC_COMMAND_3 aic_command_3;            /* AIC COMMAND WORD 3                            */
volatile int send_command     = OFF;  /* FLAG TO SEND AIC COMMAND WORD                   */
volatile int secondary_transmit = OFF;  /* FLAG TO SENT SECONDARY TRANSMIT*/
int aic_secondary = 0;          /* COMMAND TO SENT ON SECONDARY TRANSMIT                 */
/*****************************************************************************************/
/*      TMS320C30 CONTROL LOCATIONS                                                      */
/*****************************************************************************************/
/*---------------------------------------------------------------------------------------*/
/* SERIAL PORT BASE LOCATION                                                             */
/*---------------------------------------------------------------------------------------*/
volatile int (*serial_port)[16] = (volatile int (*)[16]) 0x808040;
/*---------------------------------------------------------------------------------------*/
/* TIMER BASE LOCATION                                                                   */
/*---------------------------------------------------------------------------------------*/
volatile int (*timer)[16] = (volatile int (*)[16]) 0x808020;
/*---------------------------------------------------------------------------------------*/
/* BUS BASE LOCATION                                                                     */
/*---------------------------------------------------------------------------------------*/
volatile int *bus = (volatile int *) 0x808060;
/*---------------------------------------------------------------------------------------*/
/* DMA BASE LOCATION                                                                     */
/*---------------------------------------------------------------------------------------*/
volatile int *dma = (volatile int *) 0x808000;
/*---------------------------------------------------------------------------------------*/
/* HOST COMMUNICATION REGISTER LOCATION                                                  */
/*    NOTE: ONLY THE PC HOST CAN READ WHAT THE TMS320C30 WRITES.                         */
/*        CONVERSELY ONLY THE TMS320C30 CAN READ WHAT THE HOST PC                        */
/*      WRITES.                                                                          */
/*---------------------------------------------------------------------------------------*/
volatile int *host = (volatile int *) 0x804000;
```

```
/*****************************************************************************************************/
/* INIT.ASM                                                                                          */
/* EVM PROGRAM FOR DEFINING AND INITIALIZATION BUFFERS AND TABLES                                    */
/* PROGRAMMED BY ARMEIN LANGI
        */
/* UNIVERSITY OF MANITOBA                                                                            */
/* @ 1990, 1991, 1992                                                                                */
/*****************************************************************************************************/
FP    .set  AR3
      .global _SCB
      .global _ACB
      .global _ACBGainTable
      .global _SCBGainTable
      .global _BandwidthExpandingFactor
      .global _HammingWindowData
      .global _TableLSP
      .global _DelayElements
      .global _OriginalSpeech
      .global _CELP
      .global _LPC
      .global _PreviousLSP
      .global _FloatLSPP
      .global _FloatLSPQ
      .global _ExpandedLPC
      .global _ACBSearchingTarget
      .global _SCBSearchingTarget
      .global _StackHolder
      .global _WeightedImpulseResponse
      .global _ExcitationPulses
      .global _SyntheticSpeech

      .data
_StackHolder
      .word  0
*********************************************************
      .global _SynthDelayElements
_SynthDelayElements
      .space  60
*********************************************************
      .global _SynthPreviousLSP
_SynthPreviousLSP
      .space  10
*********************************************************
      .global _SynthLPC
_SynthLPC
      .space  11
*********************************************************
      .global _SynthFloatLSPP
_SynthFloatLSPP
      .space  6
*********************************************************
      .global _SynthFloatLSPQ
_SynthFloatLSPQ
      .space  6
*********************************************************
      .global _SynthACB
_SynthACB
```

```
    .space  148
*********************************************************
    .global _SynthExcitationPulses
_SynthExcitationPulses
    .space  60
*********************************************************
    .global _SynthCELP
_SynthCELP
    .space  26
*********************************************************

    .text
*********************************************************
    .globl  _PCMToFloat
;>>>>  void PCMToFloat(int *pcm, float *Float)
;>>>>  unsigned i;
*********************************************************
* FUNCTION DEF : _PCMToFloat
*********************************************************
_PCMToFloat:
    PUSH   FP
    LDI    SP,FP

    LDI    *-FP(3),AR1
    LDI    *-FP(2),AR0
    LDI    239,RC
    RPTB   PTFLoop1
    LDI    *AR0++(1),R0
    LSH    16,R0
    ASH    -16,R0
    FLOAT  R0
PTFLoop1      STF    R0,*AR1++(1)


    LDI    *-FP(1),R1
    BD     R1
    LDI    *FP,FP
    NOP
    SUBI   2,SP
***   B      R1     ;BRANCH OCCURS

    .data
PTFConstant
    .float  1.52587890625e-4 ;1
    .text

*********************************************************
    .global _ReturnInteger
_ReturnInteger:
    PUSH   FP
    LDI    SP,FP

    LDI    *-FP(3),AR1
    LDI    *-FP(2),AR0
    LDI    8,RC
    RPTB   RILoop1
    LDI    *AR0++(1),R0
```

```
RILoop1
    STI   R0,*AR1++(1)

    LDI   *-FP(1),R1
    BD    R1
    LDI   *FP,FP
    NOP
    SUBI  2,SP
***  B    R1    ;BRANCH OCCURS

*********************************************************
    .global _ReturnFloat
_ReturnFloat:
    PUSH  FP
    LDI   SP,FP
    LDI   *-FP(3),AR1
    LDI   *-FP(2),AR0
    LDF   @IntegerToFloatConstant,R1
    LDI   8,RC
    RPTB  RFLoop1
    LDF   *AR0++(1),R0
    MPYF  R1,R0
    FIX R0,R0
RFLoop1
    STI   R0,*AR1++(1)

    LDI   *-FP(1),R1
    BD    R1
    LDI   *FP,FP
    NOP
    SUBI  2,SP
***  B    R1    ;BRANCH OCCURS
    .data
IntegerToFloatConstant
    .float  6553.6
    .text
*********************************************************
    .global _EraseBuffers
_EraseBuffers:
    PUSH  FP
    PUSH  DP

    LDP   _ACB,DP

    LDI   @ACBAddr,AR0
    LDF   0.00,R0
    LDI   146,RC
    RPTB  EBLoop1
EBLoop1 STF   R0,*AR0++(1)

    LDI   @OriginalSpeechAddr,AR0
    LDI   359,RC
    RPTB  EBLoop2
EBLoop2 STF   R0,*AR0++(1)

    LDI   @DelayElementsAddr,AR0
    LDI   9,RC
    RPTB  EBLoop3
```

```
EBLoop3 STF    R0,*AR0++(1)

    LDI @PreviousLSPAddr,AR0
    LDI 0,R0
    LDI 9,RC
    RPTB   EBLoop4
EBLoop4 STF R0,*AR0++(1)

    POP   DP
    POP   FP
    RETSU
    .data
ACBAddr .word  _ACB
OriginalSpeechAddr
    .word  _OriginalSpeech
DelayElementsAddr
    .word  _DelayElements
PreviousLSPAddr
    .word  _PreviousLSP
    .text
*********************************************************
    .global _InitializeAllParameters

_InitializeAllParameters:
    NOP
    RETSU

    .data
*********************************************************
_PreviousLSP
    .space 10
*********************************************************
_CELP  .space 26
*********************************************************
_LPC   .space 11
*********************************************************
_DelayElements
    .space 11
*********************************************************
_OriginalSpeech
    .space 360
*********************************************************
_FloatLSPP
    .space 6
*********************************************************
_FloatLSPQ
    .space 6
*********************************************************
_ExpandedLPC
    .space 11
*********************************************************
_ACBSearchingTarget
    .space 60
*********************************************************
_SCBSearchingTarget
    .space 60
*********************************************************
```

```
_WeightedImpulseResponse
    .space  60
*********************************************************
_ExcitationPulses
    .space  60
*********************************************************
_SyntheticSpeech
    .space  60
*********************************************************
_HammingWindowData
        .float  0.080000
    .float  0.080158
    .float  0.080630
    .float  0.081418
    .float  0.082520
    .float  0.083935
    .float  0.085663
    .float  0.087703
    .float  0.090052
    .float  0.092710
    .float  0.095674
    .float  0.098943
    .float  0.102514
    .float  0.106385
    .float  0.110553
    .float  0.115015
    .float  0.119769
    .float  0.124811
    .float  0.130137
    .float  0.135744
    .float  0.141628
    .float  0.147786
    .float  0.154212
    .float  0.160902
    .float  0.167852
    .float  0.175057
    .float  0.182513
    .float  0.190213
    .float  0.198153
    .float  0.206328
    .float  0.214731
    .float  0.223357
    .float  0.232200
    .float  0.241254
    .float  0.250513
    .float  0.259970
    .float  0.269619
    .float  0.279453
    .float  0.289466
    .float  0.299651
    .float  0.310000
    .float  0.320507
    .float  0.331164
    .float  0.341965
    .float  0.352901
    .float  0.363966
    .float  0.375151
```

```
.float  0.386449
.float  0.397852
.float  0.409353
.float  0.420943
.float  0.432615
.float  0.444361
.float  0.456172
.float  0.468040
.float  0.479958
.float  0.491917
.float  0.503909
.float  0.515925
.float  0.527959
.float  0.540000
.float  0.552041
.float  0.564075
.float  0.576091
.float  0.588083
.float  0.600042
.float  0.611960
.float  0.623828
.float  0.635639
.float  0.647385
.float  0.659057
.float  0.670647
.float  0.682148
.float  0.693551
.float  0.704849
.float  0.716034
.float  0.727099
.float  0.738035
.float  0.748836
.float  0.759493
.float  0.770000
.float  0.780349
.float  0.790534
.float  0.800547
.float  0.810381
.float  0.820030
.float  0.829487
.float  0.838746
.float  0.847800
.float  0.856643
.float  0.865269
.float  0.873672
.float  0.881847
.float  0.889787
.float  0.897487
.float  0.904943
.float  0.912148
.float  0.919098
.float  0.925788
.float  0.932214
.float  0.938372
.float  0.944256
.float  0.949863
.float  0.955189
.float  0.960231
```

```
.float  0.964985
.float  0.969447
.float  0.973615
.float  0.977486
.float  0.981057
.float  0.984326
.float  0.987290
.float  0.989948
.float  0.992297
.float  0.994337
.float  0.996065
.float  0.997480
.float  0.998582
.float  0.999370
.float  0.999842
.float  1.000000
.float  0.999842
.float  0.999370
.float  0.998582
.float  0.997480
.float  0.996065
.float  0.994337
.float  0.992297
.float  0.989948
.float  0.987290
.float  0.984326
.float  0.981057
.float  0.977486
.float  0.973615
.float  0.969447
.float  0.964985
.float  0.960231
.float  0.955189
.float  0.949863
.float  0.944256
.float  0.938372
.float  0.932214
.float  0.925788
.float  0.919098
.float  0.912148
.float  0.904943
.float  0.897487
.float  0.889787
.float  0.881847
.float  0.873672
.float  0.865269
.float  0.856643
.float  0.847800
.float  0.838746
.float  0.829487
.float  0.820030
.float  0.810381
.float  0.800547
.float  0.790534
.float  0.780349
.float  0.770000
.float  0.759493
```

```
.float  0.748836
.float  0.738035
.float  0.727099
.float  0.716034
.float  0.704849
.float  0.693551
.float  0.682148
.float  0.670647
.float  0.659057
.float  0.647385
.float  0.635639
.float  0.623828
.float  0.611960
.float  0.600042
.float  0.588083
.float  0.576091
.float  0.564075
.float  0.552041
.float  0.540000
.float  0.527959
.float  0.515926
.float  0.503909
.float  0.491917
.float  0.479958
.float  0.468040
.float  0.456172
.float  0.444361
.float  0.432615
.float  0.420943
.float  0.409353
.float  0.397852
.float  0.386449
.float  0.375151
.float  0.363966
.float  0.352901
.float  0.341965
.float  0.331164
.float  0.320507
.float  0.310000
.float  0.299651
.float  0.289466
.float  0.279453
.float  0.269619
.float  0.259970
.float  0.250513
.float  0.241254
.float  0.232200
.float  0.223357
.float  0.214731
.float  0.206328
.float  0.198153
.float  0.190213
.float  0.182513
.float  0.175057
.float  0.167852
.float  0.160902
.float  0.154212
```

```
    .float  0.147786
    .float  0.141628
    .float  0.135744
    .float  0.130137
    .float  0.124811
    .float  0.119769
    .float  0.115015
    .float  0.110553
    .float  0.106385
    .float  0.102514
    .float  0.098943
    .float  0.095674
    .float  0.092710
    .float  0.090052
    .float  0.087703
    .float  0.085663
    .float  0.083935
    .float  0.082520
    .float  0.081418
    .float  0.080630
    .float  0.080158
*******************************************************************************

_TableLSP
        .float 0.996917
        .float 0.991100
        .float 0.984427
        .float 0.980785
        .float 0.975917
        .float 0.964557
        .float 0.946085
        .float 0.923880
        .space  8
lsp2        .float 0.986429
        .float 0.983016
        .float 0.978419
        .float 0.973279
        .float 0.967599
        .float 0.960294
        .float 0.951057
        .float 0.940881
        .float 0.929776
        .float 0.917755
        .float 0.904827
        .float 0.887413
        .float 0.864713
        .float 0.835807
        .float 0.804376
        .float 0.770513
lsp3        .float 0.946085
        .float 0.935444
        .float 0.923880
        .float 0.911403
        .float 0.896293
        .float 0.876307
        .float 0.850582
        .float 0.820401
        .float 0.785317
```

```
        .float 0.734323
        .float 0.678801
        .float 0.619094
        .float 0.555570
        .float 0.488621
        .float 0.418660
        .float 0.346117
lsp4    .float 0.883766
        .float 0.868632
        .float 0.844328
        .float 0.811319
        .float 0.770513
        .float 0.723570
        .float 0.661312
        .float 0.606682
        .float 0.542442
        .float 0.474856
        .float 0.404344
        .float 0.331338
        .float 0.256289
        .float 0.179661
        .float 0.101925
        .float 0.023560
lsp5    .float 0.707107
        .float 0.678801
        .float 0.631353
        .float 0.581413
        .float 0.532507
        .float 0.488621
        .float 0.432873
        .float 0.375416
        .float 0.316477
        .float 0.256289
        .float 0.195090
        .float 0.117537
        .float 0.039260
        .float -0.039260
        .float -0.117537
        .float -0.195090

lsp6    .float 0.404344
        .float 0.331338
        .float 0.241075
        .float 0.133121
        .float 0.000000
        .float -0.156434
        .float -0.309017
        .float -0.453990
        .space 8

lsp7    .float 0.156435
        .float 0.094108
        .float 0.031411
        .float -0.078459
        .float -0.233445
        .float -0.368124
        .float -0.522498
```

```
        .float -0.649448
        .space 8

lsp8        .float -0.175796
        .float -0.309017
        .float -0.400749
        .float -0.488621
        .float -0.587785
        .float -0.678801
        .float -0.785317
        .float -0.872496
        .space  8

lsp9        .float -0.562083
        .float -0.637424
        .float -0.707107
        .float -0.760406
        .float -0.809017
        .float -0.856717
        .float -0.901455
        .float -0.938191
        .space  8

lsp10       .float -0.804376
        .float -0.840093
        .float -0.872496
        .float -0.898028
        .float -0.920845
        .float -0.948600
        .float -0.974173
        .float -0.991100
        .space 8

*****************************************************************************
_ACBGainTable
                .float -0.993000
        .float -0.831000
        .float -0.693000
        .float -0.555000
        .float -0.414000
        .float -0.229000
        .float 0.000000
        .float 0.193000
        .float 0.255000
        .float 0.368000
        .float 0.457000
        .float 0.531000
        .float 0.601000
        .float 0.653000
        .float 0.702000
        .float 0.745000
        .float 0.780000
        .float 0.816000
        .float 0.850000
        .float 0.881000
        .float 0.915000
        .float 0.948000
        .float 0.983000
```

```
        .float 1.020000
        .float 1.062000
        .float 1.117000
        .float 1.193000
        .float 1.289000
        .float 1.394000
        .float 1.540000
        .float 1.765000
        .float 1.991000

*****************************************************************************
_SCBGainTable
                .float -1300
        .float -870
        .float -660
        .float -520
        .float -418
        .float -340
        .float -278
        .float -224
        .float -178
        .float -136
        .float -98
        .float -64
        .float -35
        .float -13
        .float -3
        .float -1
        .float 1
        .float 3
        .float 13
        .float 35
        .float 64
        .float 98
        .float 136
        .float 178
        .float 224
        .float 278
        .float 340
        .float 418
        .float 520
        .float 660
        .float 870
        .float 1330
*****************************************************************************
_SCB    .float  0
    .float  0
    .float  0
    .float  0
    .float  -1
    .float  0
    .float  -1
    .float  0
    .float  0
    .float  0
    .float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  1
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  1
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  1
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  1
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  1
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  -1
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  -1
.float  -1
.float  -1
.float  0
.float  0
.float  0
.float  0
.float  1
.float  0
.float  0
.float  0
.float  0
.float  0
.float  0
```

```
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  -1
        .float  1
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  0
        .float  1
        .float  0
        .float  0
        .float  -1

*****************************************************************************
_ACB    .space  148
*****************************************************************************
_BandwidthExpandingFactor
        .float  0.994
        .float  0.8
*****************************************************************************

*********************************************************
        .asect "vecs",0h    ; interrupt and reset vectors

        .ref _c_int00   ; compiler defined C initialization reset
        .ref _c_int05   ; serial port transmit interrupt routine
        .ref _c_int11   ; DMA counter interrupt
        .ref _c_int99   ; unexpected interrupt handler

reset:  .word   _c_int00
int0:   .word   _c_int99
int1:   .word   _c_int99
```

```
int2:  .word  _c_int99
int3:  .word  _c_int99
xint0: .word  _c_int05
rint0: .word  _c_int99
xint1: .word  _c_int99
rint1: .word  _c_int99
tint0: .word  _c_int99
tint1: .word  _c_int99
dint:  .word  _c_int11

    .data

_intermediate
    .word  buff1
_input .word  buff2
_output .word  buff3
buff1  .space  240
buff2  .space  240
buff3  .space  240


    .global _intermediate, _input, _output,
    .global _buffer360

*********************************************************
    .end
```

```
/*****************************************************************************************/
/* ANALYZE.C                                                                             */
/* EVM PROGRAM FOR ANALYZING 240 SAMPLES OF SPEECH INTO 144 BITS                         */
/* (9 WORDS) CELP PARAMETERS, READY FOR TRANSMISSION                                     */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/

#include "analyze1.h"          /* DEFINE GLOBAL DATA                                     */
#include "analyze2.h"          /* DEFINE PROTOTYPES                                      */

/*---------------------------------------------------------------------------------------*/
/* pcm[] IS THE SPEECH SAMPLES                                                           */
/* stream[] IS THE CELP PARAMETERS                                                       */
/*---------------------------------------------------------------------------------------*/
void Analyze(int *pcm, unsigned *stream)
{
unsigned i;

    PCMToFloat(pcm,&OriginalSpeech[120]);
    AnalyzeLPC(&OriginalSpeech[120],HammingWindowData,LPC);
    ExpandBandwidth(&BandwidthExpandingFactor[0],LPC,LPC);
    ConvertToLSP(LPC,CELP,TableLSP);
    CheckLSPStability(CELP,PreviousLSP,TableLSP);
    CodebookSearching(OriginalSpeech,CELP);
    ConvertToDataStream(CELP,stream);
    ShiftTheOriginalSpeech(OriginalSpeech);
    UpdatePreviousLSP(CELP,PreviousLSP);
}

/*---------------------------------------------------------------------------------------*/
/* ROUTINE FOR CODE BOOK SEARCHING                                                       */
/* ospeech[] IS THE ORIGINAL SPEECH                                                      */
/* IntrCELP[] IS THE 26 CELP PARAMETERS                                                  */
/*---------------------------------------------------------------------------------------*/
void CodebookSearching(float *ospeech,unsigned *IntrCELP)
{
unsigned SubFrame;

    for(SubFrame = 0; SubFrame < 4; SubFrame ++)
    {
        InterpolateLSP(TableLSP,IntrCELP, PreviousLSP, SubFrame,
            FloatLSPP, FloatLSPQ);
        ConvertLSPToLPC(FloatLSPP, FloatLSPQ, LPC,TableLSP);
        ExpandBandwidth(&BandwidthExpandingFactor[1],LPC,ExpandedLPC);
        FindImpulseResponse(LPC,WeightedImpulseResponse);
        FindACBSearchingTarget(SubFrame,LPC,ExpandedLPC,ospeech,
            DelayElements,ACBSearchingTarget);
        if(SubFrame == 0 || SubFrame == 2)
        {
            ACBSearchingOdd(ACBGainTable,WeightedImpulseResponse,ACB,
                ACBSearchingTarget,IntrCELP+18+SubFrame,
                IntrCELP+22+SubFrame);
        }
        else
        {
```

```
            ACBSearchingEven(ACBGainTable,WeightedImpulseResponse,ACB,
                ACBSearchingTarget,IntrCELP+18+SubFrame,
                IntrCELP+22+SubFrame);
        }
        FindSCBSearchingTarget(ACBGainTable,IntrCELP+18+SubFrame,
            IntrCELP+22+SubFrame, ACB,WeightedImpulseResponse,
            ACBSearchingTarget, SCBSearchingTarget,ExcitationPulses);
        SCBSearching(SCBGainTable, SCB,WeightedImpulseResponse,
            SCBSearchingTarget,IntrCELP+10+SubFrame,
            IntrCELP+14+SubFrame);
        UpdateACB(ACBGainTable,SCBGainTable,LPC,IntrCELP+10+SubFrame,
            ACB,SCB, ExcitationPulses);
        GetDelayElement(LPC,ExcitationPulses,DelayElements,SyntheticSpeech);
            IntrCELP[18+SubFrame] = IntrCELP[18+SubFrame] - 20;
        if(SubFrame == 1 || SubFrame == 3)
        {
            DeltaEncodingACB(IntrCELP+18+SubFrame);
        }
    }
}

/*-------------------------------------------------------------------------------------------*/
/* EXPANDING BANDWIDTH OF LP PARAMETERS, CALLED TWICE FOR                                    */
/* (1) AVOID INSTABILITY CAUSING BY THE LSP QUANTIZATION                                     */
/* (2) PERCEPTUALLY WEIGHTING                                                                */
/* *gamma HOLDS THE EXPANSION FACTOR                                                         */
/* al[] IS THE INPUT LP PARAMETERS                                                           */
/* aout[] IS THE EXPANDED LP PARAMETERS                                                      */
/*-------------------------------------------------------------------------------------------*/
void ExpandBandwidth(float *gamma, float *al, float *aout)
{
unsigned i;
float beta;
float Gamma;
    /* Bandwidth expansion */
        beta = (float) 1;
        Gamma = *gamma;
        for(i = 0; i<11; i++)
        {
            aout[i] = al[i] * beta;
            beta = Gamma * beta;
        }
}

/*-------------------------------------------------------------------------------------------*/
/* SHIFTING THE SPEECH DATA                                                                  */
/* THIS IS USED TO PERFORM SMOOTH LP ANALYSIS WHERE THE INPUT                                */
/* OF LP ANALYSIS IS HALF OF CURRENT FRAME AND HALF OF THE NEXT                              */
/* FRAME                                                                                     */
/*-------------------------------------------------------------------------------------------*/
void ShiftTheOriginalSpeech(float *ospeech)
{
unsigned i;
    for(i = 0; i < 120; i++)
    {
        ospeech[i] = ospeech[240 + i];
    }
```

```
}

/*-------------------------------------------------------------------------------*/
/* DELTA ENCODING TO REPRESENT THE ACB ENTRIES OF EVEN FRAME IN                   */
/* A DIFFERENCE BETWEEN THE ACTUAL ENTRIES AND PREVIOUS ENTRIES                  */
/*-------------------------------------------------------------------------------*/
void DeltaEncodingACB(unsigned *intrcelp)
{
    if(*(intrcelp-1) > 31 && *(intrcelp-1) <95)
    {
        *intrcelp = 31 + *intrcelp - *(intrcelp-1);
    }
    if(*(intrcelp-1) >= 95)
    {
        *intrcelp = *intrcelp - 64;
    }

}

/*-------------------------------------------------------------------------------*/
/* WE PRESERVE THE PREVIOUS LSP PARAMETERS FOR:                                  */
/* (1) INTERPOLATION                                                             */
/* (2) STABILITY RULE                                                            */
/*-------------------------------------------------------------------------------*/
void UpdatePreviousLSP(unsigned *IntrCELP,unsigned *OldLSP)
{
unsigned i;
    for(i = 0; i < 10; i++)
    {
        OldLSP[i] = IntrCELP[i];
    }
}
```

```
/*****************************************************************************************/
/* ANALYZE1.H                                                                            */
/* INCLUDE FILE FOR ANALYZE.C                                                            */
/* DEFINING GLOBAL DATA                                                                  */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/

extern float  SCB[1082];                  /* Source of SCB                               */
extern float  ACB[147];                   /* Source of ACB                               */
extern float  BandwidthExpandingFactor[2];   /* For LPC Bandwidth Expansion              */
extern float  HammingWindowData[240];
extern float  TableLSP[160];              /* FS-1016 LSP Table                           */
extern float  ACBGainTable[32];           /* FS-1016 ACB Gain Table                      */
extern float  SCBGainTable[32];           /* FS-1016 SCB Gain Table                      */

extern float  DelayElements[11];          /* Storage of LP Filter                        */
extern float  OriginalSpeech[360];
extern float  LPC[11];                    /* LPC Parameters                              */
extern float  ExpandedLPC[11];
extern float  WeightedImpulseResponse[60];
extern float  ACBSearchingTarget[60];
extern float  SCBSearchingTarget[60];
extern float  ExcitationPulses[60];
extern int    SyntheticSpeech[60];
extern unsigned     CELP[26];             /* CELP Parameters                             */
/*    CELP[0 - 9]    = 10 LSP entries
      CELP[10 - 13]  = SCB entries for frame 1 - 4
      CELP[14 - 17]  = SCB gain for frame 1 - 4
      CELP[18 - 21]  = ACB entries for frame 1 - 4
      CELP[22 - 25]  = ACB gain for fram 1 - 4
*/
extern float  FloatLSPP[6];  /* Real value of LSP used to find LPC back                  */
extern float  FloatLSPQ[6];  /* Idem for interleave LSP                                  */

extern unsigned  PreviousLSP[10];      /* LSPs of previous frame                         */
```

```
/*****************************************************************************************/
/* ANALYZE2.H                                                                            */
/* INCLUDE FILE FOR ANALYZE.C                                                            */
/* DEFINING PROTOTYPES                                                                   */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/

void PCMToFloat(int *pcm, float *ospeech);
void AnalyzeLPC(float *ospeech,float *hw,float *al);
void ExpandBandwidth(float *gamma, float *al, float *aout);
void ConvertToLSP(float *a_en, unsigned *w,float *lsp);
void CodebookSearching(float *ospeech,unsigned *IntrCELP);
void InterpolateLSP(float *LSP,unsigned *NewLSP, unsigned *OldLSP,
     unsigned SubFrame, float *xp, float *xq);
void ConvertLSPToLPC(float *xp, float *xq, float *a_de,float *lsp);
void ShiftTheOriginalSpeech(float *ospeech);
void UpdatePreviousLSP(unsigned *IntrCELP,unsigned *OldLSP);
void ConvertFromStream(unsigned *s, unsigned *t);
void ConvertToDataStream(unsigned *t, unsigned *s);
void FindImpulseResponse(float *c, float *h);
void FindACBSearchingTarget(unsigned k,float *a,float *c,
     float *ospeech, float *delay, float *y);
void ACBSearchingOdd(float *a_gain,float *h,float *acb,float *ya,
     unsigned *Entry,unsigned *Gain);
void ACBSearchingEven(float *a_gain,float *h,float *acb,float *ya,
     unsigned *Entry,unsigned *Gain);
void FindSCBSearchingTarget(float *a_gain,unsigned *ACBEntry,unsigned *acb_g,
     float *acb, float *h,float *ya,float *ys,float *ex);
void SCBSearching(float *s_gain,float *scb,float *h,float *xs,
     unsigned *SCBEntry, unsigned *SCBGain);
void UpdateACB(float *a_gain,float *s_gain,float *trcelp,unsigned *SCBEntry,
     float *acb,float *scb,float *ex);
void GetDelayElement(float *a,float *ex,float *yl,int *sspeech);
void DeltaEncodingACB(unsigned *intrcelp);
void GenerateHammingData(float *hw);
void GenerateTableLSP(float *lsp);
void GenerateTableGain(float*a_gain,float *s_gain);
void GetSCBSource(float *scb);
void CheckLSPStability(unsigned *w,unsigned *old, float *ls);
```

```
/*****************************************************************************************/
/* LPC.C                                                                                 */
/*  EVM PROGRAM FOR ANALYZING 240 SAMPLES OF SPEECH INTO 11 LP                           */
/*  PARAMETERS                                                                           */
/*  PROGRAMMED BY ARMEIN LANGI                                                           */
/*  UNIVERSITY OF MANITOBA                                                               */
/*  @ 1990, 1991, 1992                                                                   */
/*****************************************************************************************/

/*---------------------------------------------------------------------------------------*/
/* GLOBAL DATA & PROTOTYPES                                                              */
/*---------------------------------------------------------------------------------------*/
float  WindowedSpeech[240];
float CorrelationCoefficients[11];
int LevinsonDurbin(float *r, float *al);


/*---------------------------------------------------------------------------------------*/
/* LPC ANALYSIS PROGRAM                                                                  */
/* ospeech[] IS THE ORIGINAL SPEECH                                                      */
/* hw[] IS THE HAMMING WINDOWS                                                           */
/* al[] IS THE LP PARAMETERS                                                             */
/*---------------------------------------------------------------------------------------*/
void AnalyzeLPC(float *ospeech,float *hw,float *al)
{
unsigned i,k;
int flag;
float sum;

/*---------------------------------------------------------------------------------------*/
/* Window with Hamming Window                                                            */
/*---------------------------------------------------------------------------------------*/


    for(i = 0; i<240; i++)
    {
        WindowedSpeech[i] = ospeech[i] ; /*  * hw[i];                                     */
    }

/*---------------------------------------------------------------------------------------*/
/* Get the correlation coefficients                                                      */
/*---------------------------------------------------------------------------------------*/

    for (i = 0; i<= 10; i++)
    {
        sum = (float) 0;
        for (k = 0; k<= 239 -i; k++)
        {
            sum = sum + WindowedSpeech[k] *
                WindowedSpeech[k + i];
        }
        CorrelationCoefficients[i] = sum;
    }
    flag = LevinsonDurbin(CorrelationCoefficients,al);
    if(flag != 0)
    {
```

```
    }
}

/*-------------------------------------------------------------------------------------------------*/
/* LEVINSON DURBIN ALGORITHM                                                                       */
/* TO SOLVE A SYSTEM OF LINEAR EQUATIONS                                                           */
/*-------------------------------------------------------------------------------------------------*/
int LevinsonDurbin(float *r, float *al)
{
    float  sum;
    float  pe;
    float  akk, ai, aj, ra;
    int    i,j,k;
    int    pre_err;

    pe = r[0];
    al[0] = (float) 1;
    for (k = 1; k <= 10; k++)
    {
        sum = (float) 0;
        for (i = 1; i <= k; i++)
        {

            sum = sum - al[k - i] * r[i];
        }
        akk = sum/pe;
        al[k] = akk;
        for (i = 1; i <= k/2; i++)
        {
            ai = al[i];
            aj = al[k-i];
            al[i] = ai + akk * aj;
            al[k - i] = aj + akk * ai;
        }
        pe = pe * (1 - akk * akk);
        /*printf("\npe = %f", pe);*/
        if (pe <= (float) 0)
        {

            pre_err = 1;

        }
    }
    if (pre_err == 1)
    {
        return -1;
    }
    else
    {
        return 0;
    }
}

/*-------------------------------------------------------------*/
/* FIND THE IMPULSE RESPONSE OF AN ALL POLE FILTER             */
/* USED FOR BULIDING A MATRIX REPRESENTATION OF THE FILTER     */
/*-------------------------------------------------------------*/
```

```
void FindImpulseResponse(float *c, float *h)
{
unsigned i,j;
float yz[11];

    for(i=1;i<11;i++)
    {
        yz[i]= (float) 0;
    }

    h[0] = (float) 1;
    yz[1] = (float) 1;
    for(i = 1; i < 60; i++)
    {
        yz[0] = (float) 0;
        for(j = 1; j < 11; j++)
        {
            yz[0] = yz[0] - c[j] * yz[j];
        }
        h[i] = yz[0];
        for(j = 1; j <11; j++)
        {
            yz[11-j] = yz[10-j];
        }
    }
}
```

```
/*****************************************************************************************/
/* FIND_LSP.C                                                                            */
/*  EVM PROGRAM FOR CONVERSIONS OF LPC AND LSP PARAMETERS                                */
/*  PROGRAMMED BY ARMEIN LANGI                                                           */
/*  UNIVERSITY OF MANITOBA                                                               */
/*  @ 1990, 1991, 1992                                                                   */
/*****************************************************************************************/

#include <float.h>
#include <math.h>

float sumpq(float x,float *c);

/*---------------------------------------------------------------------------------------*/
/* THIS IS FOR CONVERSION FROM LSP TO LPC                                                */
/* HOWEVER, YOU MUST RUN interpolateLSP() BEFORE USING THIS                              */
/* xp[] IS THE COSINE OF ANGLES OF ZEROS OF P(Z)                                         */
/* xq[] IS THE COSINE OF ANGLES OF ZEROS OF Q(Z)                                         */
/* a_de[] IS THE LPC PARAMETERS                                                          */
/* lsp[] A QUANTIZATION TABLE OF FS-1016 LSP                                             */
/*---------------------------------------------------------------------------------------*/
void ConvertLSPToLPC(float *xp, float *xq, float *a_de,float *lsp)
{
unsigned int i,j,m,n;
float   p[6],q[6],b[6];

    for(i=0; i <6; i++)
    {
        p[i] = (float) 0;
        q[i] = (float) 0;
        b[i] = (float) 0;
    }


    b[5] = (float) 32;

    b[4] = (float) 0;
    for(i = 0; i <5; i++)
    {
        b[4] = b[4] - xp[i+1];
    }
    b[4] = b[5] * b[4];

    b[3] = (float) 0;

    for(i = 1; i < 5; i++)
    {
        for(j = i+1; j < 6; j++)
        {
            b[3] = b[3] +xp[i]*xp[j];
        }
    }
    b[3] = b[5] * b[3];

    b[2] = (float) 0;
    for(i = 1; i < 4; i++)
    {
```

```
        for(j = i+1; j < 5; j++)
        {
            for(m = j+1; m < 6; m++)
            {
                b[2] = b[2] -xp[i]*xp[j]* xp[m];;
            }
        }
    }
    b[2] = b[5] * b[2];

    b[1] = (float) 0;
    for(i = 1; i < 3; i++)
    {
        for(j = i+1; j < 4; j++)
        {
            for(m = j+1; m < 5; m++)
            {
                for(n = m + 1; n <6; n++)
                {
                    b[1] = b[1] +xp[i]*xp[j]* xp[m]*xp[n];;
                }
            }
        }
    }
    b[1] = b[1] * b[5];

    b[0] = -b[5] * xp[1] * xp[2] * xp[3] * xp[4] * xp[5];

    p[1] = b[4]/16;
    p[2] = (b[3]+40)/8;
    p[3] = (b[2]+16*p[1])/4;
    p[4] = (b[1]+6*p[2]-10)/2;
    p[5] = b[0] + 2*p[3] - 2*p[1];



    b[5] = (float) 32;

    b[4] = (float) 0;
    for(i = 0; i <5; i++)
    {
        b[4] = b[4] - xq[i+1];
    }
    b[4] = b[5] * b[4];


    b[3] = (float) 0;

    for(i = 1; i < 5; i++)
    {
        for(j = i+1; j < 6; j++)
        {
            b[3] = b[3] +xq[i]*xq[j];
        }
    }
    b[3] = b[5] * b[3];
```

```
    b[2] = (float) 0;
    for(i = 1; i < 4; i++)
    {
        for(j = i+1; j < 5; j++)
        {
            for(m = j+1; m < 6; m++)
            {
                b[2] = b[2] -xq[i]*xq[j]* xq[m];;
            }
        }
    }
    b[2] = b[5] * b[2];

    b[1] = (float) 0;
    for(i = 1; i < 3; i++)
    {
        for(j = i+1; j < 4; j++)
        {
            for(m = j+1; m < 5; m++)
            {
                for(n = m + 1; n <6; n++)
                {
                    b[1] = b[1] +xq[i]*xq[j]* xq[m]*xq[n];;
                }
            }
        }
    }
    b[1] = b[1] * b[5];

    b[0] = -b[5] * xq[1] * xq[2] * xq[3] * xq[4] * xq[5];

    q[1] = b[4]/16;
    q[2] = (b[3]+40)/8;
    q[3] = (b[2]+16*q[1])/4;
    q[4] = (b[1]+6*q[2]-10)/2;
    q[5] = b[0] + 2*q[3] - 2*q[1];

    a_de[0] = (float) 1;

    p[0] = q[0] = (float) 1;

    for(i = 1; i < 6; i++)
    {
        a_de[i] = (p[i-1]+p[i] + q[i]-q[i-1])/2;
        a_de[11-i] = (p[i-1]+p[i]+q[i-1]-q[i])/2;
    }
}

/*-------------------------------------------------------------------------------------------*/
/* THIS IS FOR CONVERSION FROM LPC TO LSP                                                    */
/* a_en[] IS THE LPC PARAMETERS                                                              */
/* w[] IS THE LSP PARAMETERS                                                                 */
/* lsp[] A QUANTIZATION TABLE OF FS-1016 LSP                                                 */
/*-------------------------------------------------------------------------------------------*/
void ConvertToLSP(float *a_en, unsigned *w,float *lsp)
{
unsigned int i,j,m,flag,con,SpecialFlag;
    float b[5],c[5],p[6], q[6], x5, x4, x3, x2;
```

```
float x,lastxp, lastxq, sump1,sump2, sumq1, sumq2;

for(i = 0; i < 6; i++)
{
     p[i] = (float) 0;
     q[i] = (float) 0;
}
for(i = 0; i < 5; i++)
{
     w[i] = 0;
     w[9-i] = 0;

}
p[0] = q[0] = (float) 1;
for(i = 1; i < 6; i++)
{
     p[i] = a_en[i] + a_en[11 - i] - p[i-1];
     q[i] = a_en[i] - a_en[11 - i] + q[i-1];
}

b[4] = 16 * p[1];
b[3] = 8 * (p[2]-5);
b[2] = 4*(p[3]-4*p[1]);
b[1] = 2*(p[4]-3*p[2]+5);
b[0] = p[5] - 2*p[3] + 2*p[1];

c[4] = 16 * q[1];
c[3] = 8 * (q[2]-5);
c[2] = 4*(q[3]-4*q[1]);
c[1] = 2*(q[4]-3*q[2]+5);
c[0] = q[5] - 2*q[3] + 2*q[1];

sump2 = sumq2 = (float) 0;
x = (float) 1;
sump1 = sumpq(x,b);
sumq1 = sumpq(x,c);
lastxp = lastxq = (float) 1;

i=0;
SpecialFlag = 0;
while(i<5)
{
     if(i==1 || i==2)
     {
          m=16;
     }
     else
     {
          m=8;
     }
     for(j=0;j<m;j++)
     {
          flag=0;
          x=lsp[j+(32*i)];
          if(x<lastxp)
          {
               sump2 = sumpq(x,b);
```

```
                    if(sump1 != (float) 0 &&
                        sump1 * sump2 <= (float) 0.0 )
                    {
                      if(SpecialFlag == 0)
                      {
                          flag = 1;
                          if(sump2*sump2<=sump1*sump1)
                          {
                              w[i*2] = j;
                          }
                          else
                          {
                              w[i*2] = j - 1;
                          }

                          if(j == 0)
                          {
                              w[i*2] = j;
                          }
                      }
                       else
                       {
                          SpecialFlag = 0;
                       }
                       lastxp = x;
                    }
                    sump1=sump2;

                }
                if(flag==1) break;
            }
            if(flag!=1)
            {
                j=0;
                w[i*2]=m-1;
                lastxp=x;
                sump1 = sumpq(x,b);

                SpecialFlag = 1;
/*              if(i !=4)
                {
                    while(x<lsp[j+(i+1)*32])
                    {
                        j++;
                    }
                    lastxp=lsp[j+(i+1)*32];
                    x=lastxp;

                    sump1 = sumpq(x,b);

                }
*/
            }
            i=i+1;
        }

        i=0;
```

```
while(i<5)
{
    if(i==0 || i==1)
    {
        m=16;
    }
    else
    {
        m=8;
    }
    flag = 0;
    lastxq = lsp[w[i*2] + i*32];
    if(i <4)
        lastxp = lsp[w[(i+1)*2] + (i+1)*32];
    sumq1 = sumpq(lastxq,c);
    sump1 = sumpq(lastxp,c);
    if(sump1 * sumq1 > (float) 0 && i <4 )
    {
        w[i*2+1] = 0;
        while(lsp[w[i*2+1] + i*32 +16] >= lastxq)
        {
            w[i*2+1]++;
        }

    }
    else
    {
        j = 0;
        while(flag == 0)
        {
            x = lsp[j+i*32+16];
            if(x<lastxq)
            {

            sumq2 = sumpq(x,c);
            if(sumq1*sumq2 <= (float) 0)
            {
                flag = 1;
                if(sumq1*sumq1 > sumq2*sumq2)
                {
                    w[i*2+1] = j;
                }
                else
                {
                    w[i*2+1] = j-1;
                }
                if(lsp[w[i*2+1] + i*32 +16] >=
                    lastxq)
                {
                    w[i*2+1]++;
                }

                if(w[i*2+1] > m)
                {
                    w[i*2+1] = j;
                }
                lastxq = x;
            }
```

```
                    sumq1 = sumq2;
                    }

                    if(j == m - 1 && flag == 0)
                    {
                    flag == 1;
                    w[i*2+1] = j ;
                    }
                    j = j+1;
                }
            }
            i=i+1;
        }

}

/*-------------------------------------------------------------------------------*/
/* THIS IS A ROUTINE TO EVALUATE THE POLYNOMIAL P(Z) OR Q(Z)                     */
/* FOR A GIVEN COSINE OF ANGLES x IN POLAR COORDINATE                            */
/* THE X THAT GIVE 0 EVALUATION IS THE COSINE OF THE ANGLES OF A                 */
/* ZERO OF THE POLYNOMIAL                                                        */
/* c[] IS COEFFICIENT OF THE POLYNOMIAL                                          */
/*-------------------------------------------------------------------------------*/
float sumpq(float x,float *c)
{
float sumq2, x5, x4, x3, x2;
                    x2 = x*x;
                    x3 = x*x2;
                    x4 = x*x3;
                    x5 = x*x4;

                    sumq2 = (32*x5) + (c[4]*x4)
                    + (c[3]*x3) + (c[2]*x2)
                    + (c[1]*x) + c[0];

                        return sumq2;
}

/*-------------------------------------------------------------------------------*/
/* THIS IS A ROUTINE IS TO INTERPOLATE THE LSP PARAMETERS                        */
/* THE RESULTS ARE THE COSINE OF ANGLES OF POLYNOMIAL P(Z) AND Q(Z)              */
/* HERE LSP[] IS A TABLE OF QUANTIZED US-FS-1016 LSP                             */
/*-------------------------------------------------------------------------------*/
void InterpolateLSP(float *LSP,unsigned *NewLSP, unsigned *OldLSP,
unsigned SubFrame, float *xp, float *xq)
{
unsigned i,j,m,n;

    for(n = 0; n < 5; n++)
    {
        i = OldLSP[2*n] + 32 * n;
        j = NewLSP[2*n] + 32 * n;
        xp[n+1]= LSP[i] * (float) (9 - 2 * (SubFrame + 1));
        xp[n+1] += LSP[j] * (float) ((SubFrame+1) * 2 - 1);
        xp[n+1] = xp[n+1] / (float) 8;
        i = OldLSP[2*n+1] + 32 * n + 16;
        j = NewLSP[2*n+1] + 32 * n + 16;
```

```
            xq[n+1]= LSP[i] * (float) (9 - 2 * (SubFrame + 1));
            xq[n+1] += LSP[j] * (float) ((SubFrame+1) * 2 - 1);
            xq[n+1] = xq[n+1] / (float) 8;

        }
}

/*-------------------------------------------------------------------------------------------------*/
/* THIS ROUTINE TEST THE MONOTONICITY OF THE LSP ACCORDING                                         */
/* TO THE STABILITY RULE.  IF THE LSP FAIL, IT IS REPLACED BY THE                                  */
/* PREVIOUS LSP                                                                                    */
/*-------------------------------------------------------------------------------------------------*/
void CheckLSPStability(unsigned *w,unsigned *old, float *ls)
{
unsigned i,flag;
float now,before;
    before = (float) 1.0;
    flag = 0;
    i = 0;
    do
    {

        now=ls[i*16+w[i]];
        if(now > before)
        {
            flag = 1;
        }
        before = now;
        i = i + 1;
    }while(flag == 0 && i < 10);

    if(flag == 1)
    {
        for(i = 0; i < 10; i++)
        {
            w[i] = old[i];
        }
    }

}
```

```
/*****************************************************************************************************/
/* FIND_ACB.C                                                                                        */
/* EVM PROGRAM FOR SEARCHING ACB PARAMETERS                                                          */
/* PROGRAMMED BY ARMEIN LANGI                                                                        */
/* UNIVERSITY OF MANITOBA                                                                            */
/* @ 1990, 1991, 1992                                                                                */
/*****************************************************************************************************/

float   bestPeak,va[187],vas[99];

void find_code_20(float *acb,float *va,float *vas,float *h);
void find_code_big(unsigned code,float *acb,float *va,float *h);
void find_code_small(unsigned code,float *acb,float *va,float *vas,float *h);
void joint_op(float *table_gain,unsigned int entry,float *y,float *v,unsigned
*bestPC,unsigned *inbestGain,float *bestPeak);

/*---------------------------------------------------------------------------------------------------*/
/* THIS IS FOR EVEN SUBFRAMES                                                                        */
/* a_gain[] IS A TABLE OF QUANTIZED GAIN                                                             */
/* h[] ARE THE IMPULSE RESPONSES OF THE FILTER W                                                     */
/* acb[] IS THE ACB                                                                                  */
/* ya[] IS THE SEARCHING TARGET                                                                      */
/* entry AND Gain ARE THE SEARCHING RESULTS                                                          */
/*---------------------------------------------------------------------------------------------------*/
void ACBSearchingEven(float *a_gain,float *h,float *acb,float *ya,
    unsigned *Entry,unsigned *Gain)
{
unsigned int    code,i;
    bestPeak = (float) 0;
    *Entry = 20;
    *Gain = 6;
    for(i = 0; i<187; i++)
    {
        va[i] = (float) 0;
    }
    for(i = 0; i < 99; i++)
    {
        vas[i] = (float) 0;
    }
    if(*(Entry-1) <=31)
    {
        find_code_20(acb,va,vas,h);
        joint_op(a_gain,20,ya,vas+39,Entry,Gain,&bestPeak);
        for(code = 21; code < 60; code++)
        {
            find_code_small(code,acb,va,vas,h);
            joint_op(a_gain,code,ya,vas+59-code,Entry,Gain,&bestPeak);
        }
        for(code = 60; code < 84; code++)
        {
            find_code_big(code,acb,va,h);
            joint_op(a_gain,code,ya,va+147-code,Entry,Gain,&bestPeak);
        }
    }
    if(*(Entry-1) >31 && *(Entry-1) <71)
    {
        find_code_20(acb,va,vas,h);
```

```
        for(code=21;code < *(Entry-1)-11; code++)
        {
            find_code_small(code,acb,va,vas,h);
        }
        for(code = *(Entry-1)-11; code < 60; code++)
        {
            find_code_small(code,acb,va,vas,h);
            joint_op(a_gain,code,ya,vas+59-code,Entry,Gain,&bestPeak);
        }
        for(code = 60; code < *(Entry-1)+53; code++)
        {
            find_code_big(code,acb,va,h);
            joint_op(a_gain,code,ya,va+147-code,Entry,Gain,&bestPeak);
        }
    }
    if(*(Entry-1) >=71 && *(Entry-1) <95)
    {
        find_code_20(acb,va,vas,h);
        for(code=21;code < 60; code++)
        {
            find_code_small(code,acb,va,vas,h);
        }
        for(code = 60; code <*(Entry-1)-11;code++)
        {
            find_code_big(code,acb,va,h);
        }
        for(code = *(Entry-1)-11; code < *(Entry-1)+53; code++)
        {
            find_code_big(code,acb,va,h);
            joint_op(a_gain,code,ya,va+147-code,Entry,Gain,&bestPeak);
        }
    }
    if(*(Entry-1) >=95)
    {
        find_code_20(acb,va,vas,h);
        for(code = 21; code < 60; code++)
        {
            find_code_small(code,acb,va,vas,h);
        }
        for(code = 60; code < 84; code++)
        {
            find_code_big(code,acb,va,h);
        }
        for(code = 84;code <148; code++)
        {
            find_code_big(code,acb,va,h);
            joint_op(a_gain,code,ya,va+147-code,Entry,Gain,&bestPeak);
        }
    }
}

/*-------------------------------------------------------------------------------------------*/
/* THIS IS FOR ODD SUBFRAMES                                                                 */
/* a_gain[] IS A TABLE OF QUANTIZED GAIN                                                     */
/* h[] ARE THE IMPULSE RESPONSES OF THE FILTER W                                             */
/* acb[] IS THE ACB                                                                          */
/* ya[] IS THE SEARCHING TARGET                                                              */
```

```
/* entry AND Gain ARE THE SEARCHING RESULTS                                    */
/*-----------------------------------------------------------------------------*/
void ACBSearchingOdd(float *a_gain,float *h,float *acb,float *ya,
    unsigned *Entry,unsigned *Gain)
{
unsigned code,i;
    bestPeak = (float) 0;
    *Entry = 20;
    *Gain = 6;
    for(i = 0; i<187; i++)
    {
        va[i] = (float) 0;
    }
    for(i = 0; i < 99; i++)
    {
        vas[i] = (float) 0;
    }

    find_code_20(acb,va,vas,h);
    joint_op(a_gain,20,ya,vas+39,Entry,Gain,&bestPeak);


    for(code = 21; code < 60; code++)
    {
        find_code_small(code,acb,va,vas,h);
        joint_op(a_gain,code,ya,vas+59-code,Entry,Gain,&bestPeak);
    }

    for(code = 60; code < 148; code++)
    {
        find_code_big(code,acb,va,h);
        joint_op(a_gain,code,ya,va+147-code,Entry,Gain,&bestPeak);
    }
}


/*-----------------------------------------------------------------------------*/
/* CONVOLUTION OF THE FIRST ENTRY                                              */
/*-----------------------------------------------------------------------------*/
void find_code_20(float *acb,float *va,float *vas,float *h)
{
unsigned int    i,j;
float  c;

    for(j = 0; j <20 ; j++)
    {
        c =  (float) acb[127+j];

        for(i = j ; i < 60; i++)
        {
            *(va+127+i) = *(va+127+i)+h[i-j] * c;
            *(vas+39+i) = *(va+127+i);
        }

    }

 for(j = 20;j<40;j++)
 {
```

```
        *(vas+39+j) = *(vas+39+j)+ *(va+j+107);
    }

    for(j=40;j<60;j++)
    {
            *(vas+39+j) = *(vas+39+j)+ *(va+j+107)+ *(va+j+87);
    }

}

/*-------------------------------------------------------------------------------------------*/
/* CONVOLUTION OF THE ENTRIES >= 60                                                          */
/*-------------------------------------------------------------------------------------------*/
void find_code_big(unsigned code,float *acb,float *v,float *h)
{
unsigned int    i,dums;
    float  c,sum;
    dums = 147-code;
    c = (float) acb[dums];
    for(i = 0; i <60; i++)
    {
        sum= h[i]*c;
        sum= *(v+dums+i)+sum;
        *(v+dums+i) = sum;
        sum = sum;
    }
}

/*-------------------------------------------------------------------------------------------*/
/* CONVOLUTION OF THE ENTRIES > 20 AND < 60                                                  */
/*-------------------------------------------------------------------------------------------*/
void find_code_small(unsigned code,float *acb,float *va,float *vas,float *h)
{
unsigned   int  code1,code2,j,dums;
    float  c;
    code1 = code*2;
    dums = 147 - code;

    c =  (float) acb[dums];
    for(j = 0 ; j < 60; j++)
    {
        *(va+dums+j) = *(va+dums+j)+h[j] * c;
        *(vas+dums-88+j) = *(va+dums+j);
    }

    if(code1<60)
    {
        code2 = code1;
    }
    else
    {
        code2 = 60;
    }

    for(j = code; j < code2; j++)
    {
        *(vas+dums-88+j) = *(vas+dums-88+j)+ *(va+dums+j-code);
```

```
    }

    if(code1<60)
    {
        for(j=code1; j<60; j++)
        {
            *(vas+dums-88+j) = *(vas+dums-88+j)+ *(va+dums+j-code1)+ *(va+dums+j-
code);

        }
    }
}


/*-------------------------------------------------------------------------------------------------*/
/* FIND THE ACB SEARCHING TARGET y[]                                                               */
/* FOR GIVEN:                                                                                      */
/* k = SUBFRAME #                                                                                  */
/* a = LP FILTER                                                                                   */
/* c = WEIGHTED LP FILTER                                                                          */
/* ospeech = ORIGINAL SPEECH                                                                       */
/* delay = DELAY ELEMENTS OF THE SYNTHESIZER                                                       */
/*-------------------------------------------------------------------------------------------------*/
void FindACBSearchingTarget(unsigned k,float *a,float *c, float *ospeech,
     float *delay, float *y)
{
unsigned    i,j;
float   yk[11];

    for(i = 0; i <11; i++)
    {
        yk[i] = delay[i];
    }

    for(i = 0; i < 60; i++)
    {
        yk[0] = (float) 0;
        for( j = 1; j < 11; j++)
        {
            yk[0] = yk[0] - a[j] * yk[j];
        }

        y[i] = ospeech[i+(60*k)] - yk[0];

        for(j = 1; j<11; j++)
        {
            yk[11-j] = yk[10-j];
        }

    }

}
```

```
/*****************************************************************************************/
/* FIND_SCB.C                                                                            */
/* EVM PROGRAM FOR SEARCHING SCB PARAMETERS                                              */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/

float  vh[1082];
float  xa[60];

void joint_op(float *table_gain,unsigned int entry,float *y,float *v,unsigned
*bestPC,unsigned *inbestGain,float *bestPeak);
void find_entry_more(unsigned int entry,float *scb,float *vh,float *h);
void find_entry_0(float *scb, float *vh,float *h);
void Quantize_gain(float bestgain,float *a_gain,unsigned *intrcelp);

/*---------------------------------------------------------------------------------------*/
/* s_gain[] IS A TABLE OF QUANTIZED GAIN                                                 */
/* h[] ARE THE IMPULSE RESPONSES OF THE FILTER W                                         */
/* scb[] IS THE SCB                                                                      */
/* xs[] IS THE SEARCHING TARGET                                                          */
/* SCBentry AND SCBGain ARE THE SEARCHING RESULTS                                        */
/*---------------------------------------------------------------------------------------*/
void SCBSearching(float *s_gain,float *scb,float *h,float *xs,
    unsigned *SCBEntry, unsigned *SCBGain)
{

unsigned int entry;
float bestPeak;
unsigned i;

    bestPeak = (float) 0;
    *SCBEntry = 0;
    *SCBGain = 16;
    for(i = 0; i < 1082; i++)
    {
        vh[i] =(float) 0;
    }

    find_entry_0(scb,vh,h);
    joint_op(s_gain,0,xs,vh+(2*511),SCBEntry,SCBGain,&bestPeak);
    for(entry = 1; entry < 512; entry++)
    {
        find_entry_more(entry,scb,vh,h);
        joint_op(s_gain,entry,xs,vh+(2*(511-entry)),SCBEntry,
            SCBGain,&bestPeak);
    }

}

/*---------------------------------------------------------------------------------------*/
/* JOINT OPTIMIZATION SCHEME                                                             */
/* table_gain[] IS A TABLE OF QUANTIZED GAIN                                             */
/* [] ARE THE IMPULSE RESPONSES OF THE FILTER W                                          */
/* v[] IS THE CONVOLUTION RESULT OF ENTRY entry                                          */
/* y[] IS THE SEARCHING TARGET                                                           */
```

```
/* bestPC, inbestGain, and bestPeak ARE THE WINNER OF COMPETITION                */
/* bestPC HOLDS THE BEST entry THAT IS THE CANDIDATE OF THE WINNER               */
/* inbestGain HOLDS THE WINNING ENTRY OF GAIN TABLE.                             */
/* bestPeak HOLDS THE BEST SCORE OF THE WINNING ENTRY                            */
/*-------------------------------------------------------------------------------*/
void joint_op(float *table_gain,unsigned int entry,float *y,float *v,
    unsigned *bestPC,unsigned *inbestGain,float *bestPeak)
{
float   gain,pc,ip1,ip2;
unsigned    i,j;


    ip1 = (float) 0;
    ip2 = (float) 0;
    for (i = 0 ; i <60 ; i++)
    {
        ip1 = ip1 + y[i]*v[i];
        ip2 = ip2 + v[i]*v[i];
    }

    if(ip2 == (float) 0 )
    {
        gain = (float) 0;
        Quantize_gain(gain,table_gain,&j);
        pc = gain * ip1;
        if (pc >= *bestPeak)
        {
            *bestPC = entry;
            *inbestGain = j;
            *bestPeak = pc;

        }
 /*       printf("\n%d %d %d Energy = 0 BestPC = %d",l,k,entry,bestPC );
   */
    }
    else
    {
        gain = ip1/ip2;
        Quantize_gain(gain,table_gain,&j);
        gain=table_gain[j];
        pc = (2*ip1 - gain * ip2) * gain;
        if (pc > *bestPeak)
        {
            *bestPC = entry;
            *inbestGain = j;
            *bestPeak = pc;

        }
/*    printf("k=%u pc=%f gain=%f bestPC=%u\n",entry,pc,gain,*bestPC);
    */
    }
}

/*-------------------------------------------------------------------------------*/
/* CONVOLUTION OF THE FIRST ENTRY                                                */
/*-------------------------------------------------------------------------------*/
void find_entry_0(float *scb,float *v,float *h1)
```

```
{
unsigned int    dums,i,j;
    float  b;
    dums = 2*511;
    for(j = 0; j <60 ; j++)
    {
        b = (float) scb[j+dums];
        if (b > (float) 0)
        {
            for(i = j ; i < 60; i++)
            {
                *(v+dums+i) = *(v+dums+i)+h1[i-j];
            }
        }
        if (b < (float) 0)
        {

            for(i = j ; i < 60; i++)
            {
                *(v+dums+i) = *(v+dums+i)-h1[i-j];
            }
        }
    }
}

/*-------------------------------------------------------------------------------------------------------------------*/
/* CONVOLUTION OF THE ENTRIES > 0                                                                                    */
/*-------------------------------------------------------------------------------------------------------------------*/
void find_entry_more(unsigned int entry,float *scb,float *vh,float *h)
{
float  b;
unsigned int    i,j,dums;

    dums = 2*(511-entry);

    for(j = 0; j < 2; j++)
    {
        b = (float) scb[j+dums];
        if (b > (float) 0)
        {
            for(i = j ; i < 60; i++)
            {
                *(vh+dums+i) = *(vh+dums+i)+h[i-j];
            }
        }
        if (b < (float) 0)
        {
            for(i = j ; i < 60; i++)
            {
                *(vh+dums+i) = *(vh+dums+i)-h[i-j];
            }
        }
    }
}

/*-------------------------------------------------------------------------------------------------------------------*/
/* FIND THE SCB SEARCHING TARGET ys[]                                                                                */
/* FOR GIVEN:                                                                                                        */
```

```
/* a_gain = ACB GAIN TABLE                                                    */
/* ACBEntry = ACB ENTRY                                                       */
/* acb_g = ACB GAIN                                                           */
/* acb = ACB                                                                  */
/* h = IMPULSE RESPONSES OF FILTER W                                          */
/* ya = ACB SEARCHING TARGET                                                  */
/* ys = SCB SEARCHING TARGET                                                  */
/* HERE, ex IS A HELPING BUFFER TO STORE THE EXCITATION FROM ACB              */
/*----------------------------------------------------------------------------*/
void FindSCBSearchingTarget(float *a_gain,unsigned *ACBEntry,
     unsigned *acb_g,float *acb, float *h,float *ya,float *ys,float *ex)
{
unsigned      i,j,dums;
float         ACBGain;

    ACBGain = a_gain[*acb_g];
    if(*ACBEntry<60)
    {
        dums=147 - *ACBEntry;
        j=0;
        for(i=0;i<60;i++)
        {
            if(j+dums >146) j=0;
            ex[i]=ACBGain * *(acb+dums+j);
            j=j+1;
        }

    }
    else
    {
        dums=147- *ACBEntry;
        for(i=0;i<60;i++)
        {
            ex[i]=*(acb+dums+i) * ACBGain;
        }
    }

    for(i=0;i<60;i++)
    {
        xa[i]= (float) 0;
        for(j=0;j<i+1;j++)
        {
            xa[i]=xa[i]+h[i-j]*ex[j];
        }
        ys[i]=ya[i]-xa[i];
    }

}


/*----------------------------------------------------------------------------*/
/* QUANTIZATION OF GAIN                                                       */
/* SELECT ONE OF 32 ENTRIES                                                   */
/* bestgain IS THE UNQUANTIZED GAIN                                           */
/* g IS A GAIN TABLE                                                          */
/* ingain THE BEST ENTRY CORRESPONDING TO THE bestgain                        */
/*----------------------------------------------------------------------------*/
```

```
void Quantize_gain(float bestgain,float *g,unsigned *ingain)
{
unsigned i,j;
    *ingain=0;
    if(bestgain >= g[16])
    {
        *ingain=16;
    }
    if(bestgain >= g[*ingain+8])
    {
        *ingain=*ingain+8;
    }
    if(bestgain >= g[*ingain+4])
    {
        *ingain=*ingain+4;
    }
    if(bestgain >= g[*ingain+2])
    {
        *ingain=*ingain+2;
    }
    if(bestgain >= g[*ingain+1])
    {
        *ingain=*ingain+1;
    }
    if(*ingain!=31)
    {
        if(bestgain-g[*ingain]>g[*ingain+1]-bestgain)
        {
            *ingain=*ingain+1;
        }
    }
}
```

```
/*****************************************************************************************************/
/* STREAM.C                                                                                          */
/*  EVM PROGRAM FOR CONVERSIONS CELP PARAMETERS AND DATA STREAM                                      */
/*                                                                                                   */
/*  PROGRAMMED BY ARMEIN LANGI                                                                       */
/*  UNIVERSITY OF MANITOBA                                                                           */
/*  @ 1990, 1991, 1992                                                                               */
/*****************************************************************************************************/
/*---------------------------------------------------------------------------------------------------*/
/* CELP PARAMETERS TO FS-1016 DATA STREAM                                                            */
/* t ARE THE CELP PARAMETERS                                                                         */
/* s ARE DATA STREAM                                                                                 */
/*---------------------------------------------------------------------------------------------------*/
void ConvertToDataStream(unsigned *t, unsigned *s)
{
unsigned i;
unsigned    e;
    e = 0;

    s[0] = (t[25] & 16)  >> 4 |
           (t[20] & 16)  >> 3 |
           (t[0]  & 2)   << 1 |
           (t[15] & 16)  >> 1 |
           (t[12] & 8)   << 1 |
           (t[10] & 256) >> 3 |
           (t[21] & 1)   << 6 |
           (t[7]  & 1)   << 7 |
           (t[23] & 8)   << 5 |
           (t[16] & 1)   << 9 |
           (t[18] & 32)  << 5 |
           (t[2]  & 8)   << 8 |
           (t[11] & 8)   << 9 |
           (t[13] & 16)  << 9 |
           (t[19] & 2)   << 13|
           (t[9]  & 1)   << 15;

    s[1] = (t[22] & 8)   >> 3 |
           (t[17] & 1)   << 1 |
           (t[4]  & 4) |
           (t[20] & 1)   << 3 |
           (e  & 1)   << 4 |
           (t[10] & 2)   << 4 |
           (t[13] & 256) >> 2 |
           (t[1]  & 4)   << 5 |
           (t[24] & 2)   << 7 |
           (t[21] & 32)  << 4 |
           (t[14] & 8)   << 7 |
           (t[12] & 32)  << 6 |
           (t[6]  & 1)   << 12 |
           (t[11] & 2)   << 12 |
           (t[20] & 128) << 7 |
           (t[10] & 1)   << 15;

    s[2] = (t[25] & 1)    |
           (t[3] & 8)     >> 2 |
           (t[16]  & 2)   << 1 |
           (t[10] & 32)   >> 2 |
```

```
            (t[19] & 1)  << 4 |
            (t[13] & 2)  << 4 |
            (t[8] & 1) << 6 |
            (t[12] & 256) >> 1  |
            (t[22] & 16)  << 4 |
            (t[15] & 4)  << 7 |
            (t[18] & 8)  << 7 |
            (t[5] & 2)  << 10 |
            (t[12] & 16)  << 8 |
            (t[11] & 4)  << 11 |
            (t[14] & 16) << 10 |
            (t[19] & 8)  << 12;

s[3] = (t[0] & 4) >> 2   |
            (t[24] & 4)   >> 1 |
            (t[0] & 2)  << 1 |
            (t[20] & 2)  << 2 |
            (t[17] & 8)  << 1 |
            (t[7] & 2)  << 4 |
            (t[24] & 1) << 6 |
            (t[11] & 256)  >> 1 |
            (t[21] & 2)  << 7 |
            (t[13] & 1)  << 9 |
            (t[2] & 4)  << 8 |
            (t[23] & 1)  << 11 |
            (t[18] & 64)  << 6 |
            (t[15] & 1)  << 13 |
            (t[12] & 64) << 8 |
            (t[9] & 2)  << 14;

s[4] = (t[22] & 2) >> 1   |
            (t[13] & 128)   >> 6 |
            (t[20] & 8)  >> 1 |
            (t[14] & 4)  << 1 |
            (t[4] & 8)  << 1 |
            (t[10] & 64)  >> 1 |
            (t[1] & 1) << 6 |
            (t[24] & 16)  << 3 |
            (t[18] & 16)  << 4 |
            (t[16] & 4)  << 7 |
            (t[6] & 2)  << 9 |
            (t[11] & 128)  << 4 |
            (t[12] & 1)  << 12 |
            (t[19] & 32)  << 8 |
            (t[3] & 2) << 13 |
            (t[14] & 1)  << 15;

s[5] = (t[25] & 8) >> 3   |
            (t[8] & 2)  |
            (t[20] & 64)  >> 4 |
            (t[10] & 16)  >> 1 |
            (t[15] & 2)  << 3 |
            (t[5] & 4)  << 3 |
            (t[13] & 8) << 3 |
            (t[23] & 4)  << 5 |
            (t[21] & 8)  << 5 |
            (t[0] & 1)  << 9 |
```

```
            (t[17] & 4)  << 8 |
            (t[7] & 4)  << 9 |
            (t[11]  & 16)  << 8 |
            (e & 4)  << 11 |
            (t[19] & 4) << 12 |
            (t[2] & 1)  << 15;

s[6] = (t[22] & 4) >> 2   |
            (t[16] & 16) >> 3  |
            (t[9]  & 4) |
            (t[13] & 32)  >> 2 |
            (t[11]  & 1)  << 4 |
            (t[18] & 4)  << 3 |
            (t[4] & 2) << 5 |
            (e) |
            (t[25] & 4)  << 6 |
            (t[15] & 8)  << 6 |
            (t[1] & 2)  << 9 |
            (t[21] & 16)  << 7 |
            (t[10]  & 4)  << 10 |
            (t[23] & 2)  << 12 |
            (t[12] & 128) << 7 |
            (t[3] & 1)  << 15;

s[7] = (t[11] & 32) >> 5   |
            (t[18] & 128) >>6  |
            (t[22]  & 1)  << 2 |
            (t[17] & 16)  >> 1 |
            (t[4]  & 1)  << 4 |
            (t[21] & 4)  << 3 |
            (t[10] & 8) << 3 |
            (t[12]  & 2)  << 6 |
            (t[6] & 4)  << 6 |
            (t[13] & 4)  << 7 |
            (t[18] & 2) << 9 |
            (t[23] & 16)  << 7 |
            (t[16] & 8)  << 9 |
            (t[2]  & 2)  << 12 |
            (t[10] & 128)  << 7 |
            (t[20] & 4) << 13 ;

s[8] = (t[11] & 64) >> 6   |
            (t[8] & 4) >> 1  |
            (t[25]  & 2)  << 1 |
            (t[14] & 2)  << 2 |
            (t[19]  & 16) |
            (e) |
            (t[5] & 1) << 6 |
            (t[24]  & 8)  << 4 |
            (t[13] & 64)  << 2 |
            (t[18] & 1)  << 9 |
            (t[1] & 8) << 7 |
            (t[17] & 2)  << 10 |
            (t[12] & 4)  << 10 |
            (t[3]  & 4)  << 11 |
            (t[20] & 32)  << 9 |
            (e);
```

```
}

/*-------------------------------------------------------------------------*/
/* FS-1016 DATA STREAM TO CELP PARAMETERS                                  */
/* t ARE THE CELP PARAMETERS                                               */
/* s ARE DATA STREAM                                                       */
/*-------------------------------------------------------------------------*/
void ConvertFromStream(unsigned *s, unsigned *t)
{
    t[0] = (s[5] & 512) >> 9 | (s[0] & 4) >> 1 | (s[3] & 1) << 2;
    t[1] = (s[4] & 64) >> 6 | (s[6] & 1024) >> 9 |
           (s[1] & 128) >> 5 | (s[8] & 1024) >> 7;
    t[2] = (unsigned) (s[5] & (unsigned) 32768) >> 15 | (s[7] & (unsigned) 8192) >> 12 |
           (s[3] & (unsigned) 1024) >> 8 | (s[0] & (unsigned) 2048) >> 8;
    t[3] = (s[6] & (unsigned) 32768) >> 15 | (s[4] & 16384) >> 13 |
           (s[8] & 8192) >> 11 | (s[2] & 2) << 2;
    t[4] = (s[7] & 16) >> 4 | (s[6] & 64) >> 5 |
           (s[1] & 4) | (s[4] & 16) >> 1;
    t[5] = (s[8] & 64) >> 6 | (s[2] & 2048) >> 10 |
           (s[5] & 32) >> 3 ;
    t[6] = (s[1] & 4096) >> 12 | (s[4] & 1024) >> 9 |
           (s[7] & 256) >> 6;
    t[7] = (s[0] & 128) >> 7 | (s[3] & 32) >> 4 | (s[5] & 2048) >>9;
    t[8] = (s[2] & 64) >> 6 | (s[5] & 2) | (s[8] & 2) << 1;
    t[9] = (s[0] & (unsigned) 32768) >> 15 |
         (s[3] & (unsigned) 32768) >> 14 | (s[6] & 4);

    t[10] = (s[1] & (unsigned) 32768) >> 15 | (s[1] & 32) >> 4 |
         (s[6] & 4096) >> 10 | (s[7] & 64) >> 3 |
         (s[5] & 8) << 1 | (s[2] & 8) << 2 |
         (s[4] & 32) << 1 | (s[7] & 16384) >> 7 |
         (s[0] & 32) << 3 ;

    t[11] = (s[6] & 16) >> 4 | (s[1] & 8192) >> 12 |
         (s[2] & 8192) >> 11 | (s[0] & 4096) >> 9 |
         (s[5] & 4096) >> 8 | (s[7] & 1) << 5 |
         (s[8] & 1) << 6 | (s[4] & 2048) >> 4 |
         (s[3] & 128) << 1 ;

    t[12] = (s[4] & 4096) >> 12 | (s[7] & 128) >> 6 |
         (s[8] & 4096) >> 10 | (s[0] & 16) >> 1 |
         (s[2] & 4096) >> 8 | (s[1] & 2048) >> 6 |
         (s[3] & 16384) >> 8 | (s[6] & 16384) >> 7 |
         (s[2] & 128) << 1 ;

    t[13] = (s[3] & 512) >> 9 | (s[2] & 32) >> 4 |
         (s[7] & 512) >> 7 | (s[5] & 64) >> 3 |
         (s[0] & 8192) >> 9 | (s[6] & 8) << 2 |
         (s[8] & 256) >> 2 | (s[4] & 2) << 6 |
         (s[1] & 64) << 2 ;

    t[14] = (s[4] & (unsigned) 32768) >>15 | (s[8] & 8) >>2 |
         (s[4] & 8) >> 1 | (s[1] & 1024) >> 7 |
         (s[2] & 16384) >> 10;

    t[15] = (s[3] & 8192) >>13 | (s[5] & 16) >>3 |
         (s[2] & 512) >> 7 | (s[6] & 512) >> 6 |
```

```
        (s[0] & 8) << 1;

    t[16] = (s[0] & 512) >>9 | (s[2] & 4) >>1 |
        (s[4] & 512) >> 7 | (s[7] & 4096) >> 9 |
        (s[6] & 2) << 3;

    t[17] = (s[1] & 2) >> 1 | (s[8] & 2048) >> 10 |
        (s[5] & 1024) >> 8 | (s[3] & 16) >> 1 |
        (s[7] & 8) << 1;

    t[18] = (s[8] & 512) >> 9 | (s[7] & 1024) >> 9 |
        (s[6] & 32) >> 3 | (s[2] & 1024) >> 7 |
        (s[4] & 256) >> 4 | (s[0] & 1024) >> 5 |
        (s[3] & 4096) >> 6 | (s[7] & 2) << 6;

    t[19] = (s[2] & 16) >> 4 | (s[0] & 16384) >> 13 |
        (s[5] & 16384) >> 12 | (s[2] & (unsigned) 32768) >> 12 |
        (s[8] & 16) | (s[4] & 8192) >> 8;

    t[20] = (s[1] & 8) >> 3 | (s[3] & 8) >> 2 |
        (s[7] & (unsigned) 32768) >> 13 | (s[4] & 4) << 1 |
        (s[0] & 2) << 3 | (s[8] & 16384) >> 9 |
        (s[5] & 4) << 4 | (s[1] & 16384) >> 7;

    t[21] = (s[0] & 64) >> 6 | (s[3] & 256) >> 7 |
        (s[7] & 32) >> 3 | (s[5] & 256) >> 5 |
        (s[6] & 2048) >> 7 | (s[1] & 512) >> 4;

    t[22] = (s[7] & 4) >>2 | (s[4] & 1) << 1 |
        (s[6] & 1) << 2 | (s[1] & 1) << 3 |
        (s[2] & 256) >> 4;

    t[23] = (s[3] & 2048) >>11 | (s[6] & 8192) >> 12 |
        (s[5] & 128) >> 5 | (s[0] & 256) >> 5 |
        (s[7] & 2048) >> 7;

    t[24] = (s[3] & 64) >>6 | (s[1] & 256) >>7 |
        (s[3] & 2) << 1 | (s[8] & 128) >> 4 |
        (s[4] & 128) >> 3;

    t[25] = (s[2] & 1)  | (s[8] & 4) >> 1 |
        (s[6] & 256) >> 6 | (s[5] & 1) << 3 |
        (s[0] & 1) << 4;


}
```

```
/*****************************************************************************************************/
/* SYNTH.C                                                                                           */
/*  EVM PROGRAM FOR SYNTHESIZING 240 SAMPLES OF SPEECH FROM 144                                      */
/*  BITS (9 WORDS) CELP PARAMETERS, DRIECTLY FROM TRANSMISSION                                       */
/*  PROGRAMMED BY ARMEIN LANGI                                                                       */
/*  UNIVERSITY OF MANITOBA                                                                           */
/*  @ 1990, 1991, 1992                                                                               */
/*****************************************************************************************************/
#include <math.h>
#include "synth1.h"                 /* DEFINE GLOBAL DATA                                            */
#include "synth2.h"                 /* DEFINE PROTOTYPES                                             */

/*---------------------------------------------------------------------------------------------------*/
/* pcm[] IS THE SPEECH SAMPLES                                                                       */
/* stream[] IS THE CELP PARAMETERS                                                                   */
/*---------------------------------------------------------------------------------------------------*/
void   Synthesize(unsigned *stream, int *pcm)
{
unsigned SubFrame,i;

    ConvertFromStream(stream, SynthCELP);
    DeltaDecoding(SynthCELP+19);
    DeltaDecoding(SynthCELP+21);

    for (SubFrame = 0; SubFrame < 4; SubFrame++)
    {
        InterpolateLSP(TableLSP,SynthCELP, SynthPreviousLSP,
            SubFrame, SynthFloatLSPP, SynthFloatLSPQ);
        ConvertLSPToLPC(SynthFloatLSPP, SynthFloatLSPQ, SynthLPC,
            TableLSP);
        SynthCELP[18 + SubFrame] = SynthCELP[18 + SubFrame] + 20;

        UpdateACB(ACBGainTable,SCBGainTable,SynthLPC,
            SynthCELP+10+SubFrame, SynthACB,SCB,
            SynthExcitationPulses);
        GetDelayElement(SynthLPC,SynthExcitationPulses,
            SynthDelayElements,pcm+(SubFrame*60));
/*    ConvertToPCM(SyntheticSpeech, SubFrame, pcm);                                                  */
    }
    UpdatePreviousLSP(SynthCELP,SynthPreviousLSP);
}

/*---------------------------------------------------------------------------------------------------*/
/* SINCE THE SYNTHESIZER PRODUCES FLOATING POINT NUMBERS,                                            */
/* WE CONVERT THEM TO INTEGER DATA FOR D/A                                                           */
/* pcm[] IS THE INTEGER SYNTHETIC SPEECH SAMPLES                                                     */
/* speech IS THE FLOATING POINT SYNTHETIC DATA                                                       */
/* SubFrame IS THE CURRENT SUBFRAME OF 60 SAMPLES                                                    */
/*---------------------------------------------------------------------------------------------------*/
void ConvertToPCM(float *speech,unsigned SubFrame, int *pcm)
{
unsigned i;
    for(i = 0; i < 60 ; i++)
    {
        pcm[i+(SubFrame*60)] = (int) speech[i];
    }
}
```

```
/*-------------------------------------------------------------------------------------------------*/
/* DELTA DECODING TO OBTAIN THE ACTUAL ACB ENTRIES OF EVEN FRAME                                   */
/*-------------------------------------------------------------------------------------------------*/
void DeltaDecoding(unsigned *intrcelp)
{
    if(*(intrcelp-1) > 31 && *(intrcelp-1) <95)
    {
        *intrcelp = *intrcelp + *(intrcelp-1) - 31;
    }
    if(*(intrcelp-1) >= 95)
    {
        *intrcelp = *intrcelp + 64;
    }
}
```

```
/*****************************************************************************************/
/* SYNTH1.H                                                                              */
/* INCLUDE FILE FOR SYNTH.C                                                              */
/* DEFINING GLOBAL DATA                                                                  */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/

extern unsigned SynthCELP[26];
extern unsigned SynthPreviousLSP[10];
extern float    SynthFloatLSPQ[6];
extern float    SynthFloatLSPP[6];
extern float    SynthLPC[11];
extern float    SynthACB[148];
extern float    SynthDelayElements[60];
extern float    SynthExcitationPulses[60];

extern float    TableLSP[];
extern float    ACBGainTable[32];
extern float    SCBGainTable[32];
extern float    SCB[1082];
extern float    SyntheticSpeech[60];
```

```
/*****************************************************************************************/
/* SYNTH2.H                                                                              */
/* INCLUDE FILE FOR SYNTH.C                                                              */
/* DEFINING PROTOTYPES                                                                   */
/* PROGRAMMED BY ARMEIN LANGI                                                            */
/* UNIVERSITY OF MANITOBA                                                                */
/* @ 1990, 1991, 1992                                                                    */
/*****************************************************************************************/
void ConvertFromStream(unsigned *s, unsigned *t);
void DeltaDecoding(unsigned *intrcelp);
void InterpolateLSP(float *LSP,unsigned *NewLSP, unsigned *OldLSP,
    unsigned SubFrame, float *xp, float *xq);
void ConvertLSPToLPC(float *xp, float *xq, float *a_de,float *lsp);
void UpdatePreviousLSP(unsigned *IntrCELP,unsigned *OldLSP);
void UpdateACB(float *a_gain,float *s_gain,float *trcelp,unsigned *SCBEntry,
    float *acb,float *scb,float *ex);
void GetDelayElement(float *a,float *ex,float *yl,int *sspeech);
void ConvertToPCM(float *synpeech, unsigned k, int *pcm);
```

```
/*****************************************************************************/
/* MAKEFILE                                                                  */
/*****************************************************************************/
PROJ   =CELP
CC  =cl30
AS  =asm30
CFLAGS_G   = -s
CFLAGS_D   = -ic:\tms320\c30\include
CFLAGS_R   =
CFLAGS  =$(CFLAGS_G) $(CFLAGS_D)
AFLAGS_G   =
AFLAGS_D   =
AFLAGS_R   =
AFLAGS  =$(AFLAGS_G) $(AFLAGS_D)
LFLAGS_G   =
LFLAGS_D   =
LFLAGS_R   =
LFLAGS  =$(LFLAGS_G) $(LFLAGS_D)
RUNFLAGS   =
OBJS_EXT =
LIBS_EXT =

.c.obj: ; $(CC) $(CFLAGS) $*.c

all:   $(PROJ).OUT

celp.obj:  celp.c

c30.obj:   c30.c


$(PROJ).OUT:   celp.obj \
          c:\armein\tms\test\c\init.obj \
          c:\armein\tms\test\c\analyze.obj \
          c:\armein\tms\test\c\lpc.obj \
          c30.obj \
          c:\armein\tms\test\c\find_lsp.obj \
          c:\armein\tms\test\c\stream.obj \
          c:\armein\tms\test\c\find_acb.obj \
          c:\armein\tms\test\c\find_scb.obj \
          c:\armein\tms\test\c\do_trans.obj \
          c:\armein\tms\test\c\synth.obj $(OBJS_EXT)
     echo >NUL @<<$(PROJ).crf
celp.obj +
analyze.obj +
do_trans.obj +
find_acb.obj +
find_lsp.obj +
find_scb.obj +
lpc.obj +
init.obj+
getpcm.obj+
stream.obj +
$(OBJS_EXT)
$(PROJ).OUT

$(LIBS_EXT);
```

```
<<
    lnk30 $(PROJ).cmd

run: $(PROJ).OUT
    $(PROJ).bat
```

```
/*************************************************************************/
/*  CELP.CMD                                                             */
/*                                                                       */
/*  TMS320C30 EVALUATION DEMO PROGRAM LINK COMMAND FILE                  */
/*                                                                       */
/*  (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                 */
/*************************************************************************/
/*  LINK COMMAND FILE FOR LPC10 PROGRAM                                  */
/*************************************************************************/
-c
c30
celp
c:\armein\tms\test\c\lpc
c:\armein\tms\test\c\find_lsp
c:\armein\tms\test\c\find_acb
c:\armein\tms\test\c\find_scb
c:\armein\tms\test\c\do_trans
c:\armein\tms\test\c\stream
c:\armein\tms\test\c\synth
c:\armein\tms\test\c\analyze
c:\armein\tms\test\c\init

-o celp.out
-m celp.map
-l c:\tms320\c30\lib\rts.lib

MEMORY
{
  VECS:  org = 0       len = 0x40     /* RESERVED VECTOR LOCATIONS          */
  SRAM:  org = 0x40    len = 0x3FC0   /* PRIMARY BUS SRAM (16K)             */
  RAM :  org = 0x809800 len = 0x800   /* INTERNAL RAM (2K)                  */
}

SECTIONS
{
  .text:  {} > SRAM              /* CODE                                    */
  .cinit: {} > SRAM              /* C INITIALIZATION TABLES                 */
  .stack: {} >  RAM              /* STACK (MODIFIED TO 64 WDS)*/
  .bss:   {} > SRAM              /* C VARIABLES                             */
  .data:  {} > SRAM              /* ASSEMBLY CODE CONSTANTS                 */
  vecs:   {} > VECS              /* RESET/INTERRUPT VECTORS                 */
     sdata: {} > SRAM
     yzdelay    align(16): {} >  RAM
     zdelay     align(16):  {}   >   RAM
     xkdelay    align(16):  {}   >   RAM
     Sdelay     align(16):  {}   >   RAM
     ACB_src          align(256):  {}   >   RAM
     SACB_data  align(256): {} >RAM
     y_n_buff         align(16)   :   {}   >   RAM
}
```

```
*******************************************************
MAP FILE PRODUCED BY TMS320C30 Linker , Version 4.00
*******************************************************
Wed Feb 26 10:02:48 1992

OUTPUT FILE NAME:  <celp.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 0000131f


MEMORY CONFIGURATION

         name    origin    length    attributes
        ------  --------  --------  ----------
VECS     00000000  000000040    RWIX
SRAM     00000040  000003fc0    RWIX
RAM      00809800  000000800    RWIX


SECTION ALLOCATION MAP

output                                  attributes/
section  page    origin      length       input sections
-------  ----  ----------  ----------  ----------------
vecs     0    00000000  0000000c
                  00000000  0000000c    c:\armein\tms\test\c\init.obj (vecs)

.text    0    00000040  0000133f
                  00000040  000000c2    c30.obj (.text)
                  00000102  00000173    celp.obj (.text)
                  00000275  000000bd    c:\armein\tms\test\c\lpc.obj (.text)
                  00000332  0000042c    c:\armein\tms\test\c\find_lsp.obj (.text)
                  0000075e      000002e1    c:\armein\tms\test\c\find_acb.obj (.text)
                  00000a3f  000001d0    c:\armein\tms\test\c\find_scb.obj (.text)
                  00000c0f  0000008a    c:\armein\tms\test\c\do_trans.obj (.text)
                  00000c99  00000463    c:\armein\tms\test\c\stream.obj (.text)
                  000010fc  00000090    c:\armein\tms\test\c\synth.obj (.text)
                  0000118c  0000014a    c:\armein\tms\test\c\analyze.obj (.text)
                  000012d6  00000047    c:\armein\tms\test\c\init.obj (.text)
                  0000131d  00000019    c:\tms320\c30\lib\rts.lib (.text)
                  00001336  00000021    c:\tms320\c30\lib\rts.lib (.text)
                  00001357  00000028    c:\tms320\c30\lib\rts.lib (.text)

.cinit   0    0000137f  0000006b
                  0000137f  00000024    c30.obj (.cinit)
                  000013a3  00000007    celp.obj (.cinit)
                  000013aa  00000004    c:\armein\tms\test\c\lpc.obj (.cinit)
                  000013ae  00000003    c:\armein\tms\test\c\find_lsp.obj (.cinit)
                  000013b1  00000006    c:\armein\tms\test\c\find_acb.obj (.cinit)
                  000013b7  00000004    c:\armein\tms\test\c\find_scb.obj (.cinit)
                  000013bb  00000003    c:\armein\tms\test\c\stream.obj (.cinit)
                  000013be  0000000e    c:\armein\tms\test\c\synth.obj (.cinit)
                  000013cc  00000017    c:\armein\tms\test\c\analyze.obj (.cinit)
                  000013e3  00000006    c:\tms320\c30\lib\rts.lib (.cinit)
                  000013e9  00000001    --HOLE-- [fill = 00000000]

.bss     0    000013ea  000007ea    UNINITIALIZED
                  000013ea  00000011    c30.obj (.bss)
```

```
                    000013fb   000000fc    celp.obj (.bss)
                    000014f7   000000fd    c:\armein\tms\test\c\lpc.obj (.bss)
                    000015f4   00000001    c:\armein\tms\test\c\find_lsp.obj (.bss)
                    000015f5   00000123    c:\armein\tms\test\c\find_acb.obj (.bss)
                    00001718   00000478    c:\armein\tms\test\c\find_scb.obj (.bss)
                    00001b90   00000001    c:\armein\tms\test\c\stream.obj (.bss)
                    00001b91   0000000c    c:\armein\tms\test\c\synth.obj (.bss)
                    00001b9d   00000015    c:\armein\tms\test\c\analyze.obj (.bss)
                    00001bb2   00000022    c:\tms320\c30\lib\rts.lib (.bss)

.data    0    00001bd4   00000da6
                    00001bd4   00000da6    c:\armein\tms\test\c\init.obj (.data)

.stack   0    00809800   00000400    UNINITIALIZED
                    00809800   00000400    c:\tms320\c30\lib\rts.lib (.stack)

sdata    0    00000040   00000000    UNINITIALIZED

yzdelay  0    00809800   00000000    UNINITIALIZED

zdelay   0    00809800   00000000    UNINITIALIZED

xkdelay  0    00809800   00000000    UNINITIALIZED

Sdelay   0    00809800   00000000    UNINITIALIZED

ACB_src  0    00809800   00000000    UNINITIALIZED

SACB_dat 0    00809800   00000000    UNINITIALIZED

y_n_buff 0    00809800   00000000    UNINITIALIZED


GLOBAL SYMBOLS

address  name                          address  name
-------  ----                          -------  ----
000013ea .bss                          00000040 _c_int11
00001bd4 .data                         00000040 .text
00000040 .text                         00000054 _c_int99
00001336 DIV_F                         00000057 _init_evm
0000075e _ACBSearchingEven             0000005f _init_aic
00002611 _ACB                          000000c9 _configure_aic
00002197 _ACBGainTable                 000000d8 _recording
000008d2 _ACBSearchingOdd              000000ed _playing
00001edb _ACBSearchingTarget           00000102 _main
0000118c _Analyze                      00000109 _c_int05
00000275 _AnalyzeLPC                   00000146 _init_arrays
000026a5 _BandwidthExpandingFactor     00000149 _wait_buffer
00001d2c _CELP                         0000015c _command_process
0000072b _CheckLSPStability            00000228 _SignAdjust
000011cc _CodebookSearching            00000245 _ReadAD
00000ecb _ConvertFromStream            0000025c _WriteDA
00000332 _ConvertLSPToLPC              00000275 _AnalyzeLPC
00000c99 _ConvertToDataStream          000002a7 _LevinsonDurbin
000004e0 _ConvertToLSP                 000002fb _FindImpulseResponse
00001162 _ConvertToPCM                 00000332 _ConvertLSPToLPC
```

000015e7 _CorrelationCoefficients
00001d51 _DelayElements
00001175 _DeltaDecoding
000012af _DeltaEncodingACB
00001303 _EraseBuffers
00001f8f _ExcitationPulses
00001288 _ExpandBandwidth
00001ed0 _ExpandedLPC
00000a02 _FindACBSearchingTarget
000002fb _FindImpulseResponse
00000b73 _FindSCBSearchingTarget
00001ec4 _FloatLSPP
00001eca _FloatLSPQ
00000c60 _GetDelayElement
00002007 _HammingWindowData
0000131b _InitializeAllParameters
000006a9 _InterpolateLSP
00001d46 _LPC
000002a7 _LevinsonDurbin
00001d5c _OriginalSpeech
000012d6 _PCMToFloat
00001d22 _PreviousLSP
00000bcc _Quantize_gain
00000245 _ReadAD
000012f3 _ReturnFloat
000012e6 _ReturnInteger
000021b7 _SCBGainTable
00001f17 _SCBSearchingTarget
000021d7 _SCB
00000a3f _SCBSearching
000012a0 _ShiftTheOriginalSpeech
00000228 _SignAdjust
00001bd4 _StackHolder
00001c32 _SynthACB
00001d02 _SynthCELP
00001bd5 _SynthDelayElements
00001cc6 _SynthExcitationPulses
00001c26 _SynthFloatLSPP
00001c2c _SynthFloatLSPQ
00001c1b _SynthLPC
00001c11 _SynthPreviousLSP
000010fc _Synthesize
00001fcb _SyntheticSpeech
000020f7 _TableLSP
00000c0f _UpdateACB
000012c6 _UpdatePreviousLSP
00001f53 _WeightedImpulseResponse
000014f7 _WindowedSpeech
0000025c _WriteDA
0000137a _abort
000013f6 _aic_command_0
000013f7 _aic_command_1
000013f8 _aic_command_2
000013f9 _aic_command_3
000013ef _aic_secondary
00001367 _atexit
000015f5 _bestPeak
000013fc _buffer9
000004e0 _ConvertToLSP
00000688 _sumpq
000006a9 _InterpolateLSP
0000072b _CheckLSPStability
0000075e _ACBSearchingEven
000008d2 _ACBSearchingOdd
00000945 _find_code_20
00000992 _find_code_big
000009ae _find_code_small
00000a02 _FindACBSearchingTarget
00000a3f _SCBSearching
00000a92 _joint_op
00000ae6 _find_entry_0
00000b2b _find_entry_more
00000b73 _FindSCBSearchingTarget
00000bcc _Quantize_gain
00000c0f _UpdateACB
00000c60 _GetDelayElement
00000c99 _ConvertToDataStream
00000ecb _ConvertFromStream
000010fc _Synthesize
00001162 _ConvertToPCM
00001175 _DeltaDecoding
0000118c _Analyze
000011cc _CodebookSearching
00001288 _ExpandBandwidth
000012a0 _ShiftTheOriginalSpeech
000012af _DeltaEncodingACB
000012c6 _UpdatePreviousLSP
000012d6 _PCMToFloat
000012e6 _ReturnInteger
000012f3 _ReturnFloat
00001303 _EraseBuffers
0000131b _InitializeAllParameters
0000131f _c_int00
00001336 DIV_F
00001357 _exit
00001367 _atexit
0000137a _abort
0000137f cinit
0000137f etext
000013ea _dma_int0
000013ea .bss
000013eb _dma_int1
000013ec _dma_int2
000013ed _send_command
000013ee _secondary_transmit
000013ef _aic_secondary
000013f0 _serial_port
000013f1 _timer
000013f2 _bus
000013f3 _dma
000013f4 _host
000013f5 _buffer
000013f6 _aic_command_0
000013f7 _aic_command_1
000013f8 _aic_command_2
000013f9 _aic_command_3

```
00001405 _buffer240
000013f5 _buffer
000013f2 _bus
0000131f _c_int00
00000109 _c_int05
00000040 _c_int11
00000054 _c_int99
0000015c _command_process
000000c9 _configure_aic
000013f3 _dma
000013ea _dma_int0
000013eb _dma_int1
000013ec _dma_int2
00001357 _exit
00000945 _find_code_20
00000992 _find_code_big
000009ae _find_code_small
00000ae6 _find_entry_0
00000b2b _find_entry_more
000013f4 _host
000013fb _index
0000005f _init_aic
00000146 _init_arrays
00000057 _init_evm
000026a8 _input
000026a7 _intermediate
00000a92 _joint_op
00000102 _main
000026a9 _output
000000ed _playing
000000d8 _recording
000013ee _secondary_transmit
000013ed _send_command
000013f0 _serial_port
00000688 _sumpq
000013f1 _timer
000015f6 _va
000016b1 _vas
00001718 _vh
00000149 _wait_buffer
00001b52 _xa
0000137f cinit
0000297a edata
00001bd4 end
0000137f etext

[123 symbols]
```

```
000013fb _index
000013fc _buffer9
00001405 _buffer240
000014f7 _WindowedSpeech
000015e7 _CorrelationCoefficients
000015f5 _bestPeak
000015f6 _va
000016b1 _vas
00001718 _vh
00001b52 _xa
00001bd4 .data
00001bd4 end
00001bd4 _StackHolder
00001bd5 _SynthDelayElements
00001c11 _SynthPreviousLSP
00001c1b _SynthLPC
00001c26 _SynthFloatLSPP
00001c2c _SynthFloatLSPQ
00001c32 _SynthACB
00001cc6 _SynthExcitationPulses
00001d02 _SynthCELP
00001d22 _PreviousLSP
00001d2c _CELP
00001d46 _LPC
00001d51 _DelayElements
00001d5c _OriginalSpeech
00001ec4 _FloatLSPP
00001eca _FloatLSPQ
00001ed0 _ExpandedLPC
00001edb _ACBSearchingTarget
00001f17 _SCBSearchingTarget
00001f53 _WeightedImpulseResponse
00001f8f _ExcitationPulses
00001fcb _SyntheticSpeech
00002007 _HammingWindowData
000020f7 _TableLSP
00002197 _ACBGainTable
000021b7 _SCBGainTable
000021d7 _SCB
00002611 _ACB
000026a5 _BandwidthExpandingFactor
000026a7 _intermediate
000026a8 _input
000026a9 _output
0000297a edata
```

APPENDIX C

SETS OF UTTERANCES AND QUESTIONAIRE

SETS OF UTTERANCES

1. Utterances for Word Intelligibility Test

The utterances are:

Consonant	Main Word	OW-1	OW-2	OW-3	OW-4
/s/	sale	male	tale	pale	bale
/t/	top	hop	pop	mop	cop
/b/	back	lack	pack	jack	sack
/m/	moon	noon	coon	boon	soon
/l/	law	saw	jaw	paw	raw
/p/	page	rage	cage	sage	wage
/r/	rock	dock	mock	cock	lock
/w/	went	sent	rent	bent	tent
/k/	kill	fill	will	till	bill
/h/	heel	peel	reel	feel	keel
/f/	fight	light	right	might	night
/d/	day	pay	may	way	say
/n/	not	hot	got	pot	lot
/g/	god	rod	cod	sod	nod
/dz/	just	bust	must	rust	dust
/v/	vile	mile	file	tile	pile
/j/	yet	set	let	get	met
/z/	zeal	heal	deal	seal	meal

OW = other word

2. Utterances for Sentence Intelligibility Test

We use the same sentences from sets M and F described next in subjective quality test. The selected sentences from set M are sentences no. 4, 9, 10, 11, and 12. The selected sentences from set F are sentences no. 1, 8, 9, 10, and 12.

3. Utterances for Subjective Quality Test

Set M (spoken by a male)

1. (28) The soft cushion broke the man's fall.
2. (2) Glue the sheets to the dark blue background.
3. (113) The pipe began to rust while new.
4. (147) Write a fond note to the friend you cherish.
5. (1) The birch canoe slid on the smooth plank.
6. (230) The wreck occured by the bank on Main Street.
7. (231) A pencil with black lead writes best.
8. (273) The train brought our hero to the big town.
9. (284) The red tape bound the smuggled food.
10. (311) Slide the box into that empty space.
11. (167) The pirates seized the crew of the lost ship. (used for anchoring)
12. (89) The pearl was worn in a thin silver ring. (used for anchoring)

Set F (spoken by a female)

1. (360) Next Tuesday we must vote.
2. (364) The dirt piles were lines along the road.
3. (428) The sand drifts over the still of the old house.
4. (439) The straw nest housed five robins.
5. (4) These days a chicken leg is a rare dish.

6. (488) The youth drove with zest, but little skill.
7. (531) Press the pedal with your left foot.
8. (572) The child crawled into the dense grass.
9. (591) Every word and phrase he speaks is true.
10. (619) A six comes up more often than a ten.
11. (3) It's easy to tell the depth of a well. (used for anchoring)
12. (450) The marsh will freeze when cold enough. (used for anchoring)

QUESTIONAIRE

Thank you for participating in these speech *intelligibility* and *quality* tests. The objective of these tests is to asses the performance of a speech processing system.

The tests, consisting of three parts, should take approximately 25 minutes.

They are a word intelligibility test, a subjective quality test, and a sentence intelligibility test.

Part 1. Word Intelligibility Test.

You are going to hear 18 repeated blocks each made up of a spoken test number, a 3 second pause, followed by a word, which is one of the five words written in your answering sheets. You are asked:

- to identify the spoken word, by marking one of the blank space corresponding to the word, and
- to indicate the difficulty to identify the correct word, by marking the corresponding difficulty level.

The difficulty levels are as follows.

Levels	Difficulties
D	Difficult
DE	Fairly Difficult/Fairly Easy
E	Easy

At the end of the word, a pause of 5 seconds preceeds the next block.

Working Sheet

Test number 1

__ male __ pale __ tale __ sale __ bale D DE E

Test number 2

__ hop __ cop __ pop __ mop __ top D DE E

Test number 3

__ pack __ lack __ back __ jack __ sack D DE E

Test number 4

__ noon __ coon __ moon __ boon __ soon D DE E

Test number 5

__ raw __ saw __ law __ paw __ jaw D DE E

Test number 6

__ page __ rage __ cage __ sage __ wage D DE E

Test number 7

__ dock __ rock __ mock __ cock __ lock D DE E

Test number 8

__ bent __ sent __ rent __ went __ tent D DE E

Test number 9

__ bill __ fill __ will __ till __ kill D DE E

Test number 10

__ peel __ heel __ reel __ feel __ keel D DE E

Test number 11

__ fight __ light __ right __ might __ night D DE E

Test number 12

__ say __ pay __ may __ way __ day D DE E

Test number 13

__ pot __ hot __ got __ not __ lot D DE E

Test number 14

__ nod __ rod __ cod __ sod __ god D DE E

Test number 15

__ just __ bust __ must __ rust __ dust D DE E

Test number 16

__ tile __ mile __ file __ vile __ pile D DE E

Test number 17

__ set __ yet __ let __ get __ met D DE E

Test number 18

__ deal __ heal __ zeal __ seal __ meal D DE E

Part 2. Subjective Quality Test.

You are going to hear 20 repeated blocks each made up of a spoken test number, a 3 second pause, followed by a spoken sentence. You are to judge the quality of the speech you hear. You are asked **to mark** one of the blank space corresponding to the **quality** you perceive. The quality ratings are as follows.

EXCELLENT
GOOD
FAIR
POOR
UNSATISFASCTORY

At the end of the word, a pause of 5 seconds preceeds the next block.

Prior to and in the middle of the test, you are to hear two spoken sentences as references. The first sentence is associated with EXCELLENT, while the second sentence is associated with UNSATISFACTORY.

First Referencing (Anchoring)

The next sentence is to be judged as EXCELLENT
"The pirates seized the crew of the lost ship"

The next sentence is to be judged as UNSATISFACTORY
"The pearl was worn in a thin silver ring"

Working Sheet

Test number 1
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 2
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 3
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 4
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 5
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 6
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 7
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 8
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 9
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 10
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Second Referencing (Anchoring)

The next sentence is to be judged as EXCELLENT
"It's easy to tell the depth of a well"

The next sentence is to be judged as UNSATISFACTORY
"The marsh will freeze when cold enough"

Test number 11
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 12
__EXCELLENT __GOOD __FAIR __POOR __UNSATISFACTORY

Test number 13

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 14

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 15

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 16

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 17

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 18

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 19

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Test number 20

__ EXCELLENT __ GOOD __ FAIR __ POOR __ UNSATISFACTORY

Part 3. Sentence Intelligibility Test.

You are going to hear the 10 blocks each made up of a spoken test number, a 3 second pause, followed by a spoken sentence, taken from the previous test. You are asked **to write down** the sentence that you hear.

At the end of the word, a pause of 5 seconds preceeds the next block.

Working Sheet

Test number 1

Test number 2

Test number 3

Test number 4

Test number 5

Test number 6

Test number 7

Test number 8

Test number 9

Test number 10

The test is over. Thank you for your participation. Please fill the following blank spaces. These data are needed to accompany the test results.

You are a	__ Male / __ Female
Your age is	____ years old.
You are a	__ Student / ___ Professor / ___ Other