FINAL REPORT

ADAPTIVE MULTILEVEL 16KB/S SPEECH CODER

DCA CONTRACT DCA-100-76-C-002

14 JUNE 1976

GTE SYLVANIA INCORPORATED
ELECTRONIC SYSTEMS GROUP
EASTERN DIVISION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited
This report describes software designed by GTE Sylvania for a high quality 16,000 bit per sec. speech terminal. This software operates in a full duplex mode on two Sylvania Programmable Signal Processors.
FINAL REPORT

ADAPTIVE MULTILEVEL 16KB/S SPEECH CODER

DCA CONTRACT DCA-100-76-C-0002

14 June 1976

GTE SYLVANIA
INCORPORATED
ELECTRONIC SYSTEMS GROUP
EASTERN DIVISION
77 "A" STREET
NEEDHAM HEIGHTS, MASSACHUSETTS 02194

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>1</td>
</tr>
<tr>
<td>List of Illustrations</td>
<td>1-iv</td>
</tr>
<tr>
<td>1 ADAPTIVE MULTILEVEL 16 KBPS SPEECH CODER</td>
<td>1</td>
</tr>
<tr>
<td>1.0 Introduction</td>
<td>1-3</td>
</tr>
<tr>
<td>2 SOFTWARE SIMULATIONS</td>
<td>4</td>
</tr>
<tr>
<td>2.0 Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Operating Principles of the APCQ Terminal</td>
<td>5-15</td>
</tr>
<tr>
<td>2.2 Error Signal Quantizers</td>
<td>16-18</td>
</tr>
<tr>
<td>2.2.1 The Jayant Quantizer</td>
<td>19-21</td>
</tr>
<tr>
<td>2.2.2 The Modified Jayant Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>2.2.3 The Forney Quantizer</td>
<td>23-25</td>
</tr>
<tr>
<td>2.2.4 The Fixed/Frame Quantizer</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Fortran Fixed-Point Simulations</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Simulation Results</td>
<td>27</td>
</tr>
<tr>
<td>2.4.1 Coder Performance Without Channel Errors</td>
<td>27-32</td>
</tr>
<tr>
<td>2.4.2 Coder Performance with Channel Errors</td>
<td>32-37</td>
</tr>
<tr>
<td>2.4.3 Improved Error Signal Quantizers</td>
<td>37-42</td>
</tr>
<tr>
<td>3 REAL-TIME SOFTWARE</td>
<td>43</td>
</tr>
<tr>
<td>3.1 Description of APCQ Program</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1 General Discussion</td>
<td>43-45</td>
</tr>
<tr>
<td>3.1.2 Initialization and Wait Loop</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3 Synchronization</td>
<td>46-48</td>
</tr>
<tr>
<td>3.1.4 Line-side Interrupt</td>
<td>48-51</td>
</tr>
<tr>
<td>3.1.5 Speech Side Interrupt</td>
<td>51</td>
</tr>
<tr>
<td>3.1.6 Speech Analysis and Synthesis</td>
<td>51-62</td>
</tr>
<tr>
<td>3.2 Operation Procedures for the APCQ Program</td>
<td>63</td>
</tr>
<tr>
<td>3.2.1 Connecting the System</td>
<td>63</td>
</tr>
<tr>
<td>3.2.2 Reading in the EDM Software</td>
<td>63-66</td>
</tr>
<tr>
<td>3.2.3 Initializing the EDM</td>
<td>66</td>
</tr>
<tr>
<td>3.3 Description of Continuous Variable Slope Delta Modulation Program</td>
<td>67</td>
</tr>
<tr>
<td>3.3.1 Basic Principles of Operation</td>
<td>67-69</td>
</tr>
<tr>
<td>3.3.2 Standard Three Bit Memory (CVSD-B)</td>
<td>69-70</td>
</tr>
<tr>
<td>3.3.3 Forney and Qreshi's 3 Bit Memory (CVSD-C)</td>
<td>71-73</td>
</tr>
<tr>
<td>3.3.4 Jayant's One Bit Memory (CVSD-D)</td>
<td>73</td>
</tr>
<tr>
<td>3.4 Discussion of Real-Time Code</td>
<td>73-74</td>
</tr>
<tr>
<td>3.4.1 Wait Loop</td>
<td>75</td>
</tr>
<tr>
<td>3.4.2 The Line Side and Speech Side Interrupts</td>
<td>75</td>
</tr>
<tr>
<td>3.4.3 Speech Analysis and Synthesis</td>
<td>75</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (cont.)

Section                                    Page

3.5 Operating Procedures of CVSD Program    75
   3.5.1 Connecting the System               75
   3.5.2 Reading in the EDM Software         75
   3.5.3 Initialization of the EDM            75-84
   3.5.4 Program Options                     85-86
3.6 I/O Speech Filters                      87-88

4 LSI IMPLEMENTATION STUDY                  89
   4.1 Introduction                          89-91
   4.2 Basic LSI Signal Processor Architecture 91-95
   4.3 Estimates of LSI Implementation of APCQ Algorithm 95
       4.3.1 APCQ Implementation               95-99
       4.3.2 Examples                         99-107
       4.3.3 Timing and Memory Estimates       108-111
       4.3.4 Cost and Power Estimates          112
       4.4 Other LSI Processor Structures      112-114
   4.4.1 Timing and Memory Estimates          114-119
       4.4.2 Estimates of Cost and Power of Index Processor and Multiplier 120
   4.5 Discussions                           120

5 RECOMMENDATIONS AND CONCLUSIONS           121
   5.1 Conclusions                           121-122
   5.2 Recommendations                       122-123

REFERENCES                                  124

APPENDIX A MORE DETAILED FLOW CHARTS OF APCQ CODER SIMULATION     A1
   A.1 Introduction                           A1
   A.2 Main Program Flow Charts and Listings  A1-A18
   A.3 Quantization of the Error Signal       A18-A20
   A.4 Dequantization of the Error Signal     A21
   A.5 Other Subroutines                      A21
       A.5.1 Introduction                      A21-A24
   A.5.2 Subroutines IALPQ and IALPD          A24
   A.5.3 Subroutine IFARQ and IFARDD          A25-A26
   A.5.4 Subroutines RANERR and IXOR          A27-A28
   A.5.5 Subroutine NORMLl                    A29
   A.5.6 Function IADD, ISUB, IPMUL, IIMUL and IFDIV  A29-A31
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Pulse Heights for FORNEY CVGD Quantizer</td>
<td>71</td>
</tr>
<tr>
<td>4-1</td>
<td>The Estimates of Transmitter Timing of APCQ</td>
<td>108</td>
</tr>
<tr>
<td>4-2</td>
<td>Microprogram Memory Requirements</td>
<td>109</td>
</tr>
<tr>
<td>4-3</td>
<td>Macroprogram Memory Requirements</td>
<td>110</td>
</tr>
<tr>
<td>4-4</td>
<td>Materials, Quantities, Power and Cost</td>
<td>111</td>
</tr>
<tr>
<td>4-5</td>
<td>Processor Time Requirements vs. Hardware Complexity</td>
<td>115</td>
</tr>
<tr>
<td>4-6</td>
<td>Memory Requirements vs. Hardware Complexity</td>
<td>117</td>
</tr>
<tr>
<td>4-7</td>
<td>Number of Memory Chips</td>
<td>118</td>
</tr>
<tr>
<td>4-8</td>
<td>Cost and Power of Various Forms of LSI Processors</td>
<td>119</td>
</tr>
<tr>
<td>A-1</td>
<td>Variable Names Used in Main Program of APCQ Coder</td>
<td>A6&amp;A16-17</td>
</tr>
<tr>
<td>A-2</td>
<td>Number of Transmission Bits for Each Parameter</td>
<td>A17</td>
</tr>
<tr>
<td>A-3</td>
<td>Variable Names Used in Subroutine EDQNT3</td>
<td>A18</td>
</tr>
<tr>
<td>A-4</td>
<td>Variable Names Used in Subroutine EEDNT3</td>
<td>A21</td>
</tr>
<tr>
<td>A-5</td>
<td>Subroutines and Their Functions</td>
<td>A24</td>
</tr>
<tr>
<td>A-6</td>
<td>Variable Names Used in Subroutine RANERR</td>
<td>A28</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>2-1</td>
<td>Simplified APCQ Coder</td>
<td>6</td>
</tr>
<tr>
<td>2-2</td>
<td>Simplified APCQ System</td>
<td>10</td>
</tr>
<tr>
<td>2-3</td>
<td>Reconfiguration of Analyzer to Minimize Effects of Quantizer</td>
<td>12</td>
</tr>
<tr>
<td>2-4</td>
<td>Relationships Among 16Kb/s Speech Coder Configurations</td>
<td>14</td>
</tr>
<tr>
<td>2-5</td>
<td>8-Level Adaptive Quantizer (Jayant Quantizer)</td>
<td>17</td>
</tr>
<tr>
<td>2-6</td>
<td>Forney Adaptive Quantizer</td>
<td>24</td>
</tr>
<tr>
<td>2-7</td>
<td>Simulation of APCQ Speech Encoding System</td>
<td>28</td>
</tr>
<tr>
<td>2-8</td>
<td>Analyzer Flowchart of Fixed-Point Simulation for APCQ Coder</td>
<td>29</td>
</tr>
<tr>
<td>2-9</td>
<td>Synthesizer Flow Chart of the Fixed-Point Simulation for APCQ Coder</td>
<td>30</td>
</tr>
<tr>
<td>2-10</td>
<td>Signal-to-Noise Measurement</td>
<td>31</td>
</tr>
<tr>
<td>2-11</td>
<td>S/N of CVSD and Adaptive Predictive Coders with Jayant Quantizer (APCQ) vs. Predictor Complexity</td>
<td>33</td>
</tr>
<tr>
<td>2-12</td>
<td>S/N of Different Quantizers versus Bit Error Rate (1st Order Adaptive Predictor W/O Pitch Loop)</td>
<td>34</td>
</tr>
<tr>
<td>2-13</td>
<td>16 Kbps Speech Coder Performance vs. Random Channel Bit Errors (BER)</td>
<td>36</td>
</tr>
<tr>
<td>2-14</td>
<td>S/N of CVSD and 4th-Order Adaptive Predictive Coder (Fixed Quantizer) vs. Bit Error Rate at 16Kbps</td>
<td>38</td>
</tr>
<tr>
<td>2-15</td>
<td>S/N of Adaptive Predictive Coders Using 8-Level Modified Jayant Quantizer</td>
<td>40</td>
</tr>
<tr>
<td>2-16</td>
<td>S/N of CVSD and 4th Order Adaptive Predictive Coder (Modified Jayant Quantizer) versus Bit Error Rate at 16 Kbps</td>
<td>41</td>
</tr>
<tr>
<td>3-1</td>
<td>Characteristics of a 5-Level Quantizer</td>
<td>44</td>
</tr>
<tr>
<td>3-2</td>
<td>Initialization and Wait Loop</td>
<td>46</td>
</tr>
<tr>
<td>3-3</td>
<td>Synchronization Search</td>
<td>47</td>
</tr>
<tr>
<td>3-4</td>
<td>Format of Transmission Data</td>
<td>49</td>
</tr>
<tr>
<td>3-5</td>
<td>Line Side Interrupt</td>
<td>50</td>
</tr>
<tr>
<td>3-6</td>
<td>Interrupt Loop Speech Side</td>
<td>52</td>
</tr>
<tr>
<td>3-7</td>
<td>Analyzer for APCQ Coder</td>
<td>53</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3-8</td>
<td>Autocorrelation Calculation</td>
<td>54</td>
</tr>
<tr>
<td>3-9</td>
<td>Calculation of PARCOR Coefficients ($K_1$) and RMS Power</td>
<td>55</td>
</tr>
<tr>
<td>3-10</td>
<td>Error Signal Calculation and Quantization</td>
<td>56</td>
</tr>
<tr>
<td>3-11</td>
<td>Transmission Data Construction</td>
<td>57</td>
</tr>
<tr>
<td>3-12</td>
<td>Data Reception, Decoding and Dequantization</td>
<td>58</td>
</tr>
<tr>
<td>3-13</td>
<td>APCQ Synthesizer</td>
<td>59</td>
</tr>
<tr>
<td>3-14</td>
<td>Converting PARCOR Coefficients to A's</td>
<td>60</td>
</tr>
<tr>
<td>3-15</td>
<td>AMDF Pitch Calculation</td>
<td>61</td>
</tr>
<tr>
<td>3-16</td>
<td>Subroutine QSRT</td>
<td>62</td>
</tr>
<tr>
<td>3-17</td>
<td>System Test Configuration</td>
<td>64</td>
</tr>
<tr>
<td>3-18</td>
<td>Modem Interface Signals and Interconnection</td>
<td>65</td>
</tr>
<tr>
<td>3-19</td>
<td>Transmitter and Receiver of CVSD-B</td>
<td>68</td>
</tr>
<tr>
<td>3-20</td>
<td>Block Diagram of CVSD-C</td>
<td>72</td>
</tr>
<tr>
<td>3-21</td>
<td>Block Diagram of CVSD-D</td>
<td>74</td>
</tr>
<tr>
<td>3-22</td>
<td>Initialization and Wait Loop</td>
<td>76</td>
</tr>
<tr>
<td>3-23</td>
<td>Interrupt Loop Speech Side</td>
<td>77</td>
</tr>
<tr>
<td>3-24</td>
<td>Interrupt Loop Line Side</td>
<td>78</td>
</tr>
<tr>
<td>3-25</td>
<td>Transmitter for CVSD-B</td>
<td>79</td>
</tr>
<tr>
<td>3-26</td>
<td>Receiver for CVSD-B</td>
<td>80</td>
</tr>
<tr>
<td>3-27</td>
<td>Transmitter for CVSD-C</td>
<td>81</td>
</tr>
<tr>
<td>3-28</td>
<td>Receiver for CVSD-C</td>
<td>82</td>
</tr>
<tr>
<td>3-29</td>
<td>Transmitter for CVSD-D</td>
<td>83</td>
</tr>
<tr>
<td>3-30</td>
<td>Receiver for CVSD-D</td>
<td>84</td>
</tr>
<tr>
<td>3-31</td>
<td>I/O Filter Characteristics</td>
<td>88</td>
</tr>
<tr>
<td>4-1</td>
<td>Architectural Concept</td>
<td>90</td>
</tr>
<tr>
<td>4-2</td>
<td>Loop Processor</td>
<td>92</td>
</tr>
<tr>
<td>4-3</td>
<td>Detailed AMD2901 Microprocessor Block Diagram</td>
<td>94</td>
</tr>
<tr>
<td>4-4</td>
<td>APCQ Transmitter Operations</td>
<td>97</td>
</tr>
<tr>
<td>4-5</td>
<td>Operational Flow Macroprocessor and Loop Processor</td>
<td>101</td>
</tr>
<tr>
<td>4-6</td>
<td>Autocorrelation Function Inner Loop</td>
<td>102</td>
</tr>
<tr>
<td>4-7</td>
<td>AMDF Inner Loop</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>4-8</td>
<td>Reduced Form Inner Loop</td>
<td>104</td>
</tr>
<tr>
<td>4-9</td>
<td>Second Reduced Signal Inner Loop</td>
<td>105</td>
</tr>
<tr>
<td>4-10</td>
<td>Partial PARCOR Inner Loop</td>
<td>107</td>
</tr>
<tr>
<td>4-11</td>
<td>Loop Processor with Added Parallelism</td>
<td>113</td>
</tr>
<tr>
<td>A-1</td>
<td>Flow Chart of Fixed-Point APCQ Analyzer</td>
<td>A2-A4</td>
</tr>
<tr>
<td>A-2</td>
<td>Flow Chart of Fixed-Point APCQ Synthesizer</td>
<td>A5</td>
</tr>
<tr>
<td>A-3</td>
<td>FORTRAN Listing of Main Program of APCQ Coder</td>
<td>A7-A15</td>
</tr>
<tr>
<td>A-4</td>
<td>Flowchart of Subroutine EDQNT3 - Quantizing Error Signal (8-Level Jayant Quantizer)</td>
<td>A19</td>
</tr>
<tr>
<td>A-5</td>
<td>FORTRAN Listing of Error Signal Quantizer</td>
<td>A20</td>
</tr>
<tr>
<td>A-6</td>
<td>Flowchart of Subroutine EDDNT3 - Dequantizing Error Signal (8-Level Jayant Quantizer)</td>
<td>A22</td>
</tr>
<tr>
<td>A-7</td>
<td>FORTRAN Listing of Error Signal Dequantizer</td>
<td>A23</td>
</tr>
<tr>
<td>A-8</td>
<td>FORTRAN Listings of Subroutines IALPQ and IALPD</td>
<td>A25</td>
</tr>
<tr>
<td>A-9</td>
<td>FORTRAN Listing of PARCOR Quantizer (IPARQ) and of PARCOR Dequantizer (IPARD)</td>
<td>A27</td>
</tr>
<tr>
<td>A-10</td>
<td>FORTRAN Listing of Subroutine RANERR and Assembly Listing of IXOR</td>
<td>A29</td>
</tr>
<tr>
<td>A-11</td>
<td>Assembly Language Listing of NORML1</td>
<td>A31</td>
</tr>
<tr>
<td>A-12</td>
<td>Assembly Listings of Fixed-Point Arithmetic Functions</td>
<td>A32</td>
</tr>
</tbody>
</table>
SECTION I

Adaptive Multilevel 16 Kbps Speech Coder

1.0 Introduction

Under the eight month Adaptive Multilevel 16 Kbps Speech Coder contract, (DCA100-76-C-0002), GTE Sylvania developed software for a high quality 16,000 bit/sec speech transmission terminal having an audio bandwidth of 3000 Hz. This software was designed to operate in a full duplex mode on two GTE Sylvania Programmable Signal Processors (PSP) already owned by the Defense Communications Agency.

As a benchmark to evaluate the voice quality of the Adaptive Multilevel Speech Coder, GTE Sylvania also developed full duplex software for Continuous Variable Slope Delta Modulation (CVSD). This CVSD software enabled DCA to modify and optimize the internal operating parameters of CVSD and to experiment with new CVSD algorithms.

The development of these 16 Kbps voice coders was motivated by the expected deployment of a 16 Kbps secure voice network. A necessary part of any transmission network is the subscriber terminal or telephone. For digital communication network the transmitter terminal must convert the analog speech to digital, encrypt the data, receive incoming data, decrypt it and reproduce the speech.

Unfortunately, 16 Kbps is too low a data rate to send voice without the use of data reduction algorithms and these algorithms generally introduce audible distortion. The goal of these studies was to improve the performance existing 16 Kbps CVSD terminals and investigate the potential of new terminals employing the principles of Adaptive Predictive Coding with Adaptive Quantization (APCQ).
The significant results of this contract are threefold:

1. The voice quality of CVSD can be improved through relatively minor changes in the internal parameters or through the use of new CVSD algorithms.

2. The APCQ coder provides voice quality that is noticeably superior to the best CVSD algorithms, and

3. The APCQ coder can perform well even in the presence of channel errors exceeding $1 \times 10^3$.

These findings show that it is technically feasible to improve voice transmission at 16 Kbps over that obtained by using the present CVSD terminals. Using the first result, relatively minor changes in the CVSD telephone subscriber unit can provide voice transmission with less distortion. The second result, indicates that a new APCQ coder in the subscriber terminal could provide voice transmission with almost imperceptible distortion. Although the circuitry needed to implement these adaptive multilevel coders is considerably more complex than that needed for CVSD, studies performed under this contract indicate that largescale integration techniques can minimize this difference in complexity.

In fact, we show that the APCQ coder can be implemented around an LSI microprocessor using approximately $700$ in parts and about $42$ watts of power. Moreover, the processing speed of this microprocessor is sufficiently fast that it can analyze and synthesize 25 msec of speech in 16 msec. Thus $40\%$ of the time the LSI microprocessor can perform other functions such as synchronization, encryption and telephone line control.
Of course, CVSD will always be less costly to implement than our design for the APCQ coder. Nevertheless, our design for implementing the complete subscriber terminal with the APCQ coder may cost no more than the present subscriber terminal employing CVSD. This equality in cost occurs because analog to digital conversion of voice is only one of many features of the subscriber terminal. Telephone line control, encypherment, and synchronization can significantly impact the cost of the subscriber terminal. Our design, based around a microprocessor, provides a cost effective method of implementing these features in software by sharing microprocessor hardware with the APCQ coder. The present CVSD terminal requires separate hardware for each feature, thus raising its total complexity and cost.

Because of the improved voice quality provided by the Adaptive Multilevel coders, further work should be performed to simplify the algorithms, to improve their hardware design, and to test their performance under realistic operating environments more fully.
SECTION 2
Software Simulations

2.0 Introduction

This study consisted of two parallel efforts for developing full duplex voice processing systems. The simulation studies investigated new techniques for coding speech and reducing the data rate and the real-time software development implemented the most promising techniques and several CVSD algorithms on the GTE Sylvania PSP for further evaluation and demonstration. This real-time software was delivered to the Defense Communications Agency for operation on their two PSP's built under a prior contract (DCA100-74-C-0058) with GTE Sylvania.

Specifically, GTE Sylvania

1. Developed simulations of adaptive predictive coders with multilevel Quantizers (APCQ) at 16 Kbps using several forms of error signal quantizers.

2. Studied the performance of these APCQ systems in the presence of channel errors

3. Wrote full duplex APCQ software for the DCA PSP's.

and

4. Wrote full duplex software for three forms of continuous variable slope delta modulation (CVSD)

The remainder of this section will describe the operating principles of APCQ and the error signal quantizers and will provide the results of the FORTRAN simulations. The following section will than describe the operation of the real-time CVSD coder and the real-time APCQ coder.
2.1 Operating Principles of the APCQ Terminal

Figure 2-1 illustrates the principles of a speech transmission based on adaptive predictive coding with adaptive multilevel quantization (APCQ). The APCQ technique estimates or predicts the present input samples \( s_n \) from past history of the waveform, that is \( \hat{s}_n = P(s_{n-1}, s_{n-2}, \ldots) \).

The residual signal or error signal \( e_n = s_n - \hat{s}_n \), along with the function \( P \) provides sufficient information for the receiver to regenerate the input precisely. In general, however, the error signal is quantized and is not sent exactly. This quantization distorts the output speech so the choice of an appropriate method for quantization is important to good quality speech. A properly designed predictor will produce an error signal having less dynamic range and smaller variance than the input. Thus a quantizer operating on the error signal needs fewer bits/sample than one operating on the input, and this significantly lowers the data rate.

The optimum predictor depends upon the statistics of the input and thus the predictor parameters adapt as the input changes. These parameters are chosen to minimize the mean-squared error \( e_n \) between the predicted samples \( \hat{s}_n \) and the actual samples \( s_n \), over an analysis interval or frame of length \( T \) which is typically between 20 and 25 ms long. This frame length is often called the band length. Once calculated, the predictor parameters remain fixed for the entire interval but change from frame to frame as the input statistics change. Thus, during each analysis frame, the algorithms determine the predictor parameters from the input data and form the error signal sequence by filtering the input digitally.
Figure 2-1. Simplified APCQ Coder
For speech, the predictor function $P$ was broken into two separate smaller predictors $P_1$ and $P_2$, as illustrated in Figure 2-2. The form of $P_1$ acknowledges that speech is often quasi-periodic with period $M$. Consequently, $P_1$ estimates speech as $\hat{s}_n = a s_{n-M}$ where the pitch gain $a$ indicates that there is either a gain change from period to period or that the speech is not perfectly correlated with period $M$.

For calculation of the pitch period $M$ we chose the average magnitude difference function (AMDF) \(^{(6)}\)

$$AMDF(j) = \sum_{k=1}^{r} |s_k - s_{k+j}|$$

as a computationally efficient method for its estimation pitch $M$. This function has no multiplications and requires little numerical scaling. It also has a sharp null for that value of $j$ which usually corresponds to the period $M$.

In speech, not all pitch periods are possible and thus the terminal calculates the AMDF for those values of $j$ corresponding to pitch frequencies between 70 and 340 Hz. Accurate pitch extraction requires additional logic to test for false or multiple nulls, but since later processing can partially correct for occasional pitch errors, the period $M$ is set to that value of $j$ corresponding to a minimum value of AMDF. To speed up the processing further, the AMDF calculation in the APCQ system sums over every third value of the input. This causes spectral aliasing and occasional shifts in nulls in the AMDF, but listening tests indicate no impairment in the quality of the resulting speech. Furthermore, in calculating the AMDF, the algorithms form partial summations which are then scaled and added to form the final summation. Overflows are always clamped to the largest number of the processor but this does not affect the minimum values needed for pitch extraction.
To minimize the total squared error \( E = \sum_{n=1}^{T} (s_n - \alpha s_{n-M})^2 \), we can differentiate \( E \) with respect to \( \alpha \) and set the result to zero giving

\[
\alpha = \frac{\sum_{n=1}^{r} s_n s_{n-M}}{\sum_{n=1}^{r} s_n^2}
\]

where \( T \) is the frame baud length in samples.

For periodic sounds, such as vowels, \( \alpha \) is close to unity, but for noiselike consonants which have little correlation \( M \) samples apart, \( \alpha \) is near zero.

The reduced waveform,

\[
v_n = s_n - \alpha s_{n-M}.
\]

or first error signal still contains sufficient redundancy such that a second \( N \)th-order predictor \( P_2 \) can further reduce the output signal energy, especially if the speech is not periodic or if the period is estimated incorrectly. This second predictor uses a weighted sum of \( N \) past samples of the speech waveform to form the estimate

\[
\hat{v}_n = \sum_{i=1}^{N} a_i v_{n-i}
\]

where the \( a_i \)'s are chosen to minimize the squared error

\[
U = \sum_{n=1}^{r} (v_n - \hat{v}_n)^2
\]

and are given by the solution to the matrix equation

\[
\Phi a = C
\]

where

\[
\Phi_{ij} = \sum_{n=1}^{r} v_{n-i} v_{n-j}
\]

\[
C_{i} = \sum_{n=1}^{r} v_{n} v_{n-i}
\]
If we window the reduced waveform so that it is zero outside the interval \(1 < n < T\) (stationarity assumption), we have the autocorrelation normal equations

\[
\phi_i = \sum_{\mu=1}^{T} r_{x}(n-\mu) = R_{i-1}
\]

and

\[
\begin{bmatrix}
R_0 & R_1 & \cdots & R_{N-1} \\
R_1 & R_0 & \cdots & R_{N-2} \\
\vdots & \vdots & \ddots & \vdots \\
R_{N-1} & R_{N-2} & \cdots & R_0
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2 \\
\vdots \\
a_N
\end{bmatrix}
=
\begin{bmatrix}
R_1 \\
R_2 \\
\vdots \\
R_N
\end{bmatrix}
\]

This is a symmetric Toeplitz matrix because the elements along any diagonal parallel to the principal diagonal are identical. Efficient solutions exist that supply the \(a_1\)'s and the mean-square energy \(U\) in the difference signal. In addition, because the elements of the matrix arise from an autocorrelation function, the stationary matrix solution for the \(a_1\)'s will yield a stable filter during synthesis, with the recursive filters shown in Figure 2-2. Unfortunately, the \(a_1\)'s do not make good transmission parameters because quantization or errors in transmission can cause the poles of the receiver filter given by \(1/(1-P_1)(1-P_2)\) to move outside the unit circle in the \(Z\) plane, causing unstable waveforms. Consequently, auxiliary parameters called partial correlation (PARCOR) coefficients \(K_1\) (which are the negatives of reflection coefficients discussed by Atal) are calculated from the \(a_1\)'s and, as long as these PARCOR coefficients have magnitude less than unity, system stability is assured."
Figure 2-2. Simplified APCQ System
In the actual terminal, instead of quantizing the error signal as Figure 2-2 depicts, the analyzer is reformulated and the quantizer placed inside the filtering loop as shown in Figure 2-3. Without this quantizer, this configuration has the transfer function \((1-P_1)(1-P_2)\) as discussed earlier. With this new formulation, however, we eliminate the effects of small errors in accumulation caused by the error quantizer. This occurs because the only input to the predictor is the quantized error signal and, consequently, the predictor at the analyzer generates the same predicted sequence as the receiver does in the absence of transmission errors. Since it compares this predicted signal with the original input, the analyzer can correct for distortions caused by the error signal quantizer, altering the error signal sequence. This would not be possible without the feedback arrangement.

There are numerous methods for quantizing the error signal, and Jayant provides a good discussion of adaptive techniques which allow at least four levels or two bits per error sample (12). A major portion of this study effort was devoted to investigating various quantizers and these will be discussed in Section 2.2.

After computing the pitch gain \(\alpha\), the period \(M\), and the PARCOR coefficient \(K_i\), the analyzer digitally filters the speech and quantizes the error signal. Then the \(K_i\)'s, \(\alpha\), \(M\), the quantized error sequence \(e_n\), and a normalization factor NORM, needed to perform the operations in fixed-point arithmetic, are quantized and sent to the receiver where the predictor coefficients are regenerated in an iterative fashion by computing

\[
a_j^{(n)} = a_j^{(n-1)} - a_{-j}^{(n-1)} \cdot K_n, \quad j = 1, 2, 3, \ldots, i - 1,
\]

\[
a_j^{(n)} = K_i, \quad i = 1, 2, 3, \ldots N
\]

where \(a_j^{(1)}\) represents \(a_j\) on the 1st iteration.
Figure 2-3. Reconfiguration of Analyzer to Minimize Effects of Quantizer
The synthesizer then creates an output time waveform that is both intelligible and pleasing to listen to. Moreover, this output is reasonably insensitive to errors in pitch extraction because the P2 predictor on the reduced waveform and the quantization of the error signal can partially compensate for wrong pitch values used in the first predictor P1. In fact, if the pitch period M incorrectly doubles, as often happens in practice, then predictions made by the first predictor P1 are made from two periods before instead of the previous period. This is not a serious error. If other incorrect values for M are chosen then different values of \( \alpha \) and PARCOR coefficients are also computed to compensate, in part, for this error. Finally, the error signal, even though coarsely quantized, carries considerable information about the true pitch, should this pitch be incorrectly measured. Thus the APCQ speech process can function without severe degradation in noisy acoustic environments and on many different speakers where accurate pitch extraction is difficult.

The APCQ process can be modified to obtain other forms of speech coders currently being studied and discussed in the literature. These modifications can yield Adaptive Pulse Code Modulation Coders (APCM), Adaptive Differential PCM Coders (ADPCM), and Adaptive Predictive Coders (APC). These configurations for a 16 Kb/s voice coder are shown in Figure 2-4. The major differences between the different forms of coders are in the complexity of the predictor and in the form of the error signal quantizer. For example, in Adaptive PCM (APCM), there is no prediction, only the error signal quantizer adapts to the input speech. In Adaptive Differential PCM (ADPCM) the predictor uses a weighted value of the previous sample to form its estimate. The weight never changes and its value is typically near 0.9.
Figure 2-4. Relationships Among 16 Kb/s Speech Coder Configurations
There is no pitch prediction. Again, an adaptive quantizer is incorporated which adapts to the input. As already discussed, the adaptive predictive coder with adaptive quantization (APCQ) has two fairly sophisticated predictions: the coefficient predictor with typically 4 weights and a pitch predictor with pitch gain $\alpha$. The pitch loop can be easily eliminated without reconfiguring the coder by setting the pitch gain $\alpha$ to zero. The Adaptive Predictive Coder (APC) is identical to the APCQ coder except for the form of error signal quantizer employed. Its quantizer has only two levels whose amplitude changes once per frame interval. Its data rate is typically 6400 bps. The APCQ coder employs an error signal quantizer having 3 or more levels whose amplitudes change either once/frame of after each sample. The data rate is correspondingly higher, typically 9600 to 16,000 bps.
2.2 Error Signal Quantizers

The design of the error signal quantizer significantly affects the voice quality of the synthesized speech. Under this contract we have investigated several types of quantizers but considerably more work needs to be performed in this area to find a simple quantizer which performs well with channel errors and which will permit us to reduce the complexity of the adaptive predictor.

The significant feature of all the quantizers investigated is the adaptively changing size, Q. For example, Figure 2-5 illustrates an 8 level quantizer. This quantizer takes an input signal and converts it to one of eight discrete levels. The number of this level (000 to 111 in binary notation) is sent to the receiver which converts it back to the desired amplitude value. Since the curve describing the input/output characteristics of the quantizer looks like a staircase of step high Q (for uniform quantization), Q is commonly referred to as the step size. Until recently, these quantizers were memoryless, and once calculated, Q never changed even if the signal statistics changed. Unfortunately, if the input to the quantizer becomes large, the quantizer tends to saturate and not describe the input well. In a similar fashion, a very low level input may force the quantizer to always output its minimum step size and not track the small variations in the input. What is needed is a quantizer having memory which will increase the step size for large signals or signals with large variance and decrease it when the input signals become small or the variance becomes small.

All the error signal quantizers studied in this contract changed their step sizes at regular intervals. By changing the step
sizes in this manner, the quantizer adapts to the changing variance of its input. Two distinct classes of these quantizers exist; those that change their step size based on the transmitted value of the error signal and those that change their step size based on the variance of the unquantized error signal. P. Noll\textsuperscript{9,10} has called the first form "backward" quantizers since they look backward over previously quantized error samples to adjust their step size. He has labelled the second form "forward" quantizers because they look forward over the unquantized error sample to obtain their step size. The backward quantizers need not send the quantizer step size to the receiver because the receiver can regenerate this value by looking at the transmitted sequence representing the quantized error waveform. This is not true of the forward quantizers. Here, since the step size is based on the value of the unquantized error signal, the receiver cannot regenerate it from the transmitted sequence. Thus, forward quantizers transmit the value of the quantizer to the receiver. At a constant data rate, coders having forward quantizers have slightly lower audio input bandwidth than do those having backward quantizers.

Under this contract, we determined that both forward and backward quantizers can be made to work equally well in the presence of channel errors. This contradicts the work of Noll who believed that forward quantizers which transmitted their step size would work better. At the time of his writing, however, Noll was unaware of modifications to the backward quantizers which would improve their tolerance to channel errors.

We will first discuss these backward quantizers and the modifications needed to improve performance with channel errors. Then we will describe the forward quantizers and the necessity to send the step size.
2.2.1 The Jayant Quantizer

The Jayant quantizer adapts its step size instantaneously, expanding or contracting its value each time it quantizes a new sample. Figure 2-5, illustrating an 8 level (3 bit/sample) quantizer, shows the operation of this quantizer. If the incoming sample is in one of the inner quantizing levels, the quantizer step size is decreased for the next sample. If the incoming sample is in one of the outer levels, the step size is increased instead. These changes are based on the quantized output and thus, the Jayant quantizer is classified as backward for there is no need to send the quantizer step size as the receiver can regenerate it. The constantly changing step size, which is attempting to track the input signal variance is given by

\[ q_k = M(I_k) q_{k-1} \]

at time \( k \) where the \( M(I_k) \) represent the multiplication factors for the step size on the \( k \)th sampling interval and \( I_k \) is the \( k \)th transmission symbol. The adaptive quantizer formulation need not be restricted to quantizers having only 8 levels. Any quantizer having more than 2 levels can be made adaptive with the Jayant algorithm. In fact, a 3 level quantizer is possible. This quantizer, which uses (1.6 bits/sample) quantizer, expands the step size whenever its two outer values are used and decreases the step whenever the inner value is used. The three level quantizer is desirable whenever the data rates must not exceed 9600 bps, whereas the 8 level quantizer requires at least 16000 bps at 2500 Hz bandwidth.
In general, for a given transmission rate, there is an inverse relationship between the number of quantizer levels and the audio input bandwidth. At 16,000 bps an 8 level quantizer (3 bits/sample) limits the audio bandwidth to about 2500 Hz, while a 5 level quantizer (2.33 bits/sample) would permit about 3200 Hz bandwidth and a 4 level quantizer would allow more than 3300 Hz. Thus, using a 5 or 6 level quantizer instead of an 8 level quantizer, the audio bandwidth of the coder will increase from 2500 Hz but the quantizing noise will also increase because fewer quantizing levels are employed.

The Jayant algorithm performs well without channel errors. Unfortunately, channel errors significantly impair performance. To see this consider

$$q_k = M(I_k) q_{k-1} = \prod_{i=1}^{k} M(I_i) q_0$$

where $I_i$ is the $i$th transmitted character and $M(I_i)$ is the multiplier corresponding to $I_i$.

Assume now that one transmission error is made at time $\ell$ causing

$$M(I_\ell) \text{ to become } M(I_\ell')$$

and

$$q_\ell \text{ to become } q_\ell' = M(I_\ell') q_{\ell-1}$$
afterwards the new quantizer levels for time greater than \( t \) becomes

\[
q_k' = \sum_{L=1}^{k} M(i_1) M(i_2') q_o \cdot \left[ \frac{M(i_k')}{M(i_k)} \right]
\]

or simplifying we obtain

\[
q_k' = q_k \frac{M(i_k')}{M(i_k)}
\]

Consequently, the effects of even a single channel error never die out. In fact, an error causes a level shift in \( q_k \) by the factor \( M(i_k')/M(i_k) \). Simulations and real-time implementation show that errors cause noticeable fades and increases in output level. Only when the quantizer saturates at its maximum level or cuts off at its minimum level does the quantizer again track properly. Thus, the Jayant algorithm must be altered if it is to be useable over real communication channels.
2.2.2 The Modified Jayant Algorithm

Wilkinson and Goodman\textsuperscript{11} have recently shown how a simple change to the Jayant algorithm can make this algorithm robust to channel errors. Their modification was

\[ q_k = M(I_k)q_{k-1} = \prod_{i=1}^{k} M(I_i)q_0 \]

where \( I_i \) is the \( i \)th transmitted symbol and \( M(I_i) \) is the multiplier corresponding to \( I_i \). Again if one channel error curves at time \( \ell \)

\[ M(I_\ell) + M(I_\ell') \]

and

\[ q_k' = \prod_{i=1}^{k} M(I_i)q_{k-\ell} = \prod_{i=1}^{\ell} M(I_i)q_0 \]

or simplifying

\[ q_k' = q_k \cdot \left[ \frac{M(I_\ell')}{M(I_\ell)} \right]^{k-\ell} \]

Assuming \( \alpha < 1 \), then for times \( k \) far removed from \( \ell \) (\( k \gg \ell \)),

\[ q_k' \approx q_k \]

The speed of this convergence depends on how near \( \alpha \) is to \( 1 \). If \( \ell \) is not too close to unity, convergence is rapid.

Consequently, the effects of a single channel error decay with time for the modified Jayant algorithm and this algorithm is much more suitable for use over communications channels.
2.2.3 The Forney Quantizer

The Forney quantizer\(^5\) was another attempt at creating a backward quantizer which is insensitive to channel errors.

As shown in Figure 2-6,

\[ q_k = 2^{\varepsilon_k} \]

where

\[ \varepsilon_k = G_k + C_k \]

and

\[ G_k = \alpha G_{k-1} + M_1(I_k) \]
\[ C_k = \beta C_{k-1} + M_2(I_k) \]

where specific\( M_1(I_k), M_2(I_k) \) \( \alpha \) & \( \beta \) are given in Figure 2-6.

Consequently, this quantizer also increases its step size when large inputs occur and contrasts the step size during low level inputs.

The quantizer step size is non-uniform and reacts rapidly to occasional large spikes caused by pitch pulses.

Noting that

\[ G_k = \sum_{i=1}^{k} \alpha^{k-i} M_1(I_i) \]

and

\[ C_k = \sum_{i=1}^{k} \beta^{k-i} M_2(I_i) \]

we have

\[ q_k = 2 \left( \sum_{i=1}^{k} \alpha^{k-i} M_1(I_i) + \sum_{i=1}^{k} \beta^{k-i} M_2(I_i) \right) \]
$Q_i = 2E_i$
$E_i = G_i + C_i$

AND

$G_i = .99G_{i-1} + \begin{cases} 
9/32 & \text{if } |\text{OUTPUT}_{i-1}| = 8L_{i-1} \\
5/64 & \text{if } |\text{OUTPUT}_{i-1}| = 5L_{i-1} \\
1/128 & \text{if } |\text{OUTPUT}_{i-1}| = 2L_{i-1} \\
-1/16 & \text{if } |\text{OUTPUT}_{i-1}| = 0.5L_{i-1} \\
\end{cases}$

$C_i = .75C_{i-1} + \begin{cases} 
.875 & \text{if } |\text{OUTPUT}_{i-1}| = 8L_{i-1} \\
0 & \text{if } |\text{OUTPUT}_{i-1}| = 5L_{i-1} \\
0 & \text{if } |\text{OUTPUT}_{i-1}| = 2L_{i-1} \\
0 & \text{if } |\text{OUTPUT}_{i-1}| = 0.5L_{i-1} \\
\end{cases}$

CHARACTERISTIC: SLOWLY ADAPTIVE EXCEPT DURING OCCURRENCE OF LARGE PULSES. NO NEED TO TRANSMIT $Q_i$

Figure 2-6. Forney Adaptive Quantizer
at time \( t \) assume a channel error causes

\[
M_1(I_t) + M_1(I_t') \\
M_2(I_t) + M_2(I_t')
\]

then

\[
q_k' = 2 \left( \Sigma_{i=1}^{k} a^{i-1} M_1(I^i_t) + a^{k-\ell} M_1(I^\ell_t') + \Sigma_{i=1}^{\ell} b^{i-1} M_2(I^i_t) + b^{\ell-k} M_2(I^\ell_t') \right)
\]

by adding and subtracting

\[
a^{k-\ell} M_1(I_t) \text{ and } b^{k-\ell} M_2(I_t)
\]

to the exponent

we have

\[
q_k' = q_k \cdot 2a^{k-\ell} (M_1(I_t') - M_1(I_t)) \cdot 2b^{k-\ell} (M_2(I_t') - M_2(I_t))
\]

as \( k \gg \ell \) \( a^{k-\ell} \to 0 \)

for \( a < 1 \)

and \( q_k' \to q_k \)

Thus the Forney quantizer also is insensitive to channel errors

because after a channel error the step size approaches the desired

step size with time. Moreover, this algorithm responds more rapidly

to sudden changes in the input than does the modified Jayant

algorithm.
2.2.4 The Fixed/Frame Quantizer

Fixed/frame quantizers as originally used in APC algorithms\textsuperscript{1,2} calculate a step size for an entire frame of data. Since the receiver cannot regenerate this data from the quantized error signal sequence, the step size must be sent to the receiver once each frame. This is a forward type of quantizer as described by Noll.\textsuperscript{9,10}

The step size $Q$ is a function of the RMS energy in the input signal. The energy $U$ is calculated during estimation of the predictor coefficients and

$$Q = \frac{1}{2}(U/T)^{1/2}$$

where $T$ = the frame length and the factor $1/2$ was determined experimentally.

The fixed/frame quantizer is insensitive to errors because a single channel error effects only one value of the reconstructed error signal. Of course, if an error occurs in transmitting $Q$ then an entire frame of data is shifted in level. Fortunately, the following frames of data are unaffected.
2.3 Fortran Fixed-Point Simulations

The various APCQ configurations and error signal quantizers were investigated through FORTRAN fixed-point simulations intended to implement the algorithms with the same numerical accuracy as the real-time software. Writing these codes in FORTRAN permitted us to investigate many designs rapidly without the long debugging and check out times associated with writing real-time software.

The block diagram of the APCQ coder is shown in Figure 2-7 with a more detailed indication of the processing at the analyzer (transmitter) shown in Figure 2-3. Figure 2-8 indicates the general flow of data in the analyzer with Figure 2-9 indicating the data flow of the synthesizer (receiver). These flowcharts indicate that the speech stored on digital tape is read in a frame at a time, processed by the fixed-point algorithms, and reconstructed and placed on an output tape for listening. The I/O is handled by the digital tape handlers written by GTE Sylvania prior to this contract. Thus the I/O for the simulation was straightforward, and effect was placed on the scaling and normalization needed to perform the processing operations in fixed-point arithmetic. Appendix A provides a complete listing of the fixed-point simulation software with flow charts indicating the scaling and normalization operations in each arithmetic function.

2.4 Simulation Results

2.4.1 Coder Performance without Channel Errors

All of the speech coders of this contract attempted to preserve the time waveform shape. As a result, a reasonable performance measure for these systems is signal-to-noise ratio (S/N) as defined in Figure 2-10. Informal listening tests of these coders indicate that listeners generally prefer those systems having higher S/N ratios over those having lower ratios. The computer simulations developed S/N
Figure 2-8. Analyzer Flowchart of Fixed-Point Simulation for APCQ Coder
Figure 2-9. Synthesizer Flow Chart of the Fixed-Point Simulation for APCQ Coder
**Signal-to-Noise Measurement**

Signal = $S_N$

Noise = $S_N - R_N$

S/N Ratio (in dB) = $10 \times \log_{10} \left( \frac{\sum S_N^2}{\sum (S_N - R_N)^2} \right)$

Figure 2-10. Signal-to-Noise Measurement
ratios for each non-overlapping 25 msec speech segment. These ratios were then averaged to obtain a cumulative S/N ratio for an entire sentence. It is these cumulative S/N ratios that are used in our results presented in the graphs of Figures 2-11 through 2-12.

Figure 2-11 shows the performance of APCQ systems using an 8 level Jayant quantizer operating at 16 kbps. This data was obtained by averaging the cumulative S/N ratio obtained for each sentence over a total of six input test sentences containing both male and female voices. Each test sentence contained about 100 analysis frames and thus, each point on Figure 2-11 represents the average of 600 frames. Referring to Figure 2-11, the performance increases with increasing predictor order and with the inclusion of a pitch loop. The zeroth order predictor without pitch is equivalent to APCM. This system performs roughly 2-3dB worse than the CVSD system we have chosen as our benchmark. Other simulations performed as part of this study indicate that ADPCM has a S/N equivalent to APCQ-1 (APCQ with 1 predictive coefficient) without pitch. As Figure 2-11 indicates, APCQ-1 (without pitch) is roughly equivalent to CVSD. The addition of a pitch loop adds between 1.5 and 2.0 dB to the S/N performance.

2.4.2 Coder Performance with Channel Errors

Figure 2-12 shows the performance of these coders in the presence of channel errors for the Jayant, Modified Jayant, Forney and fixed frame quantizers. Controlled random bit errors were applied to all transmitted data equally. The coder used to evaluate these quantizers was a first order adaptive predictor (APCQ-1) without a pitch loop. In preparing the data of this Figure, only one test sentence was used due to large amount of processing involved. Using selected points taken from other test sentences showed similar trends in performance.
Figure 2-11: S/N of CVSD and Adaptive Predictive Coders with Jayant Quantizer [APCQ] vs. Predictor Complexity
Figure 2-12. S/N of Different Quantizers versus Bit Error Rate (1st Order Adaptive Predictor W/O Pitch Loop)
As can be seen from Figure 2-12, the Jayant quantizer, CVSD and the Forney algorithms are within 1dB of each other in the absence of channel errors. Under this condition, the fixed/frame quantizer is 3dB below the performance of these other quantizers and the modified Jayant quantizer is 1dB below the fixed/frame quantizer. The Jayant algorithm, however, is extremely sensitive to channel errors. Its performance falls off rapidly when errors occur. This quantizer produces noticeable distortion at error rates as low as $10^{-4}$ and is unacceptable at BER's of $10^{-3}$ and above. The Forney quantizer is more robust, maintaining good performance at $10^{-3}$, fair performance at $10^{-2}$ and unacceptable performance at $10^{-1}$. While starting at a lower performance level than the other quantizers, the fixed frame quantizer does not degrade rapidly with increasing channel errors. In fact, it provides better performance than either the Jayant or Forney quantizers at error rates higher than $10^{-2}$. The modified Jayant quantizer using a decay factor of 0.96 had a S/N that was always below the fixed frame quantizer, but at error rates above $10^{-2}$, was higher than the FORNEY quantizer. Note that CVSD, the simplest system, exhibited the best performance against channel errors until error rates exceed $10^{-1}$.

Based on the simulation data of Figure 2-12, the fixed frame quantizer was chosen to evaluate the performance of the APCQ system with more complex predictors. Figure 2-13 shows how APCQ systems using a fixed frame quantizer perform against channel errors over the same sentence material as Figure 2-12. In this figure, the channel errors are imposed uniformly on the transmitted data including the pitch M, pitch gain a, the PARCOR coefficients and the quantizer step size as well as on the quantized error signal sequence.
Figure 2-13. 16 KBPS Speech Coder Performance vs. Random Channel Bit Errors (BER)
At low error rates, a fourth order APCQ system with pitch outperforms CVSD by 3dB. As bit errors are imposed, the APCQ system loses its advantage over CVSD. At high error rates, the presence of the pitch loop becomes detrimental to APCQ performance. At error rates between $10^{-3}$ and $10^{-2}$, the APCQ system with fixed/frame quantizer and CVSD have similar S/N rates. Note that the simplest system (i.e., APCM) outperforms CVSD at error rates of $10^{-1}$. This is shown by the bottom curves of Figure 2-13 with predictor order equal to zero and no pitch loop present. Figure 2-14 provides an alternate method of graphing some of the data given in Figure 2-13. These curves compare the channel error performance of a fourth order adaptive predictor, with and without pitch, to CVSD. At low error rates, fourth order systems are superior to CVSD. At higher error rates, CVSD is either superior to APCQ with pitch or comparable to APCQ without pitch.

These results indicate that system performance is a critical function of channel error rate. As low errors occur, the complex APCQ systems outperform CVSD at 16 Kb/sec. For high channel error environments, CVSD is superior. If high quality is the objective in low error environments, then the APCQ system provides the designer with a superior voice digitization technique.

Based on these simulation results, GTE Sylvania and DCA agreed to develop a final system using a fixed frame quantizer. To increase the audio bandwidth from 2500 Hz to 3200 Hz, the 8 level quantizer was reduced to a 5 level quantizer. This lowered the S/N ratio of the APCQ system approximately 3dB from those presented in Figures 2-11 through 2-14.

2.4.3 Improved Error Signal Quantizers

The modified Jayant algorithm, however, did not become available until after we made this decision to implement with the
Figure 2-14. S/N of CVSD and 4th Order Adaptive Predictive Coder (Fixed Quantizer) vs. Bit Error Rate @ 16 KBPS
fixed/frame quantizer. Moreover as Figure 2-12 illustrates, its S/N is always lower than the other quantizers. Nevertheless, informal listening tests conducted after the decision to implement with the fixed/frame quantizer indicated little if any degradation between the Jayant and modified Jayant quantizers without channel errors.

Even more important was the discovery that the distortions introduced by the modified Jayant quantizer were less objectionable than those introduced by the fixed/frame quantizer. Consequently, during the last weeks of the contract, additional simulation runs were performed to observe the behavior of the modified Jayant algorithm further.

Figure 2-15&16 illustrates the performance of the APCQ coder with channel errors inserted into the transmitted data using the modified Jayant quantizer. The S/N ratios behave about the same way as the fixed frame quantizer in the presence of channel errors. Thus for error exponents greater than $10^{-3}$, the systems having no pitch predictor and a low order coefficient predictor performed better than the systems with a pitch loop and high order predictions. The simple systems are better since the channel errors occur on the pitch parameters and predictor coefficients and these must significantly affect the S/N ratio. On good channels ($\text{BER} \leq 10^{-4}$) the S/N ratio improves as the predictor becomes more sophisticated.

The S/N ratio of the APCQ coder with modified quantizer is always lower than that for the fixed/frame quantizer. The perceived voice quality, however, is generally higher because the types of distortion are different.

Thus the S/N curves while indicative of the expected performance of a technique are not always directly related to voice quality.
Figure 2-15. S/N of Adaptive Predictive Coders Using 8-Level Modified Jayant Quantizer
Figure 2-16. S/N of CVSD and 4th Order Adaptive Predictive Coder (Modified Jayant Quantizer) versus Bit Error Rate at 16 Kb/s
The modified Jayant quantizer has high voice quality but fairly low S/N ratios.

The modified Jayant quantizer because it does not degrade as the predictor becomes less complex offers the potential of simplifying the APCQ system dramatically.
SECTION 3
Real-Time Software

3.1 Description of APCQ Program
3.1.1 General Discussion

The real-time APCQ program written in assembly language enables the EDM to collect, analyse, transmit, receive and synthesize speech data in full duplex mode. The sequence of operations involves initialization of the system parameters, set-up of the interrupt processing, acquisition of synchronization, processing of speech via the APCQ algorithm, transmission and reception of parameters and reconstruction of speech.

The APCQ algorithm implemented is a fourth order adaptive predictive coder with a pitch loop and a five-level fixed frame quantizer. Each speech sample is predicted as a weighted linear combination of four past speech samples. The error signal, computed from the difference of the actual and the estimated speech sample, is then quantized into five levels by the quantizer shown in Figure 3-1 with level estimated from the RMS power of the error signal which remains constant throughout a whole frame of speech data.

The key to correctly sequencing the operations is the use of two interrupts in the EDM: the speech side interrupt and the line side interrupt. The speech side interrupt is set to interrupt the computer at regular intervals and transfer control to software (starting at Location 0), which inputs one speech sample into the computer and outputs one speech sample from the computer. Careful count is kept of the number of samples read in; and when a frame of data has been input, an indicator (transmit flag) is set showing the analysis can begin.

The line side interrupt also regularly interrupts the computer and transfers control to software (starting at Location 1), which outputs modem clock (five-volt levels from a flip-flop) and one bit of transmission data. At every eighth interrupt it inputs eight bits of
Figure 3-1. Characteristics of a 5-Level Quantizer
received data which have been accumulating in an eight-bit shift register. When it receives a frame of input data, it sets an indicator (receiver ready flag) showing that synthesis can begin.

Since the program operates in less than real-time, it executes instructions in a wait loop while it idles. It leaves the wait loop to analyze speech whenever it sees that the transmit ready flag has been set by the speech interrupt. After analysis it returns to the wait loop where it will leave again for the synthesis when the line side interrupt raises the receive ready flag. In addition, it also interrupts the sync flag and leaves for the sync search routine whenever the flag is set. Naturally, the speech and line interrupts are functioning even during execution of the wait loop. Speech is input and output at each sampling instant, and data is sent and received at 16,000 bps.

3.1.2 Initialization and Wait Loop

When the program is started, it immediately goes to an initialization mode where all parameters and counters of analyser and synthesizer filters are cleared. It then goes into the wait loop shown in flow chart Figure 3-2 and waits until one of three flags, namely: transmitter ready flag, receiver ready flag and sync search ready flag is set where it performs the corresponding processing. At the end of the operation, the program returns to the wait loop to use up the idle time left over. Reinitialization can always be achieved by setting one of the front panel switches.

3.1.3 Synchronization

After received 80 bauds (1.5 second) of data, the sync search ready flag is set where the program goes to sync search loop shown in Figure 3-3. 
Figure 3-2. Initialization and Wait Loop
Figure 3-3. Synchronization Search
To understand the synchronization process, it is necessary to describe the format of the transmission data. Figure 3-4 shows that the transmission baud consists of 312 bits arranged in the following manner: 280 bits for the predictor error signal, one synchronization bit which is always set to a one, three bits for fourth PARCOR coefficient, three bits for the third PARCOR coefficient, four bits for the second PARCOR coefficient, four bits for the first PARCOR coefficient, three bits for the pitch gain ALPHA, six bits for the pitch, four bits for the RMS power of error signal, three bits for the normalizer and a second one bit for synchronization which is also always a one. The receiver develops a histogram of the received data, counting the times a one is received in each of the 312 bit positions. After 80 frames (approximately 1.5 seconds) the receiver examines the histogram for positions of two values near 80 (the values may be slightly less than 80 because of transmission errors) which are 31 bit positions apart. From the position of the sync points, the software then knows where the last bit in the frame is located, and it shifts the received sequence so that the first bit of the baud lines up with the first storage location. Once synchronization has been achieved, the locations of the synchronization bits are saved and compared with that of subsequent sync searches to determine the necessity of shifts in the following received sequence.

3.1.4 Line-side Interrupt

Figure 3-5 describes the program flow whenever the clock interrupts the computer for data transmission or reception by trapping to Location 1. First the flip-flop which drives the modem clock is set
Figure 3.4. Format of Transmission Data

- CODED ERROR SIGNAL
- 280 BITS
- SYNCHRONIZATION
- PARCORS
- ALPHA
- PITCH
- Q LEVEL
- NORMALIZER
- SYNCHRONIZATION
Figure 3-5. Line Side Interrupt
or reset depending on whether the interrupt was even-numbered or odd-numbered. Whenever it is set, one bit of data is also sent and the receiver buffer is interrogated to see if eight bits of data have been received. If received data is ready, the software inputs the data, compiles the histogram, and, whenever the baud is complete, sets a receiver ready flag. Every 80 bauds it sets a flag indicating that a synchronization search could begin if desired.

3.1.5 Speech Side Interrupt

The speech side interrupt is straightforward, as shown in Figure 3-6. When the computer interrupts and traps to Location 0, it reads in one speech sample, reads out one sample, and when the buffer is full, sets the transmit ready flag. The speech data is double-buffered to prevent loss of data during analysis.

3.1.6 Speech Analysis and Synthesis

Flow charts of Figures 3-7, 3-8, 3-9 and 3-10 describe the operations of the analysis of the APCQ coder. Upon entry to the analyser the input data is first normalized. Then pitch is computed via the AMDF function as shown in Figure 3-5. Next the pitch gain ALPHA and the reduced waveform are calculated as illustrated in Figure 3-7.

In Figure 3-8 the autocorrelation coefficients of the reduced waveform are determined that are later used to compute PARCOR coefficients via the matrix inversion routine shown in Figure 3-9 from which actual predictor coefficients (a's) are derived as in Figure 3-14. Finally, the RMS power of the error signal is calculated and used to quantize error signal and coded into seven bit words as shown in Figure 3-10.

Figure 3-12 and 3-13 describes the operations of the synthesizer where all transmitted parameters are decoded and speech is reconstructed by inverse filtering.
Figure 3-6. Interrupt Loop Speech Side
Figure 3-7. Analyzer for APCQ Coder
\[
R(J) = \frac{1}{8} \sum_{K=0}^{QBL/8-1} QV(QBL-K) \cdot QV(QBL-K-J) + \frac{2}{8} \sum_{K=QBL/8}^{2 \cdot (QBL/8)-1} QV(QBL-K) \cdot QV(QBL-K-J) + \ldots + \frac{1}{8} \sum_{K=QBL/8}^{QBL-1} QV(QBL-K) \cdot QV(QBL-K-J)
\]

\[
= \frac{1}{8} \sum_{K=QBL/8}^{7/8 \cdot QBL} QV(QBL-K) \cdot QV(QBL-K-J)
\]

\[
J = J - 1
\]

\[
J = 0
\]

\[
\text{GO TO PARCOR COEFS}
\]
Figure 3-9. Calculation of PARCOR Coefficients ($K_1$) and RMS Power
Figure 3-10. Error Signal Calculation and Quantization
SCRAMBLER

XMITTER

SCRAMBLER

X1 $ 

DETERMINE WHICH BUFFER IS CURRENTLY XMITTING DATA

CODE ERROR SIGNAL & OTHER MODEM PARAMETERS INTO XMIT ARRAY OF 20 WORDS

STORE ARRAY IN BUFFER NOT CURRENTLY XMITTING

GO TO WAIT

4297-75E

Figure 3-11. Transmission Data Construction
RECEIVER UNSCRAMBLE OTHER MODEM PARAMETERS

Determine which buffer is currently receiving data

Unscramble error signal and other parameters from RCVR array

UNSCRAMBLE ERROR SIGNAL AND OTHER PARAMETERS FROM RCVR ARRAY

\[ RMIT(j) = \sum_{i=0}^{6-1} 2^{-i} \cdot L.B.(RREC(i+7j)) \]

WHERE \( j = 0, 1, \ldots, 39 \)

& L.B. = LAST BIT OF

Skip 1 word (SYNC PT) in RCVR array

Unscramble PARCOR COEFS

\[ QPRR(3) = \sum_{n=0}^{2} 2^{-i} \cdot L.B.(RREC(i+282)) \]
\[ QPRR(2) = \sum_{n=0}^{2} 2^{-i} \cdot L.B.(RREC(i+285)) \]
\[ QPRR(1) = \sum_{n=0}^{2} 2^{-i} \cdot L.B.(RREC(i+288)) \]
\[ QPRR(0) = \sum_{n=0}^{2} 2^{-i} \cdot L.B.(RREC(i+292)) \]

Figure 3-12. Data Reception, Decoding and Dequantization
Figure 3-13. APCQ Synthesizer
SUBROUTINE QCAT & QACR

\[ a_1 = K_1 / 4 \]

\[ i = 2 \]

\[ b_i = a_i - K_1 \times a_{i-1} / 2^{15} \]

FOR \( j = 1, 2 \ldots i - 1 \)

\[ a_i = b_i \]

FOR \( j = 1, 2 \ldots i - 1 \)

\[ a_i = K_i / 4 \]

INCREMENT \( i \) BY 1

NO

\( i = 5? \)

YES

RETURN

Figure 3-14. Converting PARCOR Coefficients to A's
SUBROUTINE OAMPF

PITCH CALCULATION

SET INITIAL VALUES
QAMP = 32767.
QPI T = 15.

\[ I = 60 \]

Q150 $\to$

OBTAIN PARTIAL SUM 1
\[ \text{PSUM} = \sum_{j=0}^{15} \text{QINI}(j) - \text{QINI}(j+15) \]

OBTAIN PARTIAL SUM 2
\[ \text{PSUM+1} = \sum_{j=18}^{36} \text{QINI}(j) - \text{QINI}(j+15) \]

OBTAIN PARTIAL SUM 3
\[ \text{PSUM+2} = \sum_{j=39}^{54} \text{QINI}(j) - \text{QINI}(j+15) \]

OBTAIN PARTIAL SUM 4
\[ \text{PSUM+3} = \sum_{j=57}^{72} \text{QINI}(j) - \text{QINI}(j+15) \]

AMDF = \[ \frac{1}{4} \text{PSUM} + \frac{1}{4}(\text{PSUM+1}) + \frac{1}{4}(\text{PSUM+2}) + \frac{1}{4}(\text{PSUM+3}) \]

\[ ? \] AMDF > QAMP

\[ \text{YES} \]

SET
QAMP = AMDF
QPI T = I+15

\[ I \geq 0 \]

\[ \text{GO TO} \]
Q150

\[ \text{NO} \]

\[ ? \] QAMP = AMDF

\[ \text{YES} \]

\[ \text{RETURN} \]

\[ \text{NO} \]

\[ I = I - 1 \]

\[ \text{GO TO} \]
Q150

Figure 3-15. AMDF Pitch Calculation
**Figure 3-16. Subroutine QSRT**

SUBROUTINE QSRT

SET LOWER & UPPER LIMITS
QLL = X
QUL = X + 1/4

? QLL ≤ QUL

NO

\( \hat{x} = \frac{QLL + QUL}{2} \)

YES

? \( |QUL^2 - x| < |QLL^2 - x| \)

NO

\( \sqrt{x} = QUL \)

YES

\( \sqrt{x} = QLL \)

NO

\( (x - \hat{x}^2) > 0 \)

SET QUL = \( \hat{x} \)

YES

\( (x - \hat{x}^2) = 0 \)

SET QLL = \( \hat{x} \)

NO

\( \sqrt{x} = \hat{x} \)

RETURN
3.2 Operation Procedures for the APCQ Program

3.2.1 Connecting the System

a. Connect the EDM's to each other either in back-to-back mode shown in Figure 3-17 or through a 16,000 bps modem as shown in Figure 3-18.

1. Use the terminal marked "Data" on the rear of the CPU to connect to the modem or the other EDM.

2. Place the toggle switch to appropriate RS-232 or MIL-188 interface position as required.

3. Strap the modem to operate from the external transmit clock or the leading edges of the data.

b. Connect the handsets or audio tape recorders to the proper input and output jacks at rear of CPU.

3.2.2 Reading in the EDM Software

a. Place all switch register toggle switches in the up position.

b. Power on the PSP and card reader.

c. Depress the "PM CLEAR" switch to clear program memory.

d. Set "NORMAL/TEST PMIR" switch to "NORMAL".

e. Set "RUN/STOP" switch to "STOP".

f. Load the binary deck and depress the "MOTOR" and then "START" switches on the card reader.

g. Cards (load PM, load DM, PMMA checksum, DM checksum) are read in. Card reader halts before it gets to PMIR checksum cards.

h. Depress "STOP" switch on card reader.
Figure 3-17. System Test Configuration
Figure 3-18. Modem Interface Signals and Interconnection
1. Set "NORMAL/TEST PMIR" switch to "TEST PMIR."

Depress "MANUAL INST" switch

j. Restart the card reader by depressing "MOTOR" and
"START" switches

k. After the whole deck is read, set "NORMAL/TEST PMIR"
switch back to "NORMAL"

l. Set "INTERRUPT PROGRAM/DISABLE" switch to "INTERRUPT
PROGRAM"

m. Set "RUN/STOP" switch to "RUN" to start executing
the program.

3.2.3 Initializing the EDM

After the program has been running, reinitialization
can be achieved by simultaneously holding switch register
toggle switch 15 in the down position for each EDM and then
returning it to the up position.
3.3 Description of Continuous Variable Slope Delta Modulation Program

3.3.1 Basic Principles of Operation

CVSD represents one of the simplest voice digitizers known, since there are relatively few mathematical operations per input sample. Figure 3-19 depicts a digital implementation of a CVSD transmitter and receiver operating on the DCA EDM's. The input speech is first converted to digital by a 12 bit A/D converter after lowpass filtering through an elliptic filter having a 3 dB frequency of 3200 Hz. The CVSD algorithm subtracts an estimate of the incoming sample from the input and quantizes the difference or error sample to +1 if the error is positive and -1 if negative. The quantized value is encoded for transmission by the mapping 1 into 1 and -1 into 0. The algorithm then uses a two state process to construct the estimate for the next incoming sample. The first stage computes a quantizer gain factor by interacting with time constant $T_1$ and uses this factor to adjust the level of the quantized error signal. The second stage filters this new error signal through an integrator having time constant $T_2$ to produce the new speech estimate. The CVSD receiver then reconstructs the ±1 bit stream of the transmitter's error quantizer. This data excites the same quantizer gain and error signal integrator (which formed the speech estimate at the transmitter) and generates the synthesized speech after D/A conversion and low pass filtering.

With varying gain in the error signal, the system adapts to the slope of incoming speech waveform. Severe slope overload occurs with insufficient quantizer gain perceived as muffled speech with drop of volume. On the other hand, excessive gain will cause overshoots which introduce granular noise into the synthesized speech. In this study, three slope adaptation strategies are investigated:
Figure 3-19. Transmitter and Receiver of CVSD-B
3.3.2 Standard Three Bit Memory (CVSD-B)

Figure 3-19 shows the block diagram of the CVSD-B transmitter and receiver. By employing a three bit shift register, memories of the present and past two error samples are retained and utilized to estimate the incoming speech. Whenever the three bits of the shift register are identical (111 or 000), a waveform with steep slope is usually present. Then a spike of amplitude A is introduced into a single pole smoothing filter with coefficient $a_1$. Whenever the three bits are different, the logic introduced a zero into the same filter shown as:

$$f(j) = a_1 f(j-1) + \begin{cases} A, & \text{if shift register } = 000 \text{ or } 111 \\ 0, & \text{otherwise} \end{cases} \quad (3-1)$$

Then the filter output for a constant input amplitude $A$ or 0 is given by

$$f(n) = \begin{cases} \frac{A}{1-a_1} (1-a_1^{n+1}), & \text{if shift register } = 000 \text{ or } 111 \\ a_1^n, & \text{otherwise} \end{cases} \quad (3-2)$$

When $f(j)$ is added to a small bias $\delta$, the result is a gain factor for multiplying the ±1 data coming from the error quantizer. This new waveform then generates the speech waveform by exciting a second single pole integration filter with coefficient $a_2$.

In order to determine the amplitude of $A$ that yields the best system performance, a parameter called compression ratio (CR) which is defined as

$$CR = \frac{f_{\text{max}} + \delta}{\delta} \quad \text{where } f_{\text{max}} = \frac{A}{1-a_1} \quad \ldots. \quad (3-3)$$

was investigated. This compression ratio is related to $A$ by

$$A = (1-a_1)^* \delta^* (CR-1) \quad (3-4)$$
With fixed $\delta$ and $a_1$, the amplitude of the spike $A$ is directly proportional to the compression ratio. With a higher value of compression ratio, the system tracks the incoming waveform with a smaller slope overload.

As shown in Eq. (3-4), high pulse height $A$ can also be obtained by raising the bias value $\delta$. Hence a high $\delta$ tends to yield a better estimate to a fast changing input waveform. But in the presence of a slow varying waveform, such as silence intervals between speech, the main excitation component to the integration filter is $\delta$ which yields an oscillatory reconstructed waveform with a value $\pm \delta$. Hence large $\delta$ tends to increase the granularity of the processed speech.

Besides the above parameters, the time constants $T_1$ and $T_2$ of the gain and prediction filters also contribute to the reconstruction of speech waveform. A large value for the time constant $T_1$, which is related to the coefficient $a_1$ by

$$T_1 = -T/\ln a_1$$  \hspace{1cm} (3-5)

Where $\frac{1}{T}$ is the transmission rate tends to have a smoothing effect on producing the quantizer gain which decreases the sensitivity to channel errors. A small value for time constant $T_2$ which is given by

$$T_2 = -T/\ln(a_2)$$  \hspace{1cm} (3-6)

produces a more energetic tracking of the input waveform.
3.3.3 Forney and Qureshi's 3 Bit Memory (CVSD-C)

Another strategy for updating the quantizer gain using exponentials is shown in Figure 3-20. This algorithm uses a 3-bit shift register with the last bit corresponding to the present error sample. The content of this shift register are used to generate pulses of different length which excite a smoothing filter. Table 3-1 provides the amplitudes of these pulses as a function of the shift register contents.

<table>
<thead>
<tr>
<th>Content of Shift Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>3/16</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>-5/128</td>
</tr>
<tr>
<td>011</td>
<td>5/64</td>
</tr>
<tr>
<td>100</td>
<td>5/64</td>
</tr>
<tr>
<td>101</td>
<td>-5/128</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>3/16</td>
</tr>
</tbody>
</table>

Table 3-1: Pulse Heights for FORNEY CVSD Quantizer

The output of this filter after addition of a small bias $\delta$, is used as the exponent on the expansion $2^x$. For implementation purposes, an approximation to the exponential expression is given by

$$2^{(IQK + QKF)} = 2^{(IQK)} (1 + QKF)$$  \hspace{1cm} (3-7)

where $IQK = \text{integer part of the filter output and bias}$

$QKF = \text{fractional part of the filter output and bias}$

The result of this exponentiation is then used to excite an integration filter. Because exponentials are used for both increasing and decreasing the quantizer, a faster gain can be
Figure 3-20. Block Diagram of CVSD-C
achieved within a relatively short time, resulting in a better dynamic range of the system and better tracking of rapidly increasing and decreasing waveforms.

3.3.4 Jayant's One Bit Memory (CVSD-D)

Figure 3-21 illustrates the Jayant slope adaptation logic.

In contrast to the two previous algorithms, the Jayant method uses only one bit of memory. Every time an error sample is generated, it is compared with the previous quantized value. If they are different, the present quantizer level is multiplied by a factor $Q < 1$. If they are the same, it implies a continuous rising or falling waveform. Then the present level is multiplied by a factor $P > 1$. They are used to excite the integration filter to obtain an estimate of the incoming speech waveform.

3.4 Discussion of Real-Time Code

The real time software of CVSD is straightforward when compared to APCQ program codes. The sequence of operations involve the initialization of speech and line side interrupts, speech processing using one of the quantizer gain update strategies as discussed in Section 3.2, transmission of error code and synthesis of speech afterwards.

As before, the two interrupts, namely the speech and lineside interrupts are used. The former obtains its signal from a 12-bit A/D at a constant rate and then output a 12 bit processed sample for a 12-bit D/A. The latter governs the transmission of data. At every interrupt, it outputs a transmitter clock pulse and decodes one bit of received information. At every 8 interrupts, it inputs a new eight bit receiver word. The program transfers control to a wait loop after it finishes performing either transmitter or receiver functions. Since there is no framing necessary in the CVSD algorithm, synchronization is not needed.
Figure 3-21. Block Diagram of CVSD-D
3.4.1 Wait Loop

Figure 3-22 shows a flow chart of the wait loop. Its function, as mentioned earlier, is to use up all the time left over after processing operations. It also sequences the transmitter and receiver operations by examining the transmitter and receiver ready flops and examines the front panel to see if initialization is desired.

3.4.2 The Line Side and Speech Side Interrupts

Figures 3-23 and 3-24 describe the software operations when interrupts occur. The function of these interrupts are discussed in Sections 3-14 and 3-15.

3.4.3 Speech Analysis and Synthesis

Figures 3-25 and 3-26 describe the software operation of the transmitter and receiver of CVSD-B. It allows a selection of transmission rates (9.6 kBPS or 16.0 kBPS), time constants in gain filter (6.3 msec, 10 msec, 20 msec), time constants in prediction filters (1 msec, 2 msec, 10 msec, 20 msec) and compression ratios (10, 20, 60, 100, 150, 200). Figure 3-27 and 3-28 show the software operation of Forney and Qreshi's CVSD-C. Figures 3-29 and 3-30 describe the operation of CVSD-D transmitter and receiver where one bit of memory is used.

3.5 Operating Procedures of CVSD Program

3.5.1 Connecting the System

Connect the EDM's to each other or through a modem as detailed in Section 3.2.1.

3.5.2 Reading in the EDM Software

Read in the CVSD program as shown in Section 3.2.2.

3.5.3 Initialization of the EDM

After the program has been executing, reinitialization can be achieved by simultaneously holding switch register toggle switch 15
CLEAR ALL TRANSMITTER AND RECEIVER PARAMETERS

EXAMINE FRONT PANEL SWITCHES

BIT 15 SET?

YES

NO

BIT 7 SET?

YES

NO

SET COUNTERS FOR 9.6 KBPS TRANSMISSION RATE

SET COUNTERS FOR 16 KBPS TRANSMISSION RATE

READY TO XMIT?

CLEAR XMIT READY FLAG

GO TO TRANS

CLEAR RECEIVER READY FLAG

GO TO RCVR

READY TO RECEIVE?

GO TO INIT

Figure 3-22. Initialization and Wait Loop
Figure 3-23. Interrupt Loop Speech Side
Figure 3-24. Interrupt Loop Line Side
Figure 3-25. Transmitter for CVSD-B
Figure 3-26. Receiver for CVSD-B
Figure 3-27. Transmitter for CVSD-C
Figure 3-28. Receiver for CVSD-C
Figure 3-29. Transmitter for CVSD-D
Figure 3-30. Receiver for CVSD-D
in the down position for each EDM and then returning it to the up position.

3.5.4 Program Options

The CVSD program enables selections of two transmission rates, three slope adaptation logics, six compression ratios, three time constants for gain filter and four time constants for the prediction filter. The EDM switch settings and their functions are summarized as follows:

1) Initialization

   BIT 15 down = initialization
   up = program execution

ii) Transmission Rates

   BIT 7 down = 16.0 kBPS
   up = 9.6 kBPS

iii) Slope adaptation logics

   a) BIT 8 and BIT 9 up = standard 2-bit (CVSD-B)

<table>
<thead>
<tr>
<th>BIT 2</th>
<th>BIT 1</th>
<th>BIT 0</th>
<th>Compression Ratios</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>200</td>
</tr>
</tbody>
</table>

BIT 4 BIT 3 Gain Filter Time Constant

<table>
<thead>
<tr>
<th>BIT 4</th>
<th>BIT 3</th>
<th>Gain Filter Time Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6.3 msec</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10.0 msec</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>20.0 msec</td>
</tr>
<tr>
<td>BIT 6</td>
<td>BIT 5</td>
<td>Prediction Filter Time Constant</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1.0 msec</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2.0 msec</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10.0 msec</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20.0 msec</td>
</tr>
</tbody>
</table>

b) BIT 8 down and BIT 9 up = Forney and Qreshi's 3-BIT (CVSD-C)

c) BIT 8 up and BIT 9 down = Jayant's 1-bit (CVSD-D)
3.6 I/O Speech Filters

Under this contract, GTE Sylvania supplied 4 identical I/O Speech Filters, two for each DCA PSP. The I/O speech filters are passive, low-pass elliptic filters that exhibit an essentially flat passband characteristic (0-3200 Hz at less than 0.5 dB ripple) followed by a very sharp transition region occurring just below their 3 dB attenuation frequency. These filters, supplied by ESC Electronics Corporation, Pallisades, N.J., are very stable and possess a very low noise figure as a result of their passive design. The mechanical and electrical specifications insure compatibility with the DCA PSP's and they can be plugged into card number 1 of the PSP without any modifications. Figure 3-31 provides the frequency loss characteristics of the filter (ESC part number ESC43F61) whose input and output impedance is 5000 ohms. These filters have a pin arrangement assure that will be inserted correctly into the PC board. Care should taken, however, to not damage the filter pins when inserting or removing the filter from the card.
Figure 3-31. I/O Filter Characteristics
4.1 Introduction

Because of the digital signal processing involved in the APCQ technique, its implementation should be all digital. Today large scale integration (LSI) fabrication techniques provide digital designs having lower cost, smaller size, greater reliability and less power consumption.

Under this contract, GTE Sylvania examined different LSI devices and organized them into a signal processor suitable for APCQ implementation. The processor design has a modular, bus-oriented architecture with highly program controlled parallelism.

The form of design will provide an implementation of the APCQ coder which uses about 60% of the processing time available. Our estimates indicate that the APCQ algorithm will require about 16 msec. to analyze speech and to synthesize speech during each 25 msec. data frame. Thus, the processor hardware idles about 9 msec. out of each frame. This idle time can be used for other telephone subscriber functions, such as encryption and telephone line control.

The cost of parts is also surprisingly low. Today's cost for parts in small quantities is $700 and their power requirements are just over 42 watts. Thus, the APCQ coder, while mathematically complex, can be built today using commercially available LSI processors for a relatively low cost.

The next section will present the architecture of the design followed by a discussion of the LSI software implementation of the APCQ algorithm. Estimates of timing, memory, cost and power
Figure 4-1. Architectural Concept
are then presented in Section 4.3

The processing through out of the design can certainly be increased by adding higher degrees of parallelism. In Section 4.4, we illustrate how adding an index processor or an external hardware multiplier or both can improve performance through increased hardware complexity and through a rearrangement of the busing structure. The estimates of these different designs are included as a comparison.

4.2 Basic LSI Signal Processor Architecture

The basic LSI signal processor architecture for APMQ implementation consists of a macro processor, a loop processor, memory and peripheral devices. They are interconnected via two main buses as shown in Figure 4-1.

Each of the two processors is designed to perform the tasks for which it is best suited: the macro processor for setup and program control, the loop processor (LP) for array processing in 16-bit or greater precision and the handling of interrupts at a high rate.

The macro processor is a mini-computer on a chip. The Texas Instrument TMS 9900 is chosen as this 16-bit processor. It is designed to be capable of running at a 3 MHz micro-cycle rate.

The 16-bit main memory contains both the working storage (both constants and variables) for the macro processor and the LP and the program memory for the macro processor.

The loop processor is the fast arithmetic and logic unit. As shown in Figure 4-2, it consists of an A Register to interface with the main memory and the powerful ALU chips, which are
Figure 4-2. Loop Processor
the Advanced Micro Devices AM2901 4-bit slice micro processors.

Output of the ALU chip can be either latched through the output Register (O register) to the data bus (D Bus) or through memory address register (MA register) to the address bus (A Bus). Additional A Bus to D Bus connection is provided in the design for added memory access flexibility.

The micro program memory and control hardware governs the activities of the LP. The control hardware includes micro program controlled storage of ALU status, micro jump control, subroutine nesting and priority interrupt handling.

Both interrupting and non-interrupting I/O ports are used. Each port can be individually addressed and commanded via the A Bus, either placing data onto the D Bus or accepting data from it. Interrupting ports issue service requests to the LP, which controls the buffering of input or output data on a priority interrupt basis. Non-interrupting ports can be accessed at any time by either the macro processor or the LP.

All bus-driving points use three-state output devices widely available in current technology at bipolar speeds. All devices in our design are off-the-shelf LSI, MSI and SSI Schottky bipolar chips.

The main ALU, the AM 2901, is believed to be the most powerful micro processor chip available at the present time. It meets all military specifications and can function at a clock rate of 6.66 MHz, or 150 ns cycle time.

One key feature of the processor is its 16 word 2-port RAM. The RAM serves as an on-chip register file. 2-port makes it
easy to access two registers at the same time. In addition, there is the Q register, which is intended primarily for multiplication and division routines. The detailed AM 2901 block diagram is shown in Figure 4-1.

The signal flow of the processor shown in Figure 4-2 can be explained as following. Input data are fed from data memory to the D Bus (Data Bus) and latched by the A register. They can either be saved in one of the locations in the 2-port RAM or directly feed the ALU, which does the arithmetic or logical operations. Y output from the ALU can be fed back to the processor for further ALU manipulation or can be latched to the O register or the MA register. This data can, therefore, be brought back to data memory via the D Bus or used as a memory address pointer to access new data from memory.

The branching hardware, which is not shown in Figure 4-2, looks at the result of the previous cycle. All branches are executed at the end of the cycle following the instruction containing the branch. (This is the result of a pipelined instruction fetch). All the operations are controlled by the micro program instructions and synchronised to 13.33 MHz single phase clock.

4.3 Estimates of LSI Implementation of APCQ Algorithm

This section presents the transmitter operations of the APCQ algorithm. All sections of the transmitter are analyzed and some are shown as examples. Overall timing and memory estimates of the entire APCQ implementation are supplied. Finally, the cost and power of the LSI processor based on counted chips are documented.

4.3.1 APCQ Implementation

LSI design implementation of APCQ Algorithm is based on
the PSE implementation of the same algorithm. The transmitter operations are listed in Figure 4-4.

![Diagram of transmitter operations]

**Figure 4-4 APCQ Transmitter Operations**

Each task in the transmitter is implemented as separate software for execution in the loop processor (LP). Many of the software tasks involve repeated calculations on data arrays. These calculations are performed in software loops. The macro processor will handle the loop and interrupt control. Therefore, the overall timing and memory can be estimated as those required in the LP, the macro processor and some overhead.

The use of the macro processor is presented conceptually in Figure 4-5 along with the LP operational framework and memory map. Within this organization, the macro processor acts as the master computer, and the LP as a programmable peripheral processor for doing the complex computations required by the processing algorithm.
This technique of resource use hinges on the use of two "swap areas" in memory, accessed by one processor at a time, with software control of access via two "swap flops". Initially, both "swap flops" are set to 1 and the macro processor is granted access. The LP simply spins in a loop waiting for one of the SF's to become 0.

The macro processor then sets up for a loop processor "CALL" by placing subroutine parameters into a swap area and clearing the associated SF. The LP (when finished with processing in the other area, if any) will then access the swap area and perform the called routine.

The macro processor (after checking for access to the remaining swap area) then sets up for the next LP CALL and clears that "swap flop".

Note that the macro processor always does all the work it can, until both swap areas are given over to the LP and, once given access to a swap area, the macro processor accepts data from it (when appropriate) before setting up the next LP CALL.

The LP, in its turn, simply obeys the CALL when given access to a "swap area", completing the subroutine (such as an AMDF or Autocorrelation) and then returning access to the macro processor.

Two levels of interrupt processing are indicated at the LP, one for INPUT/OUTPUT service and one to support data transfer to the monitor interface for test and debugging operations. These operations are assigned to the LP due to its speed in the first case and due to its ready access to fast registers in the second case.
This framework provides for full resource use within the confines of any given organization of a processing algorithm with both the macro processor and loop processor working simultaneously whenever possible. With this protocol as background, we proceed below to concentrate on the LP subroutines for the APCQ algorithm.

4.3.2 Examples

The functional Activity Diagram for the Loop Processor is used to analyze the APCQ implementation. All transmitter operations of the APCQ coder were analyzed and examples of the more time consuming tasks are provided on the following pages.
a. Autocorrelation Function

Figure 4-6 is the functional activity diagram of the inner loop of the 1th autocorrelation. Software multiplication is initialized by loading the multiplier into the 3 register and the multiplicand into R3. The product appears in R1 and Q. The 32-bit result is accumulated in double precision in location R9 and R5. CO represents the carry out of the low-order accumulation. R10 is used as a loop control by presetting it to 128.

The outer computation loop, which is not shown for the sake of clarity, performs the functions of storing R6, R5, increasing R2 up to maximum, setting R0, R10, R2, etc.

The outer loop can be performed in 15 machine cycles by the LP. The inner loop takes 22 cycles. Therefore, the total processing cycle for this function is:

\[ 22 \times 128 \times 5 + 15 \times 5 = 14155 \text{ cycles} \]

Autocorrelation will also be normalized such that C0=1. Thus, four normalizations will require 4 multiplications and 1 division.

The total time to calculate the autocorrelation functions plus normalization is:

\[ 14155 + 5 \times 22 + 132 = 14397 \text{ cycles} \]

b. AMDF

Figure 4-7 diagrams the inner loop of the AMDF. R5 is used to increase the memory address by three. R1 and R2 are initialized at the memory address index. The results are accumulated in register pair R3 and
<table>
<thead>
<tr>
<th>PROGRAM SEQUENCE</th>
<th>A REG</th>
<th>MA REG</th>
<th>O REG</th>
<th>FUNCTION</th>
<th>BRANCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R0</td>
<td></td>
<td></td>
<td>R0 → MA, R0 - 1 → R0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X_i</td>
<td></td>
<td></td>
<td>R0 + R_2 → MA</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X_{i+j-1}</td>
<td></td>
<td></td>
<td>D → R_3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>D SRA Q, R_3 SRA R_3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 cycles multiplication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>Result in R_4, Q</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>R_{10} - 1 → R_{10}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>R_4 + R_1 → R_4</td>
<td>Branch to 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>R_5 + CO → R_5</td>
<td>Branch exit if R_{10} = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Autocorrelation \[ C_j = \sum_{i=1}^{128} X_i \cdot X_{i+j-1} \] for j = 0, 1, 2, 3, 4

Figure 4-6. Autocorrelation Function Inner Loop
<table>
<thead>
<tr>
<th>PROGRAM SEQUENCE</th>
<th>D</th>
<th>MA REG</th>
<th>O REG</th>
<th>FUNCTION</th>
<th>BRANCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>$R_7 - 1 \rightarrow R_7$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>$R_1 + R_5 \rightarrow R_1$, MA</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$S_1$</td>
<td>i-t</td>
<td>$R_2 + R_5 \rightarrow R_2$, MA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$S_1$</td>
<td>i-t</td>
<td></td>
<td>$D \rightarrow R_0$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>$R_0 - D \rightarrow R_0$</td>
<td>SAVE SIGN IN L</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td>$R_1 + R_5 \rightarrow R_1$, MA</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$S_1$</td>
<td>i-t</td>
<td>$R_2 + R_5 \rightarrow R_2$, MA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$S_1$</td>
<td>i-t</td>
<td></td>
<td>$D \rightarrow R_4$</td>
<td>Branch to 16 if L=1</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>$R_4 - D \rightarrow R_4$</td>
<td>SAVE SIGN IN Z</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>$R_3 + R_0 \rightarrow R_3$</td>
<td>Branch to 13 if Z=1</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>$R_6 + CO \rightarrow R_6$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>$R_3 + R_4 \rightarrow R_3$</td>
<td>Branch to 1</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>$R_5 + CO \rightarrow R_6$</td>
<td>Branch to exit if $R_7=0$</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>$R_3 - R_4 \rightarrow R_3$</td>
<td>Branch to 1</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>$R_6 + CO \rightarrow R_6$</td>
<td>Branch to exit if $R_7=0$</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>$R_3 - R_0 \rightarrow R_0$</td>
<td>Branch to 13 if Z=1</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td>$R_6 + CO \rightarrow R_6$</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>$R_3 + R_4 \rightarrow R_3$</td>
<td>Branch to 1</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>$R_6 + CO \rightarrow R_6$</td>
<td>Branch exit if $R_7=0$</td>
</tr>
</tbody>
</table>

AMDF: $A_t = \sum_{i=1,3\ldots}^{72} \left| S_1 - S_{1-t} \right|$  \hspace{1cm} t = +15, ···, 45

Figure 4-7. AMDF Inner Loop
$R_6$ for double precision addition. CO represents the carry out of the lower sum. The loop computes and accumulates two absolute differences, therefore, it must be employed 12 times for each of 60 AMDF points. $R_7$ is used as a loop control, it is preset at 12.

The inner loop requires 13 cycles per iteration. The outer loop for the AMDF must reset the memory address pointer, store results etc.; it requires nine machine cycles. Therefore, the LSI processor can implement the AMDF in:

$$13 \times 12 \times 60 + 9 \times 60 = 9900 \text{cycles}$$

c. The Reduced Waveform

The speech signal is filtered by a one-tap filter using the pitch gain, $a$, as the tap coefficient. Figure 4-8 is the functional activity diagram of the inner loop of the reduced waveform. $R_2$ is the memory address pointer to pick up the pitch gain, $a$. $R_1$, $R_6$, $R_4$ are also used as pointers to pick up data from memory or store data back into memory. The inner loop takes 24 cycles to perform the function. 15 more cycles are necessary to initialize the memory pointing register and some other overhead operations. Therefore, total required cycles are:

$$15 + 24 \times 128 = 3087$$

d. The Error Signal

Figure 4-9 is the functional activity diagram of the inner loop of error signals calculation. $R_7$, $R_{10}$ and $R_{11}$ are used to accumulate the results. The registers $R_0$, $R_1$, $R_3$, $R_4$ are used as memory address
<table>
<thead>
<tr>
<th>PROGRAM SEQUENCE</th>
<th>D</th>
<th>MA REG</th>
<th>O REG</th>
<th>FUNCTION</th>
<th>BRANCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>n-M</td>
<td></td>
<td>R₁ - R₀ → MA</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sₙ₋ₘ</td>
<td>R₂</td>
<td></td>
<td>R₂ → R₂, MA</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td></td>
<td>Sₙ₋ₘ → R₃</td>
<td>16 cycles multiplication a * Sₙ₋ₘ</td>
</tr>
<tr>
<td>4</td>
<td>a → Q</td>
<td>Sₙ₋ₘ → R₃</td>
<td>Sₙ₋ₘ → R₃</td>
<td>a → Q, Sₙ₋ₘ → R₃</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>Result in R₀, Q</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>R₁</td>
<td></td>
<td></td>
<td>R₁ → MA</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Sₙ</td>
<td></td>
<td></td>
<td>R₅ - ₁ → R₅</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Sₙ₋₀</td>
<td>Sₙ₋₀ → R₀</td>
<td>Sₙ₋₀ → O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>R₄</td>
<td></td>
<td></td>
<td>R₄ - ₁ → R₄, R₄ → MA</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>O → M(MA)</td>
<td>Branch 2</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>n-M - ₁</td>
<td>R₁ - R₀ - ₁ → MA</td>
<td>Branch exit</td>
<td>IF R₅ = 0</td>
<td></td>
</tr>
</tbody>
</table>

\[ Vₙ = Sₙ - aSₙ₋ₘ \quad n = 1, 2 \cdots 128 \]

**Figure 4-8. Reduced Form Inner Loop**
<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>D</th>
<th>MA REG</th>
<th>O REG</th>
<th>FUNCTION</th>
<th>BRANCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>R0</td>
<td></td>
<td>R₀ → MA, R₀</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a₁</td>
<td>R₁ - R₀</td>
<td></td>
<td>R₁ - R₀ → Mₐ</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Vₙ₋₁</td>
<td>a₁ → Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Vₙ₋₁</td>
<td>SRA R₈ → Q, SRA Q → Q</td>
<td>a₁ * Vₙ₋₁</td>
<td>16 cycles multiplication</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>Result in R₇, Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>R₅ - 1 → R₅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>R₁₀ + R₇ → R₁₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>R₁₁ + C₀ → R₁₁</td>
<td></td>
<td></td>
<td>Branch to 2</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>R₀ - 1 → R₀, R₀ → MA</td>
<td></td>
<td></td>
<td>Branch to 24 if R₅ = 0</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>R₁ - R₃ → MA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Sₙ₋₉</td>
<td>R₄</td>
<td></td>
<td>R₄ → MA</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>a</td>
<td>Sₙ₋₉ → R₈</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>a</td>
<td>SRA Q Sₙ₋₉ → R₈, SRA R₈</td>
<td>a * Sₙ₋₉</td>
<td>16 cycles multiplication</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td>Result in R₇, Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
<td>R₁ → MA R₁ - 1 → R₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Sₙ</td>
<td>R₁₀ + R₇ → R₁₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>C₀ + R₁₁ → R₁₁</td>
<td></td>
<td></td>
<td>Branch 1</td>
</tr>
<tr>
<td>46</td>
<td>Sₙ</td>
<td>Sₙ - R₁₀ → R₁₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td>If R₁ = 0 branch exit</td>
</tr>
</tbody>
</table>

\[
C_n = S_n - a S_{n-M} - \sum_{i=1}^{4} a_i V_{n-i} \quad n = 1, 2, \ldots, 128
\]

Figure 4-9. Second Reduced Signal Inner Loop
pointers. \( R_5 \) is used as a loop control.

The inner loop takes \( 22 \times 4 + 23 \) cycles. The outer loop, which performs the loop control, register initialization and stores the results back in memory requires an additional 12 cycles. Therefore, the total required cycles are:

\[
(22 \times 4 + 23) \times 128 + 12 \times 128 = 15744
\]

e. PARCOR

Calculation coefficients of the PARCOR coefficients is a recursive process. For sake of clarity, Figure 4-10 shows only the cycled multiplication part, which contains most of the calculations.

Registers \( R_0, R_2 \) are used as memory address pointers. \( R_6 \) is used as a loop counter. Double precision accumulation ends up in \( R_3 \) and \( R_4 \). Each term in the calculation takes two multiplications and each of them cycles 1, 2 and 3 times. In addition, two separate multiplications and four divisions are necessary to complete the task. 30 overhead cycles are added to guarantee enough cycles for memory address indexing, loop control and storing results back into memory, etc.

Therefore, the required cycles are:

\[
30 \times 3 + 20 \times (1 + 2 + 3) + 20 \times (1 + 2 \times 3) + 132 \times 4 + 20 = 878
\]
<table>
<thead>
<tr>
<th>PROGRAM SEQUENCE</th>
<th>A REG</th>
<th>MA REG</th>
<th>O REG</th>
<th>FUNCTION</th>
<th>BRANCHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>$R_0 - R_2$</td>
<td></td>
<td>$R_0 - R_2 \rightarrow MA$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A(I-M)</td>
<td></td>
<td></td>
<td>$D \rightarrow R_1$</td>
<td>B(M) = A(I-M)</td>
</tr>
<tr>
<td>3</td>
<td>$R_0$</td>
<td></td>
<td></td>
<td>$R_0 \rightarrow MA$, $R_0 + 1 \rightarrow R_0$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B(M)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>$R_1 \xrightarrow{SRA} R_1$, $D \xrightarrow{SRA} Q$</td>
<td></td>
<td>16 cycles multiplication A(I-M)*B(M)</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>Result in $R_5$, Q</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>$R_3 + R_5 \rightarrow R_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>$R_4 \rightarrow CO \rightarrow R_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>$R_5 - 1 \rightarrow R_6$</td>
<td></td>
<td>Branch to 1</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>NOP</td>
<td></td>
<td>Branch exit if $R_6 = 0$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-10. Partial PARCOR Inner Loop
4.3.3 Timing and Memory Estimates

The overall program timing is derived from Figure 4-3. Total required time for the transmitter side is calculated with some of the calculations taken from the examples shown in the previous section. The result is shown in Table 4-1 below:

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>TIME (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Normalization</td>
<td>0.71</td>
</tr>
<tr>
<td>I/O Interrupt</td>
<td>0.84</td>
</tr>
<tr>
<td>AMDF</td>
<td>1.49</td>
</tr>
<tr>
<td>Pitch gain</td>
<td>0.84</td>
</tr>
<tr>
<td>Reduced waveform normalization</td>
<td>0.89</td>
</tr>
<tr>
<td>Autocorrelation normalization</td>
<td>2.16</td>
</tr>
<tr>
<td>PARCOR quantization dequantization</td>
<td>0.13</td>
</tr>
<tr>
<td>Predictor coefficients</td>
<td>0.03</td>
</tr>
<tr>
<td>Error signal</td>
<td>2.34</td>
</tr>
<tr>
<td>Error signal quantization</td>
<td>1.17</td>
</tr>
<tr>
<td>Q, quantization level</td>
<td>0.04</td>
</tr>
<tr>
<td>Total time</td>
<td>10.62 ms</td>
</tr>
</tbody>
</table>

**TABLE 4-1**

The Estimates of Transmitter Timing of APCQ

In the APCQ Algorithm, the receiver processing is one-third of the complexity of the transmitter. Therefore, the estimated timing is one-third of that required for the transmitter. Also, allowing 15 percent overhead time for subroutine control, data handling and synchronization, etc., then the overall processing of APCQ algorithm implementation is:
10.62 \times (1 + 33\%) \times (1 + 15\%) = 16.2 \text{ msec}

Memory used in the LSI processor can be divided into micro and macro memories. Micro memory having a word length of 48 bits is used for micro program storage.

Macro memory having a word length of 16 bits, contains data and the macro program.

The amount of Micro program memory (ROM) is conservatively estimated as shown in Table 4-2:

<table>
<thead>
<tr>
<th>Table 4-2 Microprogram Memory Requirements</th>
<th>Transmitter</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Normalization</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>AMPF</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>$a$, pitch gain</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Division subroutine</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Square root subroutine</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>Quantization subroutine</td>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>Dequantization subroutines</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Reduced waveform and normalization</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>Autocorrelation and normalization</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Predictor coefficients</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Error signal</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Initialization and synchronization</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>$Q$, quantization level</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>PARCOR</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>940</td>
<td>355</td>
</tr>
</tbody>
</table>
Besides the memory required as shown in Table 4-2, we should allow 15% overhead for memory for inputting and outputting the digital data. Therefore, total memory is:

\[(940 + 355) \times (1 + 15\%) = 1490\]

or \[1490 \times 48 = 71520\] bits. This is 6 chips of 16K ROM (1K x 16)\(^*\) with 500 spare words.

Macro program I/O interrupts, control memory and data buffer for the processor are as shown in Table 4-3.

<table>
<thead>
<tr>
<th>Data buffer</th>
<th>5 x 128 + 2 x 204</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMDF</td>
<td>100</td>
</tr>
<tr>
<td>I/O interrupts + macro program control</td>
<td>350</td>
</tr>
<tr>
<td>Total</td>
<td>1498</td>
</tr>
</tbody>
</table>

**TABLE 4-3 : Macroprogram Memory Requirements**

or \[1498 \times 16 = 23968\] bits. This is 6 chips 4K RAM (512 x 8)\(^*\).

In addition to these memory chips constant data memory should require 1 chip of 16K ROM (1K x 16)\(^*\).

To sum up the above estimation, the estimated memory are:

- 4K RAM 6 chips
- 16K ROM 7 chips

\(^*\) Memories are projected to 1978 technology.
<table>
<thead>
<tr>
<th>DEVICE</th>
<th>I/O</th>
<th>QTY.</th>
<th>POWER</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>5</td>
<td>.180</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>3</td>
<td>.348</td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>14</td>
<td>1.596</td>
<td>2.52</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>6</td>
<td>0.456</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>0.076</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>0.070</td>
<td>0.46</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>2</td>
<td>0.320</td>
<td>0.46</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>1</td>
<td>0.340</td>
<td>0.59</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>1</td>
<td>0.250</td>
<td>0.17</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>7</td>
<td>1.575</td>
<td>5.56</td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>1</td>
<td>0.190</td>
<td>0.58</td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>2</td>
<td>0.450</td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>3</td>
<td>0.750</td>
<td>1.92</td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>3</td>
<td>1.781</td>
<td>2.37</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>9</td>
<td>4.61</td>
<td>6.75</td>
<td></td>
</tr>
<tr>
<td>182</td>
<td>1</td>
<td>0.36</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>8</td>
<td>5.44</td>
<td>20.56</td>
<td></td>
</tr>
<tr>
<td>251</td>
<td>2</td>
<td>0.550</td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>253</td>
<td>10</td>
<td>.34</td>
<td>5.00</td>
<td></td>
</tr>
<tr>
<td>276</td>
<td>7</td>
<td>2.10</td>
<td>3.78</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>1</td>
<td>0.61</td>
<td>10.00</td>
<td></td>
</tr>
<tr>
<td>374</td>
<td>4</td>
<td>1.8</td>
<td>10.84</td>
<td></td>
</tr>
<tr>
<td>2401</td>
<td>4</td>
<td>3.7</td>
<td>120.00</td>
<td></td>
</tr>
<tr>
<td>482</td>
<td>3</td>
<td>1.426</td>
<td>30.00</td>
<td></td>
</tr>
<tr>
<td>9900</td>
<td>1</td>
<td>1.000</td>
<td>60.00</td>
<td></td>
</tr>
<tr>
<td>4K RAM</td>
<td>6</td>
<td>3</td>
<td>36.00</td>
<td></td>
</tr>
<tr>
<td>16K ROM</td>
<td>7</td>
<td>4.2</td>
<td>101.50</td>
<td></td>
</tr>
<tr>
<td>12-Bit A/D</td>
<td>1</td>
<td>2.43</td>
<td>195.00</td>
<td></td>
</tr>
<tr>
<td>8-Bit D/A</td>
<td>1</td>
<td>1.6</td>
<td>49.00</td>
<td></td>
</tr>
</tbody>
</table>

| TOTAL | 119 | 41.748 Watts | $669.36 |

Materials, Quantities, Power and Cost

TABLE 4-4
4.3.4 Cost and Power Estimates

The cost and power estimate of the LSI processor was performed by adding up the cost and power needs of each chip. Losses caused by the PC boards, mechanic assembly, power supply, etc., were not included in the estimate, but these are estimated to be small. The parts list, total power and cost are shown in Table 4-4.

4.4 Other LSI Processor Structures

The parallelism of the proposed LSI processor can be further increased by changing the loop processor as shown in Figure 4-11 to include an index processor and hardware multiplier. Now the loop processor presents three functionally independent processors.

The index processor is used for the computation of memory addresses. It consists of a 16-element file of data plus an 8-element array of pointers with a limited ALU.

The hardware multiplier performs 16-bit by 16-bit two's complement multiplication providing 32-bit results in QH and QL registers (16 bits each). Once loaded and started, it will complete its product in 8 cycles (or 4 or 2 depending on design) while other operations can continue in parallel in both the index processor and main ALU.

The main arithmetic/logic unit is the same as we mentioned before. However, its memory addressing function can be performed by the index processor and its multiplication function can be performed by the multiplier processor.

These three sections of the loop processor run synchronously under microprogram control.

In this section, we consider adding either the index processor or the multiplier processor or both, to the main
Figure 4-11. Loop Processor with Added Parallelism
ALU. The timing, memory, cost and power will be re-established to compare with the previous single main ALU processor.

4.4.1 Timing and Memory Estimates

With an index processor, the memory address pointer can access memory locations at the same time the main ALU performs other operations. It can, therefore, save one micro program instruction at the beginning of every memory index. With a hardware multiplier, a multiplication requires 10 cycles with an index processor and 11 cycles without an index processor. Without a hardware multiplier, multiplication requires 20 cycles with an index processor and 22 cycles without an index processor. Hence, the use of a hardware multiplier saves at least half the micro instructions. The added parallelism has increased the complexity of bus structures somewhat and the word length of micro instruction by three bits for the multiplier and 13 bits for the index processor.

The timing of the transmitter side of the APCQ implementation for the different structures is analyzed and listed in Table 4-6.

The receiver processing time is considered to be one-third the required time for the transmitter. Overhead time is 15 percent of the total time required for the transmitter and the receiver.

From Table 4-5, we can conclude that adding a hardware multiplier improves the processing speed by about 30 to 50%. Adding an index processor improves the processor speed by 16 to 25%.

* It is almost always possible to perform additional useful operations in either the ALU or the index processor while the multiplier is performing its function such that the multiply time is effectively reduced.
<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>With Index Processors</th>
<th>Without Index Processors</th>
<th>TIME (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
<td></td>
</tr>
<tr>
<td>Input Normalization</td>
<td>0.28</td>
<td>0.56</td>
<td>0.38</td>
</tr>
<tr>
<td>I/O Interrupt</td>
<td>0.76</td>
<td>0.76</td>
<td>0.84</td>
</tr>
<tr>
<td>AMDF</td>
<td>0.95</td>
<td>0.95</td>
<td>1.49</td>
</tr>
<tr>
<td>a, Pitch gain</td>
<td>0.34</td>
<td>0.66</td>
<td>0.36</td>
</tr>
<tr>
<td>First reduced form &amp; normalization</td>
<td>0.42</td>
<td>0.66</td>
<td>0.42</td>
</tr>
<tr>
<td>Autocorrelation &amp; normalization</td>
<td>0.89</td>
<td>1.77</td>
<td>1.18</td>
</tr>
<tr>
<td>PARCOR &amp; quantization &amp; de-quantization</td>
<td>0.07</td>
<td>0.13</td>
<td>0.07</td>
</tr>
<tr>
<td>Predictor coefficients</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>Second reduced form (error signal)</td>
<td>0.99</td>
<td>1.77</td>
<td>1.23</td>
</tr>
<tr>
<td>Error signal quantization</td>
<td>0.69</td>
<td>1.08</td>
<td>0.75</td>
</tr>
<tr>
<td>Q, quantization level</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>Transmitter processing time</td>
<td>5.95</td>
<td>8.51</td>
<td>6.77</td>
</tr>
<tr>
<td>Receiver processing time</td>
<td>1.85</td>
<td>2.84</td>
<td>2.56</td>
</tr>
<tr>
<td>Overheads</td>
<td>1.11</td>
<td>1.70</td>
<td>1.31</td>
</tr>
<tr>
<td>Total time</td>
<td>8.51</td>
<td>13.05</td>
<td>10.63</td>
</tr>
</tbody>
</table>

**TABLE 4-5**

Processor Time Requirements vs. Hardware Complexity
With the different structures micro program memory is different as is the micro instruction word length and memory size.

Table 4-6 shows these differences. Overhead cycles have taken 15% from the total transmitter and receiver memory.

From the table, it is clear that with the indicated index processor added, the overall memory required is increased, because of the increase of micro instruction word length. Other index processors with greater computational capability can provide for greater speed increases with approximately the same increase in micro control memory.

The total memory, which includes both macro and micro memory will be as shown in Table 4-7.
<table>
<thead>
<tr>
<th></th>
<th>With Index Processor</th>
<th>Without Index Processor</th>
<th>With Index Processor</th>
<th>Without Index Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
</tr>
<tr>
<td>Micro instruction</td>
<td>Word length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmitter</td>
<td>Memory size</td>
<td>605</td>
<td>780</td>
<td>730</td>
</tr>
<tr>
<td>Receiver</td>
<td>Memory size</td>
<td>205</td>
<td>265</td>
<td>275</td>
</tr>
<tr>
<td>Overheads</td>
<td></td>
<td>122</td>
<td>157</td>
<td>151</td>
</tr>
<tr>
<td>Total Bits</td>
<td></td>
<td>59648</td>
<td>73322</td>
<td>60112</td>
</tr>
<tr>
<td>Required</td>
<td>16K ROM chips</td>
<td>4 (with 800 spare words)</td>
<td>8 (with 800 spare words)</td>
<td>8 (with 500 spare words)</td>
</tr>
</tbody>
</table>

**TABLE 4-6**

Memory Requirements vs. Hardware Complexity
<table>
<thead>
<tr>
<th></th>
<th>With Index Processor</th>
<th>Without Index Processor</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
</tr>
<tr>
<td>16K ROM for micro-memory</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4K RAM for macro-memory</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>16K ROM for macro-memory</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE 4-7 Number of Memory Chips
<table>
<thead>
<tr>
<th>Adjustment</th>
<th>With Index Processor</th>
<th>Without Index Processor</th>
<th>Without Index Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Multiplier</td>
<td>Without Multiplier</td>
<td>With Multiplier</td>
</tr>
<tr>
<td>Multiplier</td>
<td>+2.5</td>
<td>50</td>
<td>2.5</td>
</tr>
<tr>
<td>Index Processor</td>
<td>5.11</td>
<td>86</td>
<td>5.11</td>
</tr>
<tr>
<td>Memory</td>
<td>-1.2</td>
<td>-29</td>
<td>+1.2</td>
</tr>
<tr>
<td>Base</td>
<td>41.158</td>
<td>669.36</td>
<td>41.748</td>
</tr>
<tr>
<td>Total</td>
<td>48.158</td>
<td>$776.36</td>
<td>48.058</td>
</tr>
</tbody>
</table>

**TABLE 4-8** Cost and Power of Various Forms of LSI Processors
4.4.2 Estimates of Cost and Power of Index Processor and Multiplier

The cost and power required for the index processor is $85 and 5.11 watts respectively. For the multiplier they are $50 and 2.5 watts respectively.

Therefore, the cost and power for the modified processor appear in Table 4-8.

4.5 Discussions

Table 4-5 and Table 4-8 have shown us many significant results. The cost and power do not differ greatly for the different structures, but the speed does. The best choice for APCQ seems to be the processor with multiplier, but without index processor.
SECTION 5

RECOMMENDATIONS AND CONCLUSIONS

5.1 Conclusions

This contract has resulted in the development of a high quality 16 Kb/s speech digitizer based on adaptive predictive coding principles which is robust to channel errors and relatively simple to implement. Informal listening tests indicate that the speech quality of the 3000 Hz audio bandwidth adaptive predictive coder is noticeably superior to Continuous Variable Slope Deltamodulation (CVSD) which we also developed as a benchmark. Experimentation with the CVSD algorithms indicated they could be improved slightly by adjusting internal parameters, especially the compression ratio defined as the maximum quantizer level over the minimum level.

Although this change lessened the slope overload distortion, making the speech sound less muffled, its voice quality was always lower than that of the Adaptive Predictive coder at 16 Kbps.

Using commercially available LSI electronics the 16 Kbps adaptive predictive coder could be implemented using about $700 in parts and would consume about 42 watts of power. Our engineering estimates of the hardware incorporated the Advanced Microprocessor Devices AMD2901 microprocessor chip to develop a programmable multilevel adaptive predictive coder.

This approach, while more complex electronically than that used for CVSD, would result in a complete telephone terminal with built in telephone line synchronization, modem interface and key interface at about the same cost as a terminal employing a CVSD speech digitizer. This occurs because the programmable approach
can timeshare the microprocessor hardware among the speech
digitizer and these other tasks while the terminal with the CVSD
digitizer must have special hardware for each task. Because they
must have this specialized hardware, present CVSD terminals are
surprisingly costly, even though the CVSD digitizer is extremely
simple. Thus, for about the same terminal cost, the multilevel
adaptive predictive coders developed under this contract would
noticeably improve the voice quality over the CVSD. Consequently,
these coders should be studied and refined still further.

5.2 Recommendations

The final multilevel adaptive predictive coder system determined
under this contract had a 3100 Hz bandwidth and a 5 level fixed/frame
quantizer. Furthermore, it employed both a pitch loop predictor
and a coefficient predictor. The fixed/frame quantizer was used
because, at the design review, it was the only quantizer known that
was robust to channel errors. The 5 level quantizer was used
because it was the most straightforward approach to obtaining a
3000 Hz audio bandwidth and the pitch loop was necessary because
the speech quality degraded when it was removed.

Just as the contract ended, we started to investigate modified
Jayant quantizers and other quantizers which are also robust to
channel errors. Preliminary investigations indicate that these
quantizers provide improved quality over the fixed/frame quantizer,
and their use can eliminate the need for the pitch loop without
loss of quality. We recommend that these robust quantizers be studied
further, and that they be incorporated into PSP software for operation
in the DCA equipment. This will provide high quality adaptive
predictive coders using these quantizers. In addition, we recommend
that variable length coding coupled with storage buffers be used to obtain a 3000 Hz audio bandwidth with either a 7 or 8 level quantizer. This will improve the signal-to-noise ratio of the system while keeping the transmission rate at 16 Kb/s. These studies should simplify the implementation hardware from the system being delivered under this contract, while improving voice quality still further.
REFERENCES


APPENDIX A
MORE DETAILED FLOW CHARTS OF APCQ CODER SIMULATION

A.1 INTRODUCTION

Figures A-1 and A-2 contain more detailed flow charts of the fixed-point simulation with 8 level Jayant Quantizer. These characters indicate the scaling needed to process the speech with 16-bit arithmetic. This scaling is necessary since the summing of the data can cause the answer to overflow and exceed the maximum number of the machine ($2^{15}$). Additionally, scaling is needed to avoid underflow in multiplication since only 16 bits of a 32-bit product of two 16-bit numbers can be retained. It is common practice to retain the upper or most significant 16 bits, which is equivalent to dividing the 32-bit product by $2^{15}$. This operation is performed automatically by the hardware and is called a fractional multiply. Here the numbers are thought of as fractions having a magnitude less than 1, and the location of the binary point is not changed after multiplication. The flow charts indicate this fractional multiply by showing the division by $2^{15}$, which, as previously stated, is performed automatically by the hardware.

A.2 MAIN PROGRAM FLOW CHARTS AND LISTINGS

Figure A-1 presents a flow chart of the analyzer section of the APCQ Coder. In this portion of the program, the data is read off the tape, pre-emphasized, normalized, and processed to compute the pitch M, pitch gain ALPHA. This portion of the program also includes computations and quantizations of the residual error signal and the encoding of the quantized error signal. Additionally, the PARCOR parameters are calculated by this code.

Figure A-2 similarly describes the receiver portion of the code. Here the transmission parameters are decoded, the predictor coefficients recomputed from the PARCOR coefficient and the error signal used to excite the reconstruction filter. Prior to output on tape, the speech is de-normalized and de-emphasized.
Figure A-1. Flow Chart of Fixed-Point APCQ Analyzer (Sheet 1 of 3)
Figure A-1. Flow Chart of Fixed-Point APCQ Coder-Analyzer (Sheet 2 of 3)
Figure A-1. Flow Chart of Fixed-Point APCQ Coder-Analyzer
(Sheet 3 of 3)
Figure A-2. Flow Chart of Fixed-Point APCQ Synthesizer
Table A-1 contains a list of the variable names and their meanings as used in the main program of the APCQ Coder. Included in this Table is a list of the subroutines and arithmetic functions used as well as the data arrays needed to filter the speech data.

The FORTRAN listing of the main program is contained in Figure A-3 and includes comment statements indicating the main functions of the program.

**TABLE A-1. VARIABLE NAMES USED IN MAIN PROGRAM OF APCQ Coder**

<table>
<thead>
<tr>
<th>SUBROUTINES</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAPE1(I)</td>
<td>Magnetic tape handling where I indicates function:</td>
</tr>
<tr>
<td></td>
<td>I=1 read</td>
</tr>
<tr>
<td></td>
<td>I=2 write</td>
</tr>
<tr>
<td></td>
<td>I=3 rewind tapes</td>
</tr>
<tr>
<td></td>
<td>I=4 search for last file</td>
</tr>
<tr>
<td></td>
<td>I=5 write end of file</td>
</tr>
<tr>
<td></td>
<td>I=6 rewind only input-search for desired input file</td>
</tr>
<tr>
<td>SW1(I, ITOG)</td>
<td>Set ITOG=1 if console switch I is up, ITOG=0 if down</td>
</tr>
<tr>
<td>DISREG (I)</td>
<td>Display I on console</td>
</tr>
<tr>
<td>NORMLL(V,VV,L,N,NS)</td>
<td>Normalize N numbers in array V to N+1 bits &amp; place result in VV. NS=# shifts to do this</td>
</tr>
<tr>
<td>RANDIN(I,J,Y)</td>
<td>Random number generator</td>
</tr>
<tr>
<td>IALPQ (IALPHA)</td>
<td>Quantizes pitch gain to 3 bits</td>
</tr>
<tr>
<td>IALPD (IALPHA)</td>
<td>Dequantize IALPHA</td>
</tr>
<tr>
<td>EDQNT3(MIE,IE(J),LEVLT1)</td>
<td>Code error vector MIE into transmission vector IE(J)</td>
</tr>
<tr>
<td>RANERR(A1,A2,A3,A4,A5,A6)</td>
<td>Inserts random errors into data stream Al according to A2-A6</td>
</tr>
<tr>
<td>EDDNT3(IEREC(J),IER,LEVLR1)</td>
<td>Dequantize error IEREC(J) into IER, adjust level LEVLR1</td>
</tr>
<tr>
<td>IPARQ(IPARCT(J))</td>
<td>Quantize PARCOR (Reflection coefficient parameter IPAR CT(J))</td>
</tr>
<tr>
<td>IPAROD(IPARCT(J))</td>
<td>Dequantize PARCOR transmission parameter</td>
</tr>
</tbody>
</table>
PROGRAM IDACRC, FTN
THIS PROGRAM SIMULATES THE APCA SYSTEM
WITH FIXED POINT ARITHMETIC
AND WITH ADAPTIVE PREDICTION AND PITCH LOOP

COMMON/IFAP50,JIN(170),JOUT(170)
COMMON/IFAP,NS,IST,NTOI,NUPS,NTOT
COMMON/IFAP2,READ,NEP,NFILE
DIMENSION ISPIN(260)
DIMENSION IP(10),IT(150),JL(130),JL(10),ITAPIR(10)
DIMENSION IFAP(10),IFAPIR(10)
DIMENSION FC(260),Y(130)
DIMENSION TV(130),TV(250),IFR(260)
DIMENSION IRIN(10),IEEC(130),IESC(10),IY(10)

1010 CONTINUE

INITIALIZATION OF QUANTIZER

LEVLI=1
LEVLI=1
LF=1024
WRITE(6,1177)
F21S=2.015
FL21S=FL21-1.

1177 TYPE IN INFORMATION ABOUT FILE TO BE PROCESSED

FORMAT(1X,'FILE NO. ='/)
READ (6,101)INFILE

101 FORMAT(1X,'FILE NO. ='/)
MTOTI=120
NTOI=120
NUPS=NTOI
NTOI=NTOI
LTH=LTH1
LTH=LTH2
LTH=75
IST=1
JRN=0
IRN=0
MSHFT=0
LSHFT=0
LSWRC=0
CSN=0.

PREDICTOR ORDER

WRITE(6,99)
FORMAT(1X,'PREDICTOR ORDER ='/)
READ (6,101)
NN=NN+1

PROCESS FROM BEGINNING OF FILE

NSKIP=1
NSKIP=NSKIP

NUMBER OF PROCESSING FRAMES

IEEC=999
WRITE(6,105)
ASK IF WE WISH TO APPEND THIS DATA TO MAGNETIC TAPE
AND SAVE ALL PREVIOUS PROCESSING ON TAPE

105 FORMAT(1X,'1=APPEND OUTPUT, 0= BEG TAPE')/ READ (6,106) IN

106 FORMAT(I1)

ASK ABOUT THE PROBABILITY OF ERROR ON TRANSMISSION
PARAMETERS FOR THE ERROR SIGNAL, THE NORMALIZER
THE PARMCOR COEFFICIENTS, PITCH AND PITCH GAIN

200 FORMAT(' FRACTION OF ERRORS IN ERROR SIGNAL')/
READ(6,200) PROBA
WRITE(6,200)
1200 FORMAT(' FRACTION OF ERRORS IN THE OTHER PARAMETERS')/
READ(6,209) PROB
209 FORMAT(F10.0)

C IS THERE PITCH ON THIS RUN

WRITE(6,210)
210 FORMAT(1X,'1=PITCH, 0=NO PITCH')/
READ (6,106) IPTN
M=15
MIO=0

C INITIALIZE RANDOM NUMBER GENERATOR

DO 1216 J=1,49
1216 CALL RANDU (INJ,IRN,YVR)

C REWIND THE TAPE

CALL TAPE1(3)
IF(IN.EQ.0) GO TO 107

C ADVANCE THE TAPE TO THE END OF THE WRITTEN MATERIAL
IF PREVIOUSLY REQUESTED

CALL TAPE1 (4)
IF(NEAR.EQ.0) GO TO 107
WRITE(6,109)
109 FORMAT(1X,'FILE NOT FOUND')/
GO TO 1010

107 LTHX=LTH-2

C READ FRONT CONSOLE SWITCH REGISTER

CALL SII(15,ITOG)
COUNT=0
JCOUNT=0

C SW 15 CHANGES - CAUSES END OF RUN
C SW 13 UP - PRINT CHANNEL ERROR INFORMATION

1000 CALL SII(15,ITOG1)
CALL SII(13,ITOG)
IF(ITOG1.NE.ITOG) GO TO 5000
ICOUNT=ICOUNT+1
IF(ICOUNT.GT.IREC) GO TO 5000
C*
C DISPLAY NUMBER OF PROCESSING FRAME ON COMPUTER CONSOLE
C*
C******** CALL DISREG(ICOUNT)
C******** DATA INPUT
C******** CALL TAPE1(1)
IF(NEND.NE.0) GO TO 5000
DO 1001 J=1,LTH
1001 F(LTH+J)=MIN(J)
FN1=F(LTH2)
FN2=F(LTH2-1)
C*
C PREFERABLE TO SPEECH 6 DB. ABOVE 500 Hz.
DO 1101 J=1,LTHX
1101 F(I)=F(I-1)+0.4*F(I-2)
FLH(I)=F(LTH(I)-FLI*0.4*FL2
FL1=FN1
FL2=FN2
C*
C NORMALIZE PRESENT AND PAST ANALYSIS INTERVALS
C******** DO 1005 J=1,LTH2
1005 ISPIN(J)=F(J)
C********
C******** NORMALIZE PRESENT AND PAST FRAMES TO 12 BITS + SIGN
C******** CALL NORMAL1(ISPIN,ISPIN,LTH2,11,NSHFT)
C******** DO NOT SHIFT THE DATABASE MORE THAN 8 PLACES AND
C******** ALWAYS SHIFT IT AT LEAST 1 PLACE
IF(NSHFT.LE.8) GO TO 2600
NP=2**((NSHFT-8)
DO 2601 I=1,LTH2
2601 ISPIN(I)=ISPIN(I)/NP
NSHFT=8
C SET UP NSHFT FOR TRANSMISSION. NSHFT QUANTIZED TO 4 BITS.
2600 NSHFT+NSHFT=1
C********
C MULTIPLY DATA IN ANALYZER BY 2**(NSHFT-LSHFT)
C******** IF(NSHFT-LSHFT)1030,1031,1032
1030 NL=2**(LHFT-NSHFT)
DO 1108 I=1,N
1108 IRIN(I)=IRIN(I)/NL
DO 1007 I=1,LTH
1007 IRIN(I)=IRIN(I)/NL
1007 LEVT1=LEVT1/NL
IF(LEVT1.EQ.0) LEVT1=1
GO TO 1031
1032 NL=2**(NSHFT-LSHFT)
DO 1109 I=1,N
1109 IRIN(I)=IRIN(I)/NL
DO 1006 I=1,LTH
1006 IRIN(I)=IRIN(I)/NL
1006 LEVT1=LEVT1/NL
IF(LEVT1.NE.LP)LEVT1=LP
CALC OF AMDF AND SELECTION OF M

IF(PTN.EQ.0) GO TO 700
M=15
IRMIN=FL215D
DO 2 I=1,15,75
ISUM=0
DO 1 J=1,LTH+3
JJ=JTH+J
ISUM=ISUM+ABS(ISPIN(JI)-ISPIN(JJ-I))/4
IF(ISUM.GT.IRMIN) GOTO 2
M=1
IRMIN=ISUM
CONTINUE

SET UP M FOR TRANSMISSION, M QUANTIZED TO 6 BITS.
M=M-15

COMPUTES ALPHA

ISUM1=0
ISUM2=0
DO 40 J=1,LTH
JJ=LTH+J
ISP=ISPIN(JJ-M)
ISUM1=ISUM1+IMUL(ISPIN(JJ),ISP)
ISUM2=ISUM2+IMUL(ISP,ISP)
ALPHA=0
IF(ISUM2.EQ.0) GO TO 700
IF(ABS(ISUM1).GE.ABS(ISUM2)) GO TO 702
ALPHA=IFDIE(ISUM1,ISUM2)
GO TO 700
ALPHA=32400*(ISUM1/ABS(ISUM1))*(ISUM2/ABS(ISUM2))

CALL IALPD(IALPHA)
MALPHA=IALPHA
CALL IALPD(IALPHA)

CALL NORM1(IU,LTH,ISUM,LTHE)
DO 8 J=1,NN
ISUM=0
LTHE=LTH-1+1
DO 6 J=1,LTHE,
ITEM=0
JJ=J+7
DO 9 K=J,JJ
ITEM=ITEM+IU(K)*IU(K+1)/4
6
7
8
9
10

CALL NORM1(IU,LTH,ISUM,LTHE)
DO 8 J=1,NN
ISUM=0
LTHE=LTH-1+1
DO 6 J=1,LTHE,
ITEM=0
JJ=J+7
DO 9 K=J,JJ
ITEM=ITEM+IU(K)*IU(K+1)/4
6
7
8
9
10
7 IR(I)=IFDUI(IR(I),IR1)
C**
C*** CALCULATION OF REFLECTION COEFFICIENTS
C***
3110 DO 31 I=1,N
IAC(I)=IR2
IPARCT(I)=IR2
IU=FL215D
IU=IU+IFMUL(IAC(I),IR2)
IAC(I)=IU/IAC(I)/4
IF(N.LE.1) GO TO 31
DO 30 I=2,N
IU=IR(I+1)/2
IZ=I-1
DO 10 MM=1,IZ
IB(MM)=IAC(I-MM)
IU=IU+IFMUL(IB(MM),IR2)*2
IU=IU/2
IK=0
IF(IABS(IW).GE.IU) GO TO 31
IX=IFDIV(IW,IU)
IPARCT(I)=IK
10 DO 50 MM=1,IZ
IAC(MM)=IAC(MM)+IFMUL(IK,IB(MM))
IAC(I)=IK/4
IU=IU+IFMUL(IK,IW)
IF(U1ющее.0) GO TO 31
30 CONTINUE
C***
C*** QUANTIZATION OF THE REFLECTION COEFFICIENTS
C***
31 IF(N.EQ.0) GO TO 4802
DO 600 I=1,N
CALL IMPARO(IPARCT(I))
C***
600 MPARCT(I)=IPARCT(I)
C*** CALL IMPADD(IPARCT(I))
C***
C*** CALCULATE PREDICTOR COEFFICIENTS FROM PARCOR COEFFICIENTS
C***
IAC(I)=IPARCT(I)/4
DO 800 I=2,N
IM1=I-1
DO 110 J=1,IM1
IB(J)=IAC(J)-IFMUL(IPARCT(I),IAC(J-J))
DO 110 J=1,IM1
IAC(J)=IB(J)
110 IAC(I)=IPARCT(I)/4
C***
C*** CALCULATE ERROR SIGNAL
C***
4802 DO 3100 J=LTH
JJ=LTH+J
C***
C*** ERROR INDICATOR
C***
IGN=0
IFORF=0
IF(N.EQ.0) GO TO 4903
DO 3320 K=1,N
I*N*K+1
I*PIN(I+1)=IRIN1(I)
IFORP+IFORP+FMUL(JA(I),IRIN1(I))
IFORP=-IFORP+4
4803 IPITCH+FMUL(1ALPHA,IRIN2(J-J-1))
MIE=IPIN(JJ)-IFORP-IPITCH
C***
C*** QUANTIZE ERROR SIGNAL AND CODE ERROR SIGNAL
C***
CALL EDGNT3(MIE,IE(J),LEULT1)
IRIN1(I)=IFORP+MIE
3100 IRIN(JJ)=IPITCH+IRIN1(I)
C***
C*** TRANSFER THE QUANTIZED ERROR SIGNAL
C***
DO 7540 I=1,LTH
7540 IECEC(I)=IE(I)
C***
C*** TRANSFER QUANTIZER PARCOR
C***
IF(N,EQ,0) GO TO 4804
DO 7550 I=1,N
7550 IFARCR(I)=IFARCT(I)
C***
C*** INTRODUCE THE CHANNEL ERRORS
C***
C*** RANDOM NO. IS LESS THAN OR EQUAL TO PROB, AN ERROR IS INTRODUCED
C*** AT THAT BIT.
C***
4804 KOUNT=0
IF(IPITN.EQ,0) GO TO 4806
IF(PROB.EQ,0.) GO TO 7706
DO 7701 J=1,6
7701 CALL PANERR(MMG,J,PROB,IRN,IRN,KOUNT)
DO 7703 J=1,4
7703 CALL PANERR(MALPHA,J,PROB,IRN,IRN,KOUNT)
4806 IF(N,EQ,0) GO TO 4805
IFL=4
DO 7704 I=1,N
7704 DO 7705 J=1,4
7705 CALL PANERR(IPARCR(I),J,PROB,IRN,IRN,KOUNT)
4805 DO 7706 J=1,3
7706 CALL PANERR(MSHFT,J,PROB,IRN,IRN,KOUNT)
7706 IF(PROB.EQ,0.) GO TO 7710
DO 7707 I=1,LTH
DO 7707 J=1,3
7707 CALL PANERR(IECEC(I),J,PROB,IRN,IRN,KOUNT)
WRITE(5,7711) KOUNT,ICOUNT
7711 FORMAT(* ERRORS*,25)
C***
C*** RECEIVE
7710 MPITN=MINO+15
MRSHFT=MSHFT+1
IRALPHA=MALPHA
CALL RALPD(IPALPH)
IF(N,EQ,0) GO TO 4807
DO 7220 J=1,N
7220 CALL IPAROD(IPARCR(J))
C***
C*****
SYNTHESIZE OUTPUT

MULTIPLY PREVIOUS FRAME BY 2***(NRSHIFT-LSHRC)

IACNT=0
IF(NRSHIFT-LSHRC) 5065,5067,5067
NL=2***(LSHRC-NRSHIFT)
LEVLR1=LEVL1/N
IF(LEVLR1.EQ.0) LEVL1=1
DO 5168 I=1,N
5168 IV1(I)=IV1(I)/NL
DO 5068 I=1,LTH
5068 IV2(I)=IV2(I)/NL
GO TO 5070
5067 NL=2***(NRSHIFT-LSHRC)
DO 5169 I=1,N
5169 IV1(I)=IV1(I)*MUL(NL1,NL1,NL1,NACN)
DO 5069 I=1,LTH
5069 IV2(I)=IV2(I)*MUL(NL1,NL1,NL1,NACN)
LEVL1=1*MUL(LEVL1,NL1,NL1,NACN)
5070 LEVL1=LEVL1,LEVL1=LP
LSHRC=NRSHIFT

RECOMPUTE THE PREDICTOR COEFFICIENTS

IF(N.EQ.0) GO TO 7950
IA1=1*IPARC1/4
IF(N.EQ.1) GO TO 7950
DO 7810 I=2,N
7810 IM=1-1
DO 7820 J=1,IM1
7820 IB(J)=IA(J)-1*MUL(IPARC1,IA(I-J))
GO TO 7830 J=1,IM1
7830 IA(J)=IB(J)
7810 IA(I)=1*IPARC1/4

FILTER THE ERROR SIGNAL

DO 401 J=1,LTH
7850 JJ=LTH+J

DEQUANTIZE ERROR SAMPLE

CALL EDINT3(IEREC(J),IER,LEVL1)
ISUM=0
IF(N.EQ.0) GO TO 4810
DO 400 K=1,N
I=N-K+1
IV1(I)=IV1(I)
IV2(I)=IV2(I)
400 ISUM=ISUM,ITEMP,IA(I)
4810 IV1(I)=IV1(I)+ISUM,ITEMP,IA(I)
401 IV2(I)=IV2(I)+ISUM,ITEMP,IA(I)

END OF SYNTHESIS

MULTIPLY OUTPUT BY 2***(NRSHIFT)
XNL=2.0*(!HSHT)
DO 4010 J=1,LTH
4010 YY(J)=YY(J)*XNL
C**** COMPUTE S/N RATIO
      SUM1=0.0
      SUM2=0.0
      DO 456 I=1,LTH
      FX=F(LTH+I)
      SUM1=SUM1+FX-YY(I)**2
      SUM2=SUM2+FX**2
      SN=10.*ALOG10(SUM2/SUM1)
      CSN=(ICOUNT/(1.+ICOUNT))*CSN+SN/(1.+ICOUNT)
      WRITE(S,7306)ICOUNT,SN,CSN
      IF(ICT,EQ,0) GO TO 7305
      WRITE(S,7305)
      CONTINUE
      IF (J06.EQ.0) GO TO 7300
      WRITE OUT PARAMETERS IF SWITCH 13 IS UP.
      WRITE(S,7300)ICOUNT
      WRITE(S,7301)M,TALPHA,NSHFT,(IPARCT(I),I=1,N),
      (IE(I),I=1,LTH)
      WRITE(S,7302)M,MRPIT,IRALPH,NSHFT,
      (IPARCR(I),I=1,N),I'REC(I),I=1,LTH)
      WRITE(S,7313)
      CONTINUE
7301 FORMAT(1X,'ORIGINAL TRANSMISSION PARAMETERS',15)
7303 FORMAT(1X,'QUANTIZED PARAMETERS')
7305 FORMAT(1X,'QUANTIZED PARAMETERS WITH ERRORS')
7307 FORMAT(1X,'PARAMETERS RECEIVED')
7311 FORMAT(3X,'M=',15.4X,'TALPHA=',16.4X,
1 'S/NSHFT=',13.3X,'IPARCT',4(3X,17)/
2 3X,15.1E-916/12(10.5)/)
7312 FORMAT(3X,'MRPIT=',13.4X,'IRALPH=',16.4X,
1 'NRSHFT=',13.3X,'IPARCR',4(3X,17)/
2 3X,15.1E-916/12(10.5)/)
7313 FORMAT(//)
C**** SPECTRAL DEPHASIS
C**** Y(1)=YY(1)+RL1-0.4*RL2
C**** Y(2)=YY(2)+Y(1)-0.4*RL1
DO 1500 J=3,LTH
1500 Y(J)=YY(J)+Y(J-1)-0.4*Y(J-2)
RL1=Y(LTH)
RL2=Y(LTH-1)
C**** WRITE OUT THE DATA
```fortran
402 DO 403 J=1,LTH
    NOUT(J)=Y(J)
    CALL TAPE1(2)

C-----
C THIS Shifts the DATA for NEXT TIME
C-----
403 DO 403 J=1,LTH
    JJ=LTH+J
    F(J)=F(JJ)
    IRINC(J)=IRINC(JJ)
    IV2(J)=IV2(JJ)
    GO TO 1000

C-----
C WRITES THE END OF FILE
C-----
5000 DO 5001 I=1,LTH
5001 NOUT(I)=0
5002 DO 5002 I=1,32
5002 CALL TAPE1(2)
5002 NSKIP=NSKIPS
5002 NEND=0
5002 IST=1
5002 CALL TAPE1(6)
5002 DO 5005 I=1,1COUNT
5005 CALL TAPE1(1)
5005 IF(NEND.NE.0) GO TO 5077
5005 DO 5006 J=1,LTH
5006 NOUT(J)=NINC(J)
5006 CALL TAPE1(2)
5006 DO 5007 I=1,LTH
5007 NOUT(I)=0
5007 DO 5008 I=1,8
5008 CALL TAPE1(2)
5008 CALL TAPE1(5)
5008 WRITE(5,5009)
5009 FORMAT(1X,'DONE')
5009 END
```

Figure A-3: FORTRAN Listing of Main Program of APCQ Coder
| TABLE A-1. VARIABLE NAMES USED IN MAIN PROGRAM OF APCQ CODER (Cont.) |
| --- | --- |
| **SUBROUTINES** | **MEANING** |
| **FUNCTIONS** |  |
| IADD (I1,I2,IOVR) | Add I1 to I2, increment IOVR if overflow |
| IFMUL (I1,I2) | Fractionally multiply I1, I2; round result |
| IFDIV (I1,I2) | Fractionally divide I1 by I2 |
| IIMUL (I1,I2,IOVR) | Integer multiply I1 by I2; increment IOVR if overflow |
| **VARIABLES** |  |
| LEVLTI, LEVLR1 | Error quantizer level at transmitter and receiver |
| LP | Maximum value of quantizer |
| IR1N1, IY1 | Coefficient predictor filter memory and transmitter receiver |
| NTOTI | Tape handler data |
| NTUPS | Tape handler data |
| NTOTO | Tape handler data |
| NSKIP | Tape handler data |
| IST | Tape handler data |
| NSHFT, NRSHFT | Present normalizer shift |
| LSHFT, LSHRC | Past frame normalizer shift |
| NIN | Speech input array |
| F | Input array after pre-emphasis |
| ISPIN | Input array after normalization |
| IV | Reduced waveform |
| IRIN2, IY2 | Pitch filter history (reconstructed waveform) |
| IA | Predictor coefficients |
| IR | Autocorrelation values |
| M | Pitch delay |
| IALPHA, IRALPH | Pitch Gain Alpha |
| MALPHA | Quantized Alpha |
### TABLE A-2. NUMBER OF TRANSMISSION BITS FOR EACH PARAMETER

<table>
<thead>
<tr>
<th>Parameter</th>
<th>No. of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>pitch gain</td>
<td>3/frame</td>
</tr>
<tr>
<td>pitch</td>
<td>6/frame</td>
</tr>
<tr>
<td>normalization</td>
<td>3/frame</td>
</tr>
<tr>
<td>PARCOR</td>
<td>12/frame</td>
</tr>
<tr>
<td>error</td>
<td>2.1/sample</td>
</tr>
</tbody>
</table>

The quantization and dequantization of the pitch gain ALPHA is performed by the subroutines IALPQ and IALPD respectively. Quantization and dequantization of the PARCOR coefficients is performed by subroutines IPARQ and IPARDD respectively, with each PARCOR quantized to 4 bits. Quantization of the pitch is performed simply by subtracting 15 from the calculated value and quantization of the normalization by subtracting 1 from the unquantized value. The quantized values then run from 0 - 60 for pitch (6 bits) and 0 - 7 (3 bits) for normalization. These numbers represent coded values for transmission.
A.3 QUANTIZATION OF THE ERROR SIGNAL

Quantization of the error signal is accomplished by means of an eight-level adaptive quantizer as described previously in Section 2.8.2. This quantizer technique was developed by Jayant as described in the IEEE Proceedings of May 1974.

Figure A-4 shows a flow chart of the subroutine EDQNT-3 that is used to implement the adaptive quantization process. This routine converts the error signal into one of eight numerical values, and adjusts the quantizer step size in accordance with the "Jayant algorithm".

Table A-3 shows the variable names and their meanings as used in subroutine EDQNT-3. Figure A-5 is a FORTRAN listing of this subroutine.

### TABLE A-3. VARIABLE NAMES USED IN SUBROUTINE EDQNT3

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>Maximum value of quantizer level</td>
</tr>
<tr>
<td>MIE</td>
<td>Input unquantized error and output quantized value</td>
</tr>
<tr>
<td>LEVEL</td>
<td>Present quantizer level</td>
</tr>
</tbody>
</table>
Figure A-4. Flowchart of Subroutine EDQNT3 - Quantizing Error Signal
(8-Level Jayant Quantizer)
SUBROUTINE EDQNT(IN, IGNT, LEVEL)
ERROR QUANTIZER AT THE TRANSMITTER
EIGHT LEVEL ADAPTIVE QUANTIZER
WITH COEFFICIENTS AS PER JAYANT
DIMENSION COEF(4)
DATA LP: 1.024
DATA COEF/0.5, 0.9, 1.25, 1.75/
SIGN=1
IF(IN.LT.0) GO TO 10
IN=-IN
SIGN=-1
10 IF(IN.LE.LEVEL) GO TO 1
IF(IN.LE.2*LEVEL) GO TO 2
IF(IN.LE.3*LEVEL) GO TO 3
TOP MOST QUANTIZER LEVEL
IGN=0
IF(SIGN.EQ.1) IGN=7
IN=3.5 LEVEL*SIGN
LEVEL=LEVEL*COEF(4)
IF(LEVEL.GT.LP) LEVEL=L
RETURN
C SECOND FROM TOP LEVEL
IGN=1
IF(SIGN.EQ.1) IGN=6
IN=2.5 LEVEL*SIGN
LEVEL=LEVEL*COEF(3)
IF(LEVEL.GT.LP) LEVEL=L
RETURN
C SECOND LEVEL FROM THE BOTTOM
IGN=2
IF(SIGN.EQ.1) IGN=5
IN=1.5 LEVEL*SIGN
LEVEL=LEVEL*COEF(2)
IF(LEVEL.EQ.0) LEVEL=1
RETURN
C BOTTOM LEVEL
IGN=3
IF(SIGN.EQ.1) IGN=4
IN=LEVEL/2*SIGN
LEVEL=LEVEL*COEF(1)
IF(LEVEL.EQ.0) LEVEL=1
RETURN
END

Figure A-5: FORTRAN Listing of Error Signal Quantizer
A.4 Dequantization of the Error Signal

The flow chart of Figure A-6 shows the dequantization of the error signal via subroutine EDDNT3. This algorithm performs the inverse operations to those performed by the quantizer at the analyzer and yields as output the sequence of quantizer error levels generated in the analyzer.

Table A-4 contains the variable names and their meanings used in this subroutine. A FORTRAN listing of this subroutine is contained in Figure A-7.

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>Maximum value of quantizer level</td>
</tr>
<tr>
<td>MIE</td>
<td>Input quantized value</td>
</tr>
<tr>
<td>ISIG</td>
<td>Output dequantized value</td>
</tr>
</tbody>
</table>

A.5 Other Subroutines

A.5.1 Introduction

This section presents various subroutines that have not been discussed previously. Table A-5 presents a list of the names of these subroutines and their functions.
Figure A-6. Flowchart of Subroutine EDDNT3 - Dequantizing Error Signal (8-Level Jayant Quantizer)
SUBROUTINE EDINT3(IN, IOUT, LEVEL)
SUBROUTINE FOR DEQUANTIZING 3 BIT ADAPTIVE
QUANTIZER SIGNAL AT THE RECEIVER
FOLLOWS THE JAYANT ALGORITHM

DIMENSION COEF(4)
DATA COEF/1.75, 1.25, 0.9, 0.9/
DATA LP/10.24/
I.IN=1
IF(IN.LT.1.OR.IN.GT.8) IIN=1
IF(IN.LE.4) GO TO 10
I.ISIGN=1
I.IN=9-IIN
GO TO (1,2,3,4)IIN

1 IOUT=3.5*I.LEVEL*I.ISIGN
LEVEL=LEVEL*COEF(InII)
IF(LEVEL.LT(LP) LEVEL=LP
RETURN
2 IOUT=2.5*I.LEVEL*I.ISIGN
GO TO 11
3 IOUT=1.5*I.LEVEL*I.ISIGN
LEVEL=LEVEL*COEF(InII)
IF(LEVEL.EQ.0) LEVEL=1
RETURN
4 IOUT=0.5*I.LEVEL*I.ISIGN
GO TO 33
END

Figure A-7: FORTRAN Listing of Error
Signal Dequantizer

COPY AVAILABLE TO BBC DOES NOT
PERMIT FULLY LEGIBLE PRODUCTION
TABLE A-5. SUBROUTINES AND THEIR FUNCTIONS

<table>
<thead>
<tr>
<th>SUBROUTINE NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IALPQ</td>
<td>Quantize Alpha To 3 Bits</td>
</tr>
<tr>
<td>IALPD</td>
<td>Dequantize Alpha</td>
</tr>
<tr>
<td>IPARQ</td>
<td>Quantize PARCOR coefficient</td>
</tr>
<tr>
<td>IPARDD</td>
<td>Dequantize PARCOR coefficient</td>
</tr>
<tr>
<td>RANERR</td>
<td>Generate and Introduce Random Errors</td>
</tr>
<tr>
<td>NORMLI</td>
<td>Normalize Input Speech</td>
</tr>
<tr>
<td>IADD</td>
<td>Fixed-Point Addition With Overflow Indication</td>
</tr>
<tr>
<td>ISUB</td>
<td>Fixed-Point Subtraction With Overflow Indication</td>
</tr>
<tr>
<td>IFMUL</td>
<td>Fixed-Point Fraction Multiply With Rounding</td>
</tr>
<tr>
<td>ITMUL</td>
<td>Fixed-Point Integer Multiply With Overflow Indication</td>
</tr>
<tr>
<td>IFDIV</td>
<td>Fixed-Point Fractional Divide</td>
</tr>
</tbody>
</table>

A.5.2 Subroutines IALPQ and IALPD

Subroutines IALPQ and IALPD each contain one argument. In IALPQ the argument upon entry is the unquantized ALPHA. The subroutine converts this argument to a three-bit representation having values 0 through 7. These values are in a form suitable for transmission. Subroutine IALPD accepts the corresponding received three-bit value and converts it into one of eight possible values for ALPHA by a table look-up procedure.

Figure A-8 contains a FORTRAN listing of both IALPQ and IALPD.
Figure A-8: FORTRAN Listing of Subroutines IALPQ and IALPD
A.5.3 Subroutine IPARQ and IPARDD

Subroutines IPARQ and IPARDD quantize and dequantize the PARCOR coefficients. Each subroutine takes one agreement and returns the answer in the same variable. In IPARQ the argument upon entry is the \( i^{th} \) unquantized PARCOR(1). The subroutine converts this in a 4 bit representation having values 0 through 15. These values are in a form suitable for transmission. Subroutine IPARDD accepts the corresponding 4 bit value and converts it into one of 16 possible values for the \( i^{th} \) PARCOR coefficient. Figure A-9 contains a FORTRAN listing of both IPARQ and IPARDD.
SUBROUTINE IPARQ(J)
J=J END
SUBROUTINE IPARDD(J)
DIMENSION IPD(16)
DATA IPD/1200,3100,5075,7025,9325,11300,13350,
1 15225,17500,19475,21450,23400,25700,27075,29550,31500/
J=J+1
IFLAG=0
IF(J.LE.16) GO TO 1
IFLAG=1
J=J-16
J=IPD(J)
IF(IFLAG.EQ.1) J=-J
RETURN
END

Figure A-9: FORTRAN Listing of PARCOR Quantizer (IPARQ) and of PARCOR Dequantizer (IPARDD)

COPY AVAILABLE TO DDC DOES NOT
PERMIT FULLY LEGIBLE PRODUCTION
A.5.4 Subroutines RANERR and IXOR

Subroutine RANERR is entered with six arguments for introducing channel errors into the transmission parameters. These arguments include the transmission data, the probability of channel bit error, states for the random number generator, and a count of the total number of errors. The assembly language program IXOR is used in conjunction with RANERR to exclusively "OR" the transmission data with the random bit error data generator within RANERR. Documentation of subroutines RANERR and IXOR are contained in Table A-6 and Figure A-10. Table A-6 lists the variable names contained in subroutine RANERR and their meanings as used in this program. Figure A-10 contains listing of the FORTRAN subroutine RANERR and the assembly language subroutine IXOR.

TABLE A-6. VARIABLE NAMES USED IN SUBROUTINE RANERR

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK</td>
<td>Mask for introducing a bit of error in either first through 8 bit of word</td>
</tr>
<tr>
<td>RANDU</td>
<td>Subroutine for obtaining random number YRN between 0 and 1</td>
</tr>
<tr>
<td>PROB</td>
<td>Probability of error any bit</td>
</tr>
<tr>
<td>IXOR</td>
<td>Function for exclusive OR in bit of error</td>
</tr>
<tr>
<td>IWORD</td>
<td>Output word with errors introduced</td>
</tr>
</tbody>
</table>
SUBROUTINE RANERR (IWORD, KEYM, PROB, IRN, JRN, KOUNT)

C IWORD - THE WORD CONTAINING THE QUANTIZED PARAMETER
C KEYM - WHICH BIT OF IWORD THE SUBROUTINE WILL OPERATE ON
C PROB - THE FRACTION OF ERRORS
C IRN AND JRN - NOS. USED BY THE RANDOM NO. GENERATOR
C KOUNT - A RUNNING TOTAL OF THE ERRORS IN THIS FRAME

C A RANDOM NUMBER BETWEEN 0 AND 1 IS GENERATED AND IF IT
C IS LESS THAN PROB, IWORD IS EXCLUSIVE OR-ED WITH THE
C APPROPRIATE MASK IN ORDER TO FLIP THE BIT POINTED TO BY KEYM.
C I.E. IF A PARAMETER IS QUANTIZED TO 4 BITS, RANERR
C IS CALLED WITH KEYM EQUAL TO 1, 2, 3, AND 4.

C DIMENSION MASK(8)
C DATA MASK /1,2,4,8,16,32,64,128/
C CALL RANDU(INN, JRN, VMN)
C IF (VMN-PROB) 200, 500, 500
C 200 JUNK=1XOR(IWORD, MASK(KEYM))
C IWORD=JUNK
C KOUNT=KOUNT+1
C 500 RETURN
C END

.TITLE IXOR
.GLOBAL IXOR

/* THIS ASSEMBLY LANGUAGE SUBROUTINE TAKES EXCLUSIVE OR OF
TWO INPUT ARGUMENT AND LEAVES ANSWER IN REGISTER 0 FOR A FORTRAN
FUNCTION CALL
R0=20
R1=21
R5=35
I
IXOR: TST (R5)+
MOV @ (R5)+, R0
MOV @ (R5)+, R1
XOR R1, R0
RTS R5
.END

Figure A-10: FORTRAN Listing of Subroutine RANERR and
Assembly Listing of IXOR
A.5.5 Subroutine NORM1

Subroutine NORM1 contains five arguments, the unnormalized input array, the normalized output array, the number of data points in the array, the value of the largest point after normalization, and the number of shifts computed by the subroutine. An assembly language listing of this subroutine is contained in Figure A-11.

A.5.6 Function IADD, ISUB, IFMUL, IIMUL and IFDIV

The routines discussed in this section are called "Functions" since they return an argument to the main program via a specified register rather than one of the calling arguments. Functions IADD and ISUB arguments each contain three. The first two are the input numbers to be added (or subtracted) while the third argument is incremented if an overflow occurs. The function IFMUL has two arguments and returns the upper sixteen bits of the product (with rounding) of these two arguments to the calling program. This function is sometimes referred to as a fractional multiply because it preserves the location of the binary point.

The function IIMUL has three arguments which are the two input numbers to be multiplied and the overflow indicator. This function returns the lower 16 bits of the 32-bit product and is referred to as an integer multiply.

The function IFDIV accepts two inputs and returns the fractional part of the division to the calling program.

Assembly listing of these fixed point arithmetic functions are given in Figure A-12.
TITLE NORML.X.FIL

ROUTINE NORML X: RCURLY SHIFTING TO THE LEFT N ARAY OF INTEGER NUMBERS OBTAINED FROM A FORTRAN ARRAY AND PLACES THE RESULT IN A NEW ARRAY WITH ONE WORD PER POINT INSTEAD OF THE TWO WORDS FOLLOWED IN FORTRAN.

CALL NORML (IN,OUT,NSPEC,NS)

IN=FORTRAN INPUT ARRAY
OUT=OUTPUT ARRAY (ONE WORD INTEGER)
NS=N# OF WORDS IN ARRAY
NSPEC=SPECI FICATION OF SHIFTED OUTPUT
NS=N# OF SHIFT COMPUTED BY NORM

RO=0
R1=0
R2=0
R3=0
R4=0
R5=0

NORML:

TST FR+1
MOU R1,R5
MOU R1,R3
MOU R3,R5
MOU R4,R8
MOU R5+R5,R0
GET ADDRESS OF INPUT
MOU R5+R5,R1
SAVE INPUT ADDRESS
MOU R5+R1,R5
OUTPUT ADDRESS
MOU R5+R2,R5
N# OF 16TH WORDS
MOU R5+R4,R5
SAME # OF WORDS IN ARRAY
MOU #R5+R3
SAVE LOCATION OF SHIF TED VALUE
MOU R3,NSPEC

CLF R4
INCLUSIVE OR

LOOP:
MOU R5+R3
GET INPUT POINT
BPL PLUS
NBS R3
CAN'T BE NEG #S

PLUS:
BIS FR+R4
1# CONTAINING SUM OF INCLUSIVE OR
TST FR+R4
TWO WORD INTEGER
SUB R5-LOOP
CLF R3
SET UP TO COUNT # OF SHIFTS
TST R4
NEW ALL

LOOP1:
INC R4
INCREMENT COUNT
BAC LOOP1
1# OF LEFT SHIFTS REQUIRED

ALL:
MOU R4,R2
NORMALIZE INPUT
MOU #R5,R0
SPECIFY LOC. OF DEC. POINT
SUB NSPEC,R0
SUB R4,R2
MOU IN#R5,R0

LOOP2:
MOU R5+R4
GET INPUT
MOU R5+R4
END WORD INTEGER
ASH R2,R4
TST R4+R1
DEPOSIT INTO OUTPUT
SUB R5-LOOP2
TST R4+R1
RETURN SHIFT COUNT
MOU R5,R0
MOU R5,R0
MOU R5,R0
MOU R5,R0
MOU R5,R0

RFS R5

BUFFER STOR=16
FLS: JUMP 0
RIS: JUMP 0
PFS: JUMP 0
FBS: JUMP 0

N: WORK= LENGTH OF INPUT ARRAY
NSPEC: WORK= # SPEC. OF SHIFTED OUTPUT

END

Figure A-11: Assembly Language Listing of NORML
.TITLE ARITH.MAC

; THIS FUNCTION ADDS TWO NUMBERS AND INCREMENTS THE
; THIRD ARGUMENT IF THERE IS AN OVERFLOW
; SAME FOR THE SUBTRACT CASE

RO:=RO
RS:=1
GLOBAL IADD,ISUB,IMUL,IFDIV,ITMUL
I=IADD(+J)
K=K+1 IF OVERFLOW

IADD:
TST (RS)+
MOU (RS)+,RO
ADD @ (RS)+,RO
BUS A
TST (RS)+
RTS RS
I

ISUB:
TST (RS)+
MOU (RS)+,RO
SUB @ (RS)+,RO
BUS A
TST (RS)+
RTS RS
I

; THIS SUBROUTINE MULTIPLIES TWO NUMBERS IN FRACTIONAL FORM WITH Rounding
; AS IN THE GTE SYLVANIA PSP INSTRUCTION MLY

IMUL:
TST (RS)+
MOU (RS)+,RO
MUL @ (RS)+,RO
ASHC #1,RO
TST R1
BGE NEND
INC RO
RTS RS
NEND:
RTS RS

FUNCTION IFDIV(DIVIDEND,DIVISOR)

IFDIV:
TST (RS)+
MOU (RS)+,RO
ADJ FOR DIVIDE
DIV @ (RS)+,RO
RTS RS

FUNCTION IMUL

IMUL(A,B,C)

ANSH=ARB WITH C INCREMENTED IF OVERFLOW

IMUL:
TST (RS)+
MOU (RS)+,RO
MUL @ (RS)+,RO
MOU R1,RO
MOU INTEGER PART TO RO
BCS A
TST (RS)+
RTS RS

; END

Figure A-12: Assembly Listing of Fixed-Point Arithmetic Functions