

# Table of Contents

<b>SECTION 1</b> <b>Introduction</b>	1.1 Standard Version of ADPCM	1-1
	1.2 Non-Standard Version of ADPCM	1-2
<b>SECTION 2</b> <b>Speech Coding</b>	2.1 Trade-Offs	2-1
	2.2 Advantages of Digital Speech Coding	2-2
<b>SECTION 3</b> <b>Types of Speech Coders</b>	3.1 Uniform PCM	3-2
	3.2 Logarithmic PCM	3-5
	3.3 ADPCM Coding	3-7
<b>SECTION 4</b> <b>The CCITT ADPCM Algorithm</b>	4.1 The Encoder Algorithm	4-4
	4.2 The CCITT Decoder Algorithm	4-18
<b>SECTION 5</b> <b>ADPCM Implementation on the DSP56001</b>	5.1 I/O Interface	5-3
	5.2 Standard Implementation	5-5
	5.2.1 Code Structure	5-6
	5.2.2 Initialization	5-9
	5.2.3 PCM Format Conversion	5-12
	5.2.4 Logarithmic Conversion	5-14
	5.2.5 Floating-Point Conversion	5-17
	5.2.6 Difference Signal Quantization	5-19
	5.2.7 Inverse Quantization	5-22
	5.2.8 Adaptive Predictor	5-24



# Table of Contents

5.2.9	Tone Detection	5-31
5.2.10	Scale Factor Adaptation	5-32
5.2.11	Decoder Synchronization	5-34
5.3	Non-Standard Implementation	5-34
5.3.1	Code Structure	5-36
5.3.2	Initialization	5-37
5.3.3	Format Conversions	5-40
5.3.4	Adaptive Predictor	5-41
5.4	Optimization Techniques	5-43
5.5	Performance Specifications	5-52
Appendix A	Terminology	A-1
References		Reference-1

# Illustrations

<b>Figure 1-1</b>	CCITT ADPCM Encoder Block Diagram	1-2
<b>Figure 1-2</b>	CCITT ADPCM Decoder Block Diagram	1-3
<b>Figure 3-1</b>	A/D Conversion Process	3-2
<b>Figure 3-2</b>	Quantization Noise Model	3-3
<b>Figure 3-3</b>	Uniform Quantizer	3-4
<b>Figure 3-4</b>	Feedforward APCM Coder	3-8
<b>Figure 3-5</b>	Feedback APCM Coder	3-8
<b>Figure 3-6</b>	DPCM Coder	3-9
<b>Figure 3-7</b>	ADPCM Coder	3-11
<b>Figure 4-1</b>	CCITT ADPCM Encoder Block Diagram (detailed)	4-3
<b>Figure 4-2</b>	PCM Conversion and Difference Signal Computation	4-5
<b>Figure 4-3</b>	Adaptive Quantizer	4-6
<b>Figure 4-4</b>	Inverse Adaptive Quantize	4-7
<b>Figure 4-5</b>	Adaptive Prediction Filter	4-8
<b>Figure 4-6</b>	Predictor Pole Coefficient Adaptation for $a_1(k)$	4-10
<b>Figure 4-7</b>	Predictor Pole Coefficient Adaptation for $a_2(k)$	4-11
<b>Figure 4-8</b>	Predictor Zero Coefficient Adaptation	4-12

# Illustrations

<b>Figure 4-9</b>	Scale Factor Adaptation	4-13
<b>Figure 4-10</b>	Speed Control Parameter Adaptation	4-15
<b>Figure 4-11</b>	Tone Detection	4-17
<b>Figure 4-12</b>	CCITT ADPCM Decoder Block Diagram (detailed)	4-19
<b>Figure 4-13</b>	Synchronous Coding Adjustment	4-20
<b>Figure 5-1</b>	Code Flow Diagram	5-8
<b>Figure 5-2</b>	Internal Data RAM Memory Map	5-11
<b>Figure 5-3</b>	Address Register Usage	5-12
<b>Figure 5-4</b>	Linear to Log Conversion Routine	5-16
<b>Figure 5-5</b>	Linear to Floating-Point Conversion Routine	5-18
<b>Figure 5-6</b>	Difference Signal Scaling and Quantization	5-20
<b>Figure 5-7</b>	Inverse Quantization and Scaling of ADPCM Codeword	5-23
<b>Figure 5-8</b>	Adaptive Predictor Data Structure	5-27
<b>Figure 5-9</b>	Internal Data RAM Memory Map (Non-standard)	5-39
<b>Figure 5-10</b>	Address Register Usage (Non-standard)	5-39
<b>Figure 5-11</b>	Adaptive Prediction Filter	5-42
<b>Figure 5-12</b>	Adaptive Predictor Data Structure (Non-standard)	5-43



# List of Tables

<b>Table 4-1</b>	Quantizer Normalized Input/Output Characteristic	4-6
<b>Table 4-2</b>	W(l) Lookup Table	4-14
<b>Table 4-3</b>	F[l(k)] Lookup Table	4-16
<b>Table 5-1</b>	Memory Usage	5-53
<b>Table 5-2</b>	Code Execution Times	5-55

## SECTION 1

# Introduction

***“This application report will first point out some of the advantages of speech coding in general and then some of the particular advantages of the CCITT standard.”***

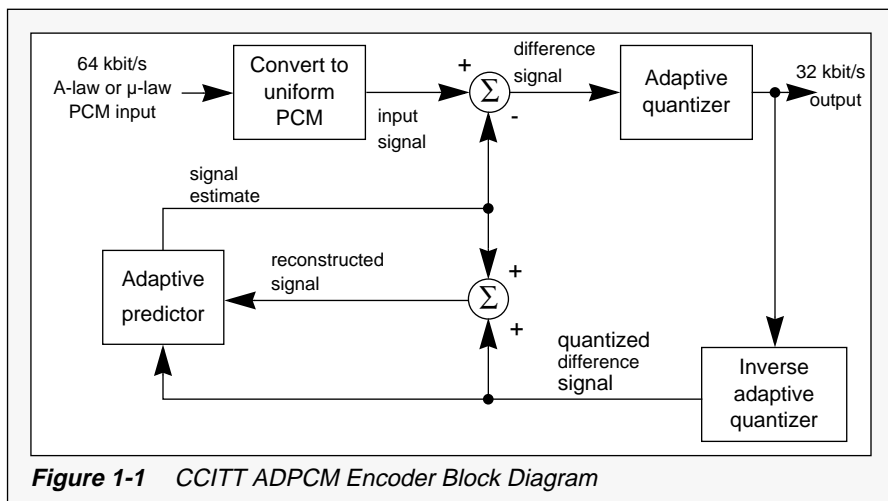
---

This application report describes the implementation of an Adaptive Differential Pulse Code Modulation (ADPCM) speech coder on the Motorola DSP56001 digital signal processor. The algorithm described in this document has been standardized by the International Telegraph and Telephone Consultative Committee (CCITT) in Recommendation G.721 [1] for digital speech coding in a telecommunications environment. The standard, as defined by the CCITT, specifies the translation of  $\mu$ -law or A-law PCM encoded speech at 64 kbit/s to ADPCM encoded speech at 32 kbits to provide a 2 to 1 compression of the speech signal with very little perceptual loss of speech quality. The algorithm also has added complexity to handle non-speech signals such as modem signals. The block diagrams of the CCITT ADPCM encoder and decoder are shown in Figure 1-1 and Figure 1-2.

## 1.1 Standard Version of ADPCM

Two implementations of the ADPCM algorithm on the DSP56001 are described in this document. The first implementation adheres completely with the CCITT Recommendation G.721 (revised version dated August 1986). It provides bit-for-bit compatibility with the

test vectors described in G.721, meaning that the code correctly passes all the digital test sequences defined by the CCITT. In addition to providing compatibility with the standard, this implementation provides full-duplex operation. This means that one DSP56001 is able to perform both an encode and a decode in real-time.

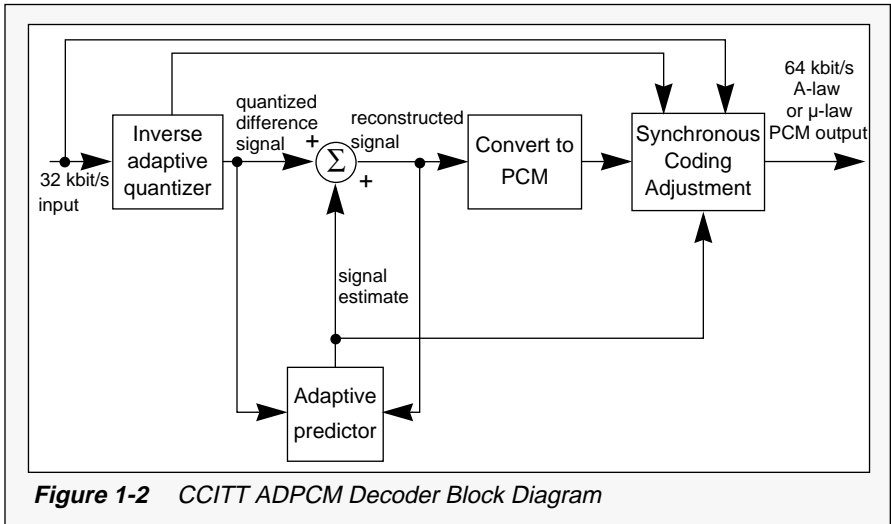


## 1.2 Non-Standard Version of ADPCM

The second implementation also provides full-duplex ADPCM operation on a single DSP56001. This non-standard version implements the same algorithm as the standard version but does so more efficiently. As a result, this version requires

less computational power and uses less memory than the standard version. In addition, the code provides a more direct, readable implementation of the algorithm, which permits easier modification of the code.

This application report presents some of the advantages of speech coding, followed by some of the particular advantages of the CCITT standard. The basic concepts of ADPCM and a detailed description of the actual CCITT algorithm are also included. After a description of the algorithm, the implementation of the algorithm on the DSP56001 is discussed, including various techniques used to improve performance. Finally the real-time performance and other technical characteristics of the DSP56001 implementations are described. ■







---

## SECTION 2

# Speech Coding

***“Digital signals . . . are less sensitive to transmission noise and they are easier to multiplex, error protect, and encrypt for security than analog signals.”***

---

**S**peech coding, or speech compression, is one of the major application areas for digital signal processing in the field of speech processing. The goal of speech coding is to digitally code speech efficiently for either storage or transmission. In telecommunication applications, the goal is typically to code an analog speech signal into a digital format, transmit the digital signal, and then decode the digital signal back into an analog waveform — all in real time.

## 2.1 Trade-Offs

While the goal seems very simple, there are many trade-offs to be considered in a practical application. These trade-offs include decoded speech quality, transmission bandwidth, coder complexity, overall system cost, and real-time considerations. Different applications may have very different requirements. For example, military communication applications often sacrifice coder complexity and speech quality to achieve very low transmission bandwidths. Other applications, such as some voice mail systems, may not require real-time performance.

## 2.2 Advantages of Digital Speed Coding

Digital speech coding has advantages beyond the compression savings. Digital signals in general have many desirable properties: they are less sensitive to transmission noise and they are easier to multiplex, error protect, and encrypt for security than analog signals. Since coding algorithms can be implemented in software, modifications and improvements to algorithms are much easier than with dedicated hardware. In many applications such as computer workstations, a common DSP may perform several functions. These applications can add speech coding to a system without adding additional hardware.

As noted above, the ADPCM algorithm defined by the CCITT is intended for use in a telecommunications environment, although it may be applied in other areas. The CCITT algorithm actually implements a PCM/ADPCM/PCM conversion process and is called a transcoder for this reason. The standard is intended for use on digital channels that contain the digital equivalent of analog signals on analog telephone channels, so the CCITT algorithm has added complexity to handle non-speech signals (such as modem and DTMF signals) that may be present on analog telephone channels. It provides 2 to 1 compression allowing two ADPCM coded signals to be easily multiplexed into one basic 64-kbit/s digital channel. ■

---

## SECTION 3

# Types of Speech Coders

*“... logarithmic PCM (log PCM) coding, effectively compresses the input signal at the transmission end and expands it at the receiving end.”*

---

**A**lthough there are many different methods for speech coding, these methods generally fall into two main categories: **waveform coders** and **source coders**. Waveform coders deal with speech on a sample by sample basis. Their goal is to have the output waveform of the decoder match the original speech waveform as closely as possible. Source coders (also called vocoders), on the other hand, attempt to describe a speech signal in terms of parameters of a speech production model. These models typically estimate vocal tract shape and vocal tract excitation. Vocoders can operate at much lower bit rates than waveform coders but their output speech quality is generally not as good. ADPCM is classified as a waveform coder and is actually a combination of several basic waveform coding techniques. It should also be noted that many speech coding algorithms are hybrids that combine a variety of techniques. For instance, ADPCM techniques are used in portions of many other coder algorithms, such as subband coders and LPC coders.

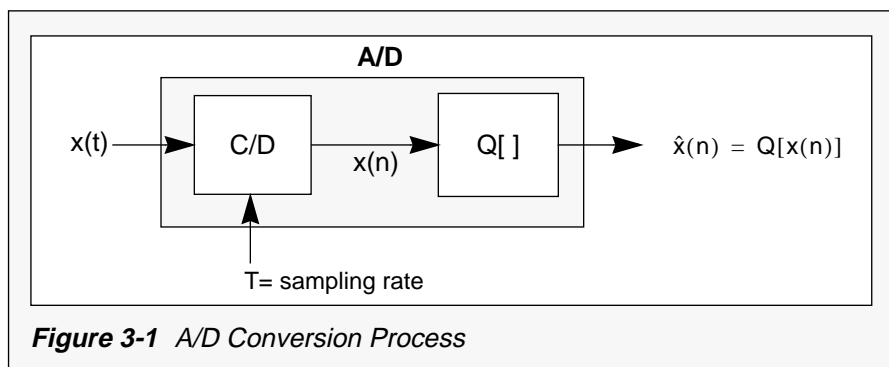
As noted above, ADPCM is a combination of several basic techniques. The following discussion is a brief

introduction to these techniques. Reference [2] also provides a good introduction to the topics discussed here as well as other types of speech coders.

## 3.1 Uniform PCM

In any digital speech coding system, an analog speech signal must be converted into a digital representation before it can be processed. Most digital coders use a form of pulse-code modulation (PCM) for the A/D conversion. The A/D conversion process is represented by a combination of two processes:

- a continuous-to-discrete conversion (C/D)
- a quantization ( $Q[\ ]$ ), as shown in Figure 3-1 [3]



**Figure 3-1** A/D Conversion Process

The C/D conversion changes a continuous time waveform into a discrete time waveform with continuous amplitude. Mathematically, this process will

not introduce any error into the input signal as long as the sampling rate is at least twice the highest frequency of the input signal. The quantizer maps each continuous amplitude sample into a digital code-word. The mapping process represented by the function ( $Q[\ ]$ ) is called the quantizer characteristic.

The quantization process will introduce an error called the quantizer error into the signal. This error can be represented as an additive white noise source in the quantizer as shown in Figure 3-2. In speech applications, this error will be evident as audible noise at the D/A output. In many speech coding applications, this quantization noise is expressed in terms of Signal-to-Quantization Noise Ratio (SNR).

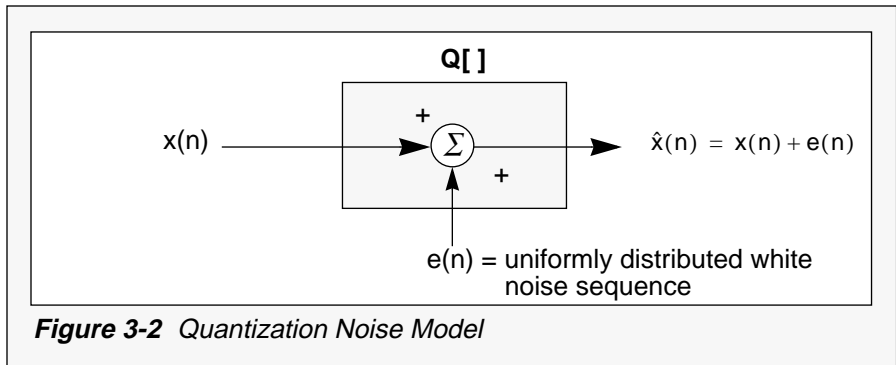
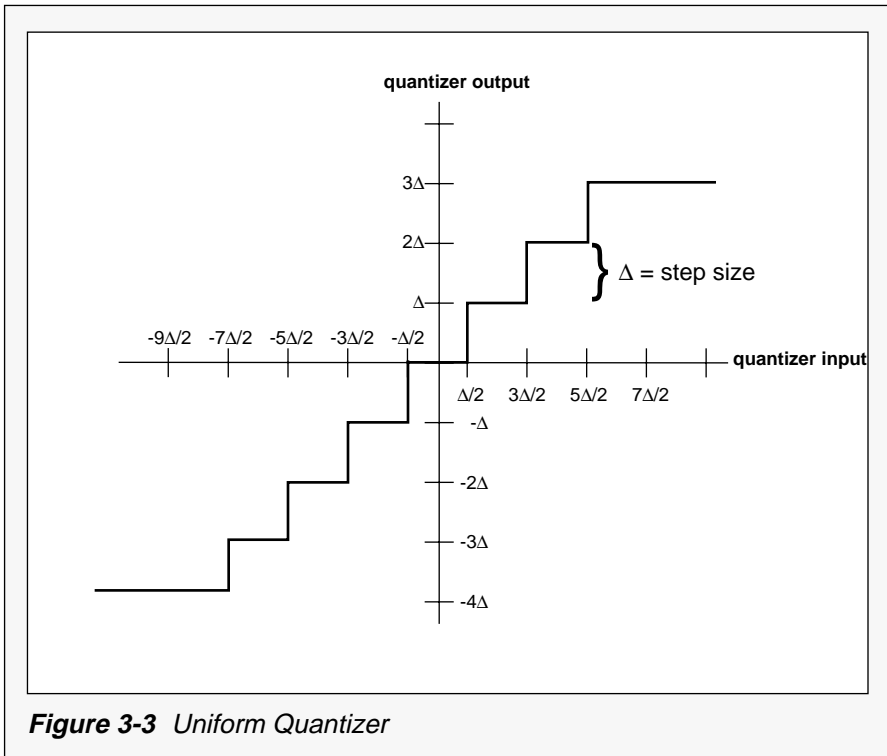


Figure 3-3 shows an example of a uniform quantizer. Uniform PCM quantizers feature a constant step size ( $\Delta$ ) [3]. To prevent large errors, the quantizer's range must represent the input signal's potentially maximum amplitude. The amount of noise that the quantizer introduces into the signal is also directly

related to the step size. Together, the quantizer range and the step size determine the number of bits required to adequately represent a signal of given quality. Telephone quality speech signals (“toll” quality) require about 35 dB of SNR in a frequency range of about 200-3200 Hz. Usually, 11-12 bits quantization at an 8 kHz sampling rate are needed to ensure this SNR over a typical range of speech signals [2].



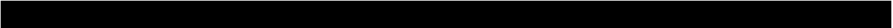
**Figure 3-3** Uniform Quantizer

## 3.2 Logarithmic PCM

One of the disadvantages of uniform PCM coding is that the SNR is not constant. For instance, during “voiced” segments of speech (vowels for example) the SNR may be high, but during “unvoiced” segments of speech (consonants for example) the SNR may be low. Consequently, bits are wasted to ensure that the maximum SNR is always less than an acceptable level. Logically, if one can reduce the maximum SNR, then fewer bits will be needed. Non-uniform quantizers have features that aim to reduce the maximum SNR, including a step size that varies according to the input signal, or alternatively, non-uniform scaling of input before it is quantized [4]. With this approach, the step size is greater for large input amplitudes than it is for small amplitudes. The goal is to obtain a uniform SNR over all input ranges so that the SNR is independent of the input.

A basic approach to achieve this goal with speech signals is to logarithmically space the quantization levels. This approach, called logarithmic PCM (log PCM) coding, effectively compresses the input signal at the transmission end and expands it at the receiving end. For this reason, log PCM coding is also referred to as companding [4]. True logarithmic quantization is not practical in reality, but two methods that approximate this technique have been standardized [5]. These methods are referred to as  $\mu$ -law and A-law companding. They achieve a SNR that is constant enough for practical purposes. Compared with uniform PCM, these techniques





need about four fewer bits per sample for equivalent speech quality. Therefore, only eight bits are needed per sample rather than the 11 or 12 bits that uniform PCM needs. Eight-bit log PCM companding is one of the simplest forms of speech coding and has also been standardized for digital telecommunications.

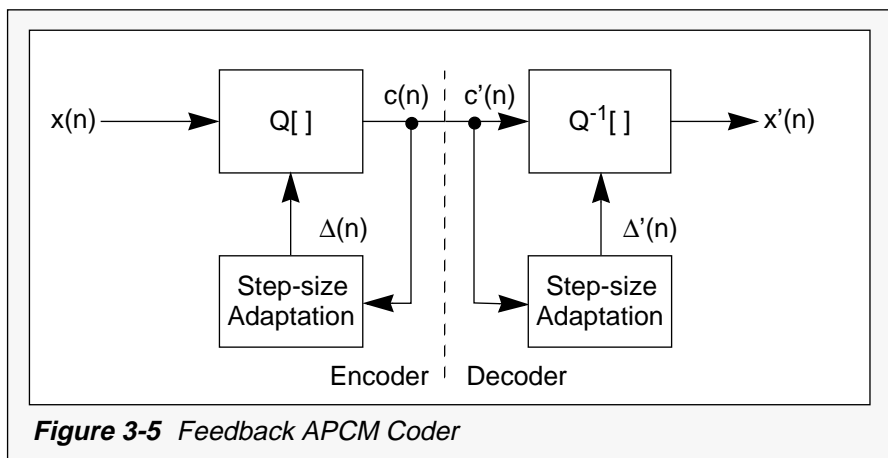
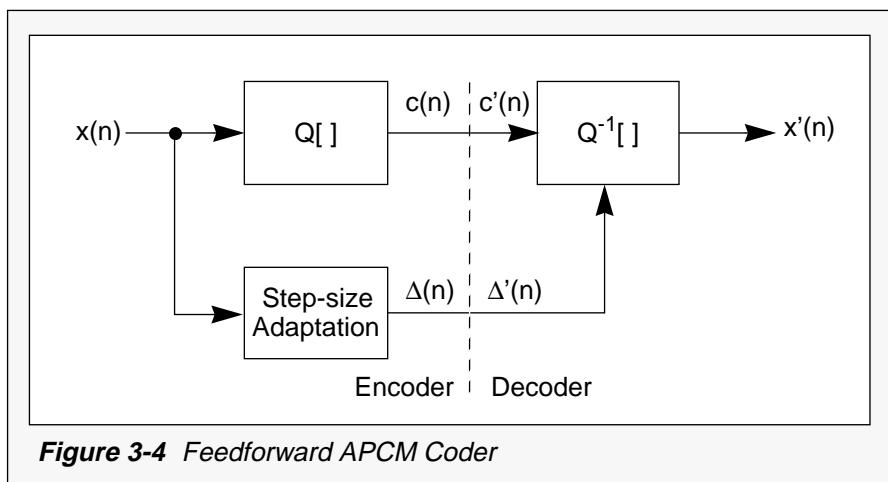
Speech coders are typically specified in terms of bit rate. The bit rate is the number of samples per second, times the number of bits per sample. As noted, telephone quality voice signals primarily range from 200 to 3200 Hz and they are typically sampled at 8 kHz to maintain this frequency range. Therefore, the basic standard data rate for  $\mu$ -law and A-law PCM data is 8 bits/sample at 8000 samples/second or 64,000 bits/second (64 kbit/s). This is the source of the data rate for a basic digital transmission channel. The process of converting log PCM signals to/from analog signals is often done using CODEC devices (such as the MC145503). In the CCITT algorithm, the log PCM data is converted to linear PCM data before the ADPCM encoding itself is performed and the decoded linear signal is converted back into log PCM form after the decoding process is completed. The process for converting between log and linear PCM data, including the implementation on the DSP56001, is described in detail in the Motorola applications report "Logarithmic/Linear Conversion Routines For DSP56000/1" [6].

## 3.3 ADPCM Coding

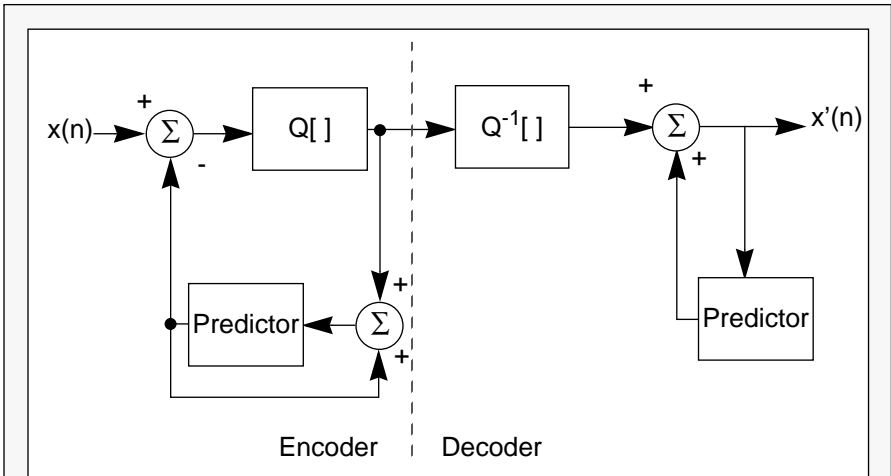
Speech coders try to remove “redundancy” in speech signals in order to further reduce the data rate of speech signals obtained by PCM methods. Speech coders do this by taking advantage of known characteristics of speech signals. ADPCM is a technique of speech compression based on a combination of two basic speech coding techniques, Adaptive PCM (APCM) and Differential PCM (DPCM). A basic difference between these techniques versus uniform and log PCM methods is that they require previous samples to be remembered while uniform and log PCM methods do not. A second key difference is that uniform and log PCM coders have fixed compression and expansion curves (fixed step sizes), while most adaptive PCM methods change their compression and expansion curves over time (adaptive step sizes).

APCM coders take advantage of the tendency for speech signals to vary relatively slowly [2]. The coders exploit this property by changing the characteristics of the coder adaptively over time. One way to doing this is to change the quantizer step size in proportion to the average speech amplitude. The step size modification can be achieved in two ways; by either directly scaling the step size or by scaling the input signal by a gain factor. Updating the step size or gain factor can also be done in one of two ways. The feedforward approach, shown in Figure 3-4, actually sends the update information to the decoder over the transmission channel [7]. The feedback approach, shown in Figure 3-5, deter-

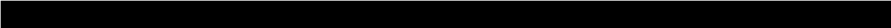
mines the information from the previously coded samples so that no extra information needs to be transmitted to the decoder [7]. APCM, in general, provides better SNR performance and speech quality at a given bit rate compared to uniform or log PCM but it does require more computation.



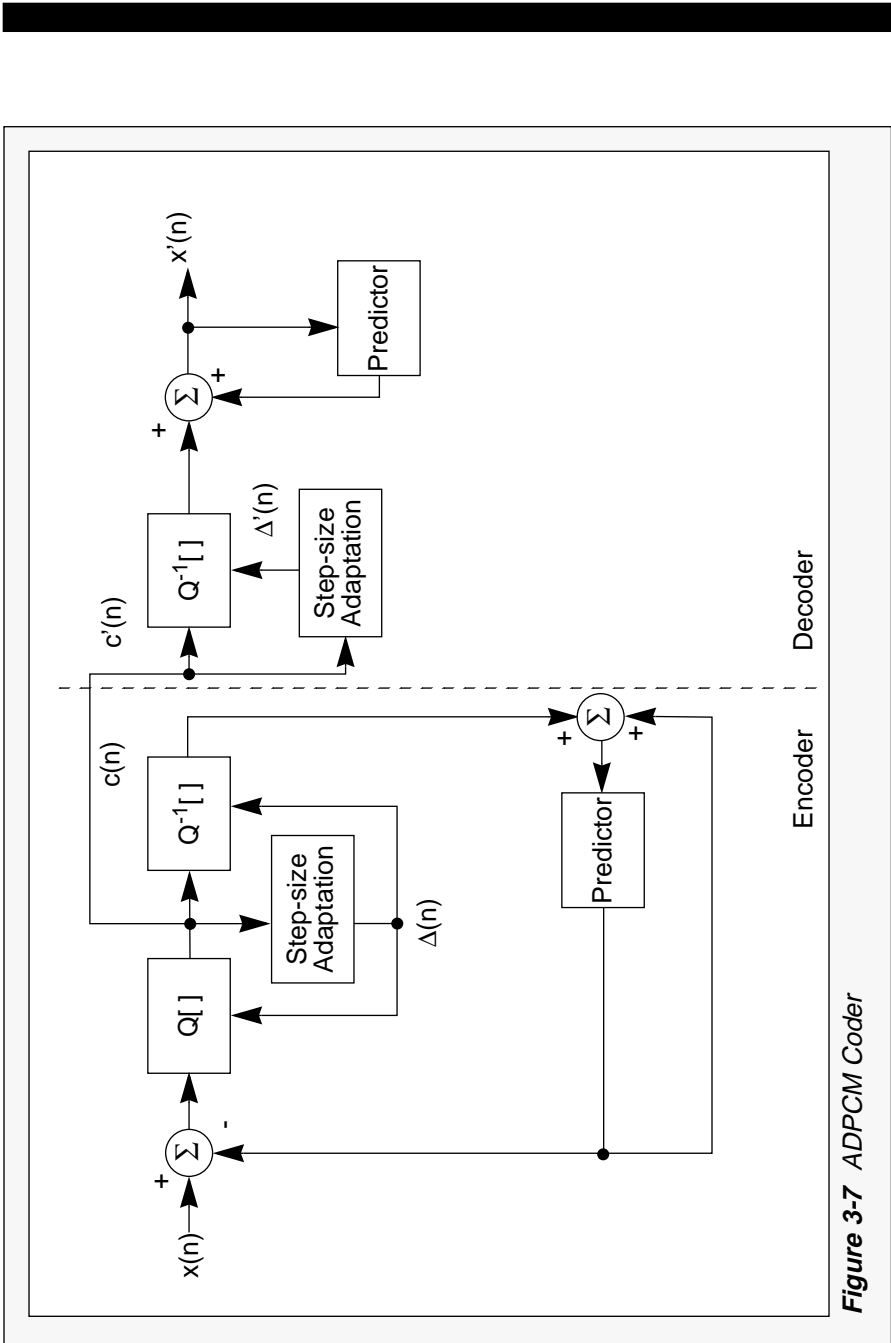
Other properties that are unique to speech signals relate to the spectral envelope. A typical short-time magnitude spectrum of a speech signal shows a slowly varying envelope whose shape is primarily determined by the vocal tract response. This causes speech samples to have a high degree of “sample-to-sample correlation” [2]. DPCM techniques try to take advantage of this characteristic. DPCM coders are characterized by the use of a predictor that forms an estimate of each incoming speech sample. This estimate is subtracted from the actual sample and the difference between them is coded instead of the actual input sample. Most DPCM coders use a form of linear prediction where the estimate is based on a linear combination of previous input samples. A block diagram of a DPCM coder is shown in Figure 3-6 [2].



**Figure 3-6** DPCM Coder



ADPCM techniques use a combination of APCM and DPCM techniques. There are many different ways to implement the concepts of APCM and DPCM. Therefore, the term ADPCM can justifiably refer to a broad range of speech coders that may have widely varying characteristics. ADPCM techniques, as well as APCM and DPCM techniques, may also be applied to non-speech signals, such as high-fidelity audio signals or video images. These implementations may not exploit properties that are specific to speech, however. The term ADPCM as used in this discussion refers specifically to the algorithm defined by the CCITT for telephone quality speech signals. Therefore the scope of this algorithm may not apply to all applications requiring signal compression. A general block diagram of the ADPCM configuration used in the CCITT algorithm is shown in Figure 3-7 [2]. ■



**Figure 3-7** ADPCM Coder



## SECTION 4

# The CCITT ADPCM Algorithm

***“The decoder portion of the CCITT algorithm uses the same routines as the encoder for the inverse quantization, linear prediction, tone detection, and adaptation functions.”***

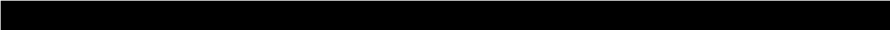
---

The CCITT ADPCM coder is designed to meet several specific requirements [8]. The design goals were to:

- provide compression while satisfying the objective signal quality requirements specified in CCITT Recommendation G.712 [10]
- retain a high enough level of subjective quality (defined by listening tests) even after a series of encodings and decodings
- provide compatibility with the existing  $\mu$ -law and A-law PCM formats
- operate stably in the presence of high-bit-error rates during transmission
- operate properly in the presence of voiceband data at up to 4.8 kbit/s

These requirements and others led to the standardization of this particular algorithm. The following paragraphs provide an overview of this algorithm, followed by a detailed description of each part. This application report focuses on the algorithm implementation rather than the complete development of the theory behind this algorithm.





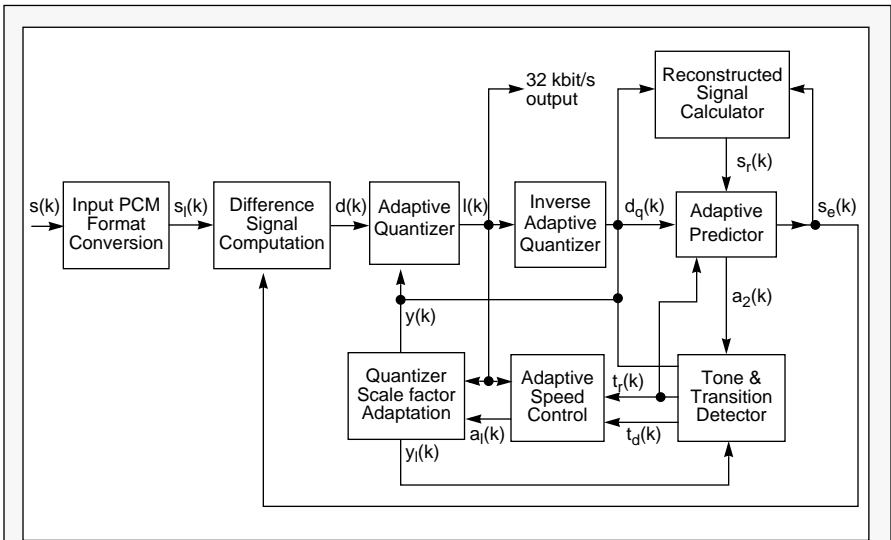
A detailed block diagram of the encoder portion of the CCITT ADPCM algorithm is shown in Figure 4-1. The decoder portion is shown in Figure 4-12. The algorithm uses the feedback method whereas the encoder uses only the coded ADPCM signal  $l(k)$  for feedback to the prediction and adaptation sections. Since this information is the same that the decoder uses for adaptation, no update parameters need to be sent over the transmission channel. This structure has two key properties:

1. The encoder and the decoder are almost functionally identical.
2. The decoder is in the same "state" as the encoder for a given sample (assuming no transmission errors).

Therefore, all common internal signals are identical, enabling the decoder to keep track of the encoder's adaptive process without explicitly receiving information from the encoder.

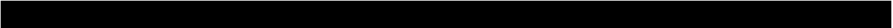
The APCM portion of the algorithm uses the gain factor approach for quantization. The difference signal to be coded is first scaled by the adaptive scale factor  $y(k)$  before it is quantized according to a fixed quantization curve. The smallest step size allowed by the overall quantization is equivalent to the smallest step size defined in  $\mu$ -law or A-law PCM. The largest step size is 1024 times the smallest step size. It should be noted that the gain factor itself and other signals used in its computation are in base 2 logarithmic form. The adaptation of the scale factor  $y(k)$  is based on a "bimodal" adaptation technique (the quantizer is also called a dynamic locking quantizer).

The scaling adaptation rate is “fast” for signals with large fluctuations, like speech, and is “slow” for signals with small fluctuations, like voiceband data and tones. A purely stationary input such as a single tone causes the quantizer to stop adapting or to “lock”. The overall speed of adaptation is a combination of the fast (unlocked) and slow (locked) scale factors.



**Figure 4-1** CCITT ADPCM Encoder Block Diagram (detailed)

The DPCM portion of the algorithm uses a linear predictor that is based on an autoregressive moving average (ARMA) process which has a combination of poles and zeros in its transfer function [4]. The structure for the predictor is based on several factors including stability in the presence of errors and



the ability to track both speech and voiceband data signals. The adaptation of the predictor coefficients is based on a gradient search or steepest descent method and all coefficients are updated for each input sample. The output of the linear predictor is the signal estimate  $s_e(k)$ . This signal is subtracted from the input signal to form the difference signal that is actually coded and sent to the decoder.

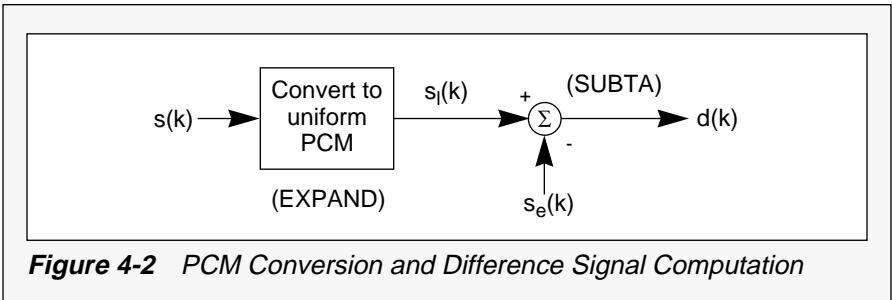
The additional blocks added to the CCITT coder are for PCM format conversion, tone transition detection, and synchronization. The tone transition detection determines when transitions between stationary tone signals occur. When a transition is detected all of the coefficients in the predictor are set to zero and the quantizer is set to the fast (unlocked) mode. The synchronization block helps prevent noise accumulation when multiple PCM/ADPCM/PCM conversions (synchronous tandem codings) are performed on a signal. This synchronization block does not affect the internal state of the decoder and has a minimal effect on the output quality of a single PCM/ADPCM/PCM conversion.

## 4.1 The Encoder Algorithm

Figure 4-1 shows a block diagram of the major portions of the CCITT ADPCM encoder. This section gives a detailed description of each block. The DSP56001 assembly code routines associated with each block are described in detail in **SECTIONS 5.2 and 5.3**.

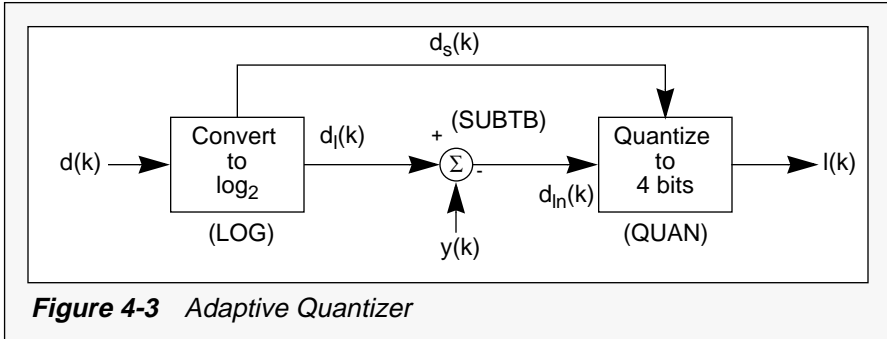
The first stage of the encoder is shown in Figure 4-2. The input to the encoder is the A-law or  $\mu$ -law PCM signal  $s(k)$ . This signal is converted to a uniform (linear) PCM signal  $s_l(k)$  in the routine EXPAND. In the next routine, SUBTA, the difference signal  $d(k)$  is calculated by subtracting the signal estimate  $s_e(k)$  from  $s_l(k)$ . As in all DPCM type coders, the difference signal is actually encoded and transmitted rather than a compressed version of the input signal.

$$d(k) = s_l(k) - s_e(k) \quad \text{Eqn. 4-1}$$



In the next stage of the encoder, shown in Figure 4-3, the difference signal  $d(k)$  is quantized by a 15-level non-uniform adaptive quantizer. The quantization process is performed by three routines. In the routine LOG the linear difference signal  $d(k)$  is converted to a base 2 logarithmic form,  $d_l(k)$  representing the magnitude, and  $d_s(k)$  representing the sign. The scale factor  $y(k)$  (also in logarithmic form) is then subtracted from  $d_l(k)$  in the routine SUBTB, in effect dividing the linear signal  $d(k)$  by a gain factor. This normalized

signal  $d_{in}(k)$  is then quantized in the routine QUAN according to the normalized input/output characteristic shown in Table 4-1.

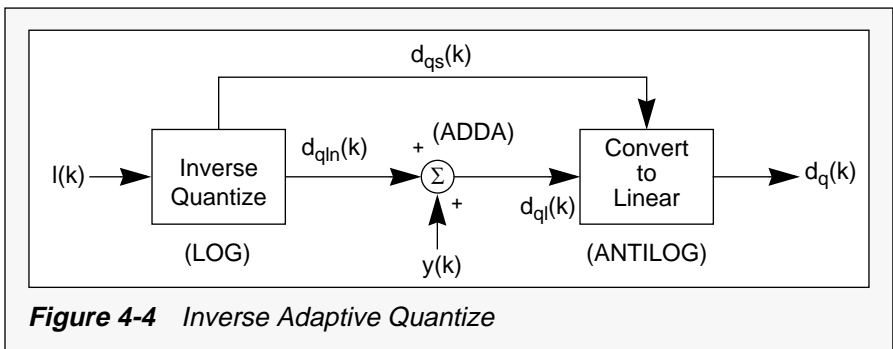


**Table 4-1 Quantizer Normalized Input/Output Characteristic**

INPUT	OUTPUT	
$\log_2 d(k) -y(k)$	$ l(k) $	$\log_2 d_g(k) -y(k)$
[3.12, $+\infty$ )	7	3.32
[2.72, 3.12)	6	2.91
[2.34, 2.72)	5	2.52
[1.91, 2.34)	4	2.13
[1.38, 1.91)	3	1.66
[0.62, 1.38)	2	1.05
[-0.98, 0.62)	1	0.031
$(-\infty, 0.98)$	0	$-\infty$

The output of the quantizer section is the 32-kbit/s ADPCM signal  $l(k)$ . This is the overall output of the encoder that is transmitted to the decoder. Each sample of  $l(k)$  contains four bits, three bits for the magnitude (from Table 4-1) and one bit for the sign (from  $d_s(k)$ ). The quantizer is a 15-level quantizer since the all-zero codeword is not allowed.

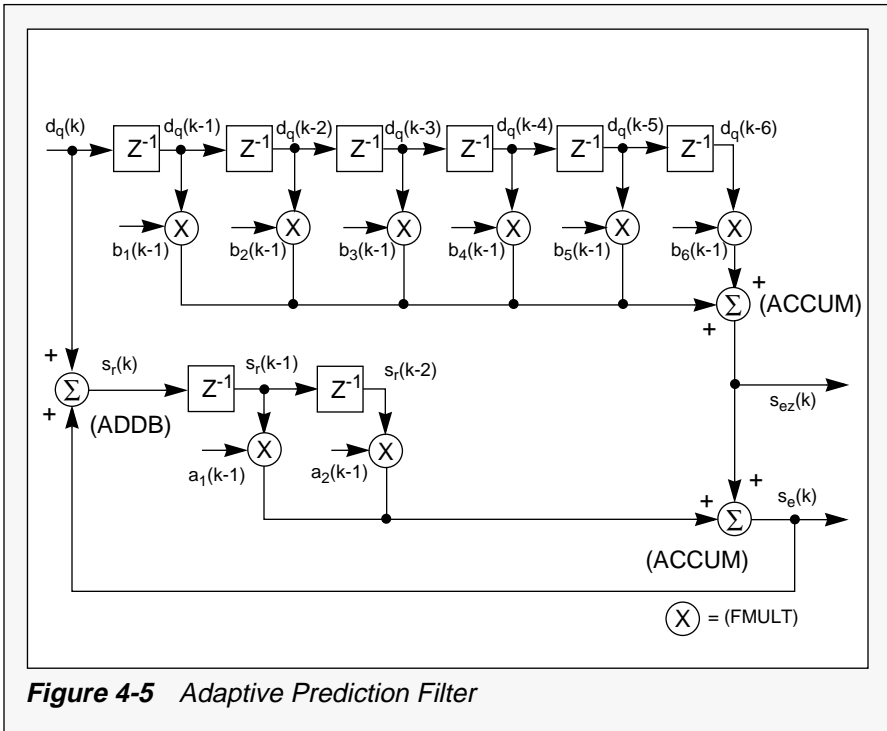
The inverse adaptive quantizer, shown in Figure 4-4, converts the ADPCM signal  $l(k)$  into the signal  $d_q(k)$ , the quantized version of the difference signal. The inverse quantization is performed in three routines that are effectively the inverse of the three quantizer routines. The routine RECONST maps the magnitude of  $l(k)$  into one of eight normalized output values shown in Table 4-1. The routine ADDA adds the scale factor  $y(k)$  (the same value as in the quantizer) to the normalized output value  $d_{qln}(k)$ , in effect multiplying by a gain factor. The routine ANTILOG then converts this logarithmic value  $d_{ql}(k)$ , along with the sign of  $l(k)$  ( $d_{qs}(k)$ ), to the linear quantized difference signal  $d_q(k)$ .



**Figure 4-4** Inverse Adaptive Quantize

The remaining portion of the encoder performs three main functions:

1. calculating the new signal estimate
2. performing the adaptation functions
3. detecting tones



The adaptive predictor's primary function is to use the past history of the quantized difference signal  $d_q(k)$  to update the signal estimate  $s_e(k)$ . The linear predictor model used for the prediction consists of a

sixth order section that models zeroes and a second order section that models poles. The prediction filter shown in Figure 4-5, is implemented in the routines FMULT and ACCUM using the equations:

$$s_{ez}(k) = \sum_{i=1}^6 b_i(k-1) \cdot d_q(k-i) \quad \text{Eqn. 4-2}$$

$$s_e(k) = \sum_{i=1}^2 a_i(k-1) \cdot s_r(k-i) + s_{ez}(k) \quad \text{Eqn. 4-3}$$

The CCITT standard specifies that the multiplies in Eqn. 4-2 and Eqn. 4-3 be done in floating point so the values of  $d_q(k)$  and  $s_r(k)$  must be converted to floating point. This is done in the routines FLOATA and FLOATB. The signal  $s_r(k)$  in Eqn. 4-2 and Eqn. 4-3 is the reconstructed signal. The routine ADDB calculates  $s_r(k)$  by adding the quantized difference signal  $d_q(k)$  to the signal estimate  $s_e(k)$  as shown in Eqn. 4-4. The reconstructed signal represents the overall output of the ADPCM algorithm. The encoder does not output this signal but uses it as feedback for the prediction.

$$s_r(k-i) = s_e(k-i) + d_q(k-i) \quad \text{Eqn. 4-4}$$

The predictor coefficients  $a_i(k)$  and  $b_i(k)$  are updated for each sample using a gradient search algorithm. The adaptation of the two pole coefficients,  $a_1(k)$



and  $a_2(k)$  is shown in Figure 4-6 and Figure 4-7 respectively. These coefficients are updated according to the following equations:

$$a_1(k) = (1 - 2^{-8})a_1(k-1) + (3 \cdot 2^{-8})\text{sgn}[p(k)]\text{sgn}[p(k-1)] \quad \text{Eqn. 4-5}$$

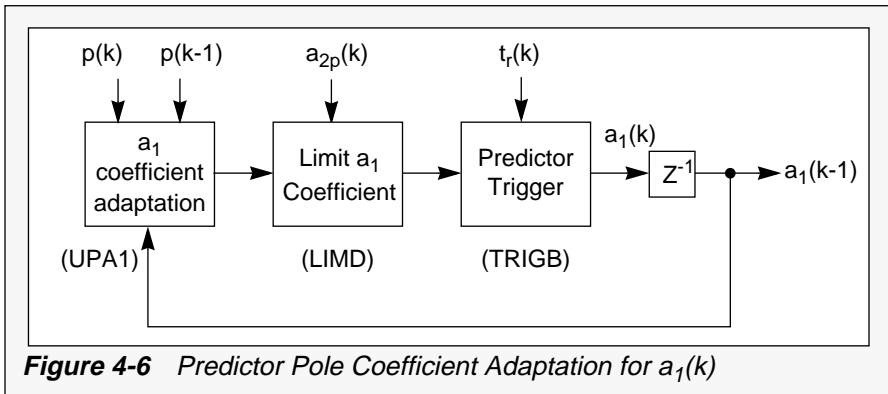
$$a_2(k) = (1 - 2^{-7})a_2(k-1) + 2^{-7}(\text{sgn}[p(k)]\text{sgn}[p(k-2)] - f[a_1(k-1)]\text{sgn}[p(k)]\text{sgn}[p(k-1)]) \quad \text{Eqn. 4-6}$$

where:

$$p(k) = d_q(k) + s_{ez}(k) \quad \text{Eqn. 4-7}$$

and:

$$f(a_1) = \begin{cases} 4a_1 & |a_1| \leq 2^{-1} \\ 2 \text{sgn}(a_1) & |a_1| > 2^{-1} \end{cases} \quad \text{Eqn. 4-8}$$

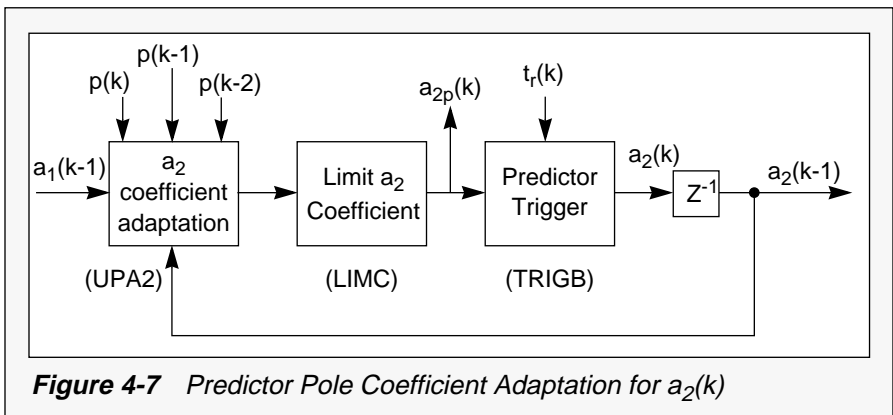


Eqn. 4-5 is implemented in the routine UPA1 while Eqn. 4-6 and Eqn. 4-8 are implemented in the routine UPA2. Eqn. 4-7 is calculated in the routine ADDC. The coefficients  $a_1(k)$  and  $a_2(k)$  are constrained for stability reasons to the following limits:

$$|a_2(k)| \leq 0.75 \quad \text{Eqn. 4-9}$$

$$|a_1(k)| \leq 1 - 2^{-4} a_2(k) \quad \text{Eqn. 4-10}$$

Eqn. 4-9 is calculated in the routine LIMC and Eqn. 4-10 is calculated in the routine LIMD.



The adaptation of the  $b_i(k)$  zero coefficients is shown in Figure 4-8. They are updated in the routines XOR and UPB according to:

$$b_i(k) = (1 - 2^{-8}) b_i(k-1) + 2^{-7} \text{sgn}[d_q(k)] \text{sgn}[d_q(k-i)]$$

for  $i = 1, 2, \dots, 6$  Eqn. 4-11

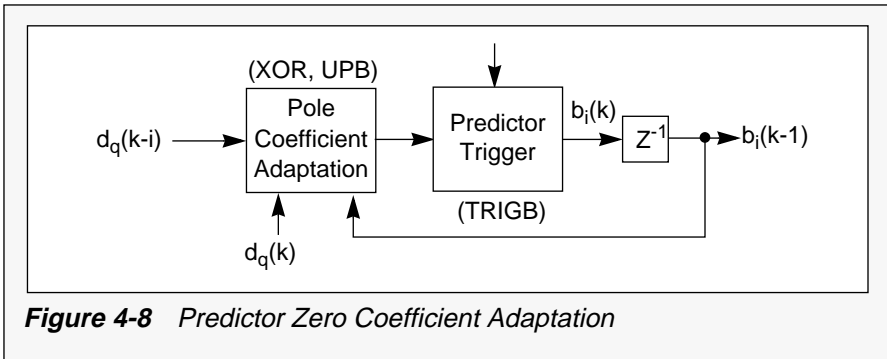
The  $b_i(k)$  coefficients are implicitly limited to  $\pm 2$ .

The function  $\text{sgn}[x]$  in Eqn. 4-5 through represents the sign of  $x$ . It is defined as:

$$\begin{aligned} \text{sgn}[x] &= +1 \text{ if } x > 0 && \text{Eqn. 4-12} \\ &= -1 \text{ if } x < 0 \\ &= +1 \text{ if } x = 0 \text{ and } i \neq 0 \\ &\quad \text{[where } x = p(k-i) \text{ or } x = d_q(k-i)] \\ &= 0 \text{ if } x = p(k) = 0 \text{ or } x = d_q(k) = 0 \end{aligned}$$

The predictor coefficients may be further modified by the tone transition signal  $t_r(k)$ . The routine TRIGB tests  $t_r(k)$  for a transition detection and sets the predictor coefficients to 0 if a transition is detected.

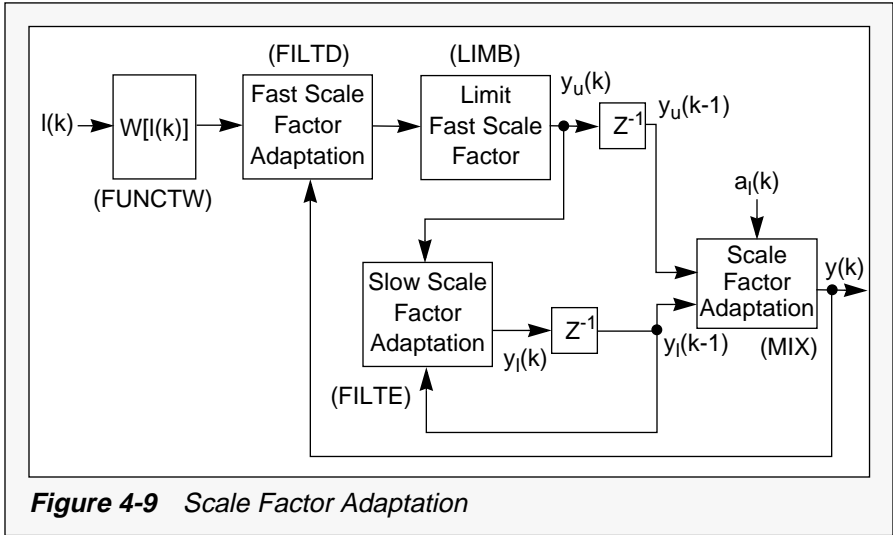
$$\text{If } t_r(k) = 1 \quad \text{then } a_i(k) = b_i(k) = t_d(k) = 0 \quad \text{Eqn. 4-13}$$



**Figure 4-8** Predictor Zero Coefficient Adaptation

The adaptation of the scale factor  $y(k)$  is based on information from past values of  $l(k)$  and the speed

control parameter  $a_i(k)$ . The overall speed of adaptation is a combination of the fast and slow scale factors  $y_u(k)$  and  $y_l(k)$ . The speed control parameter  $a_i(k)$  determines how the fast and slow scale factors are combined. A diagram of this process is shown in Figure 4-9.



The fast (unlocked) scale factor  $y_u(k)$  is calculated in the routine FILTD equation:

$$y_u(k) = (1 - 2^{-5}) y(k) + 2^{-5} W[I(k)] \quad \text{Eqn. 4-14}$$

The routine LIMB constrains  $y_u(k)$  to the limits:

$$1.06 \leq y_u(k) \leq 10.00 \quad \text{Eqn. 4-15}$$

The function  $W(l)$  according to Eqn. 4-14 is calculated in the routine FUNCTW according to Table 4-2.

**Table 4-2**  $W(l)$  Lookup Table

$ l(k) $	7	6	5	4	3	2	1	0
$W(l)$	70.13	22.19	12.38	7.00	4.00	2.56	1.13	-0.75

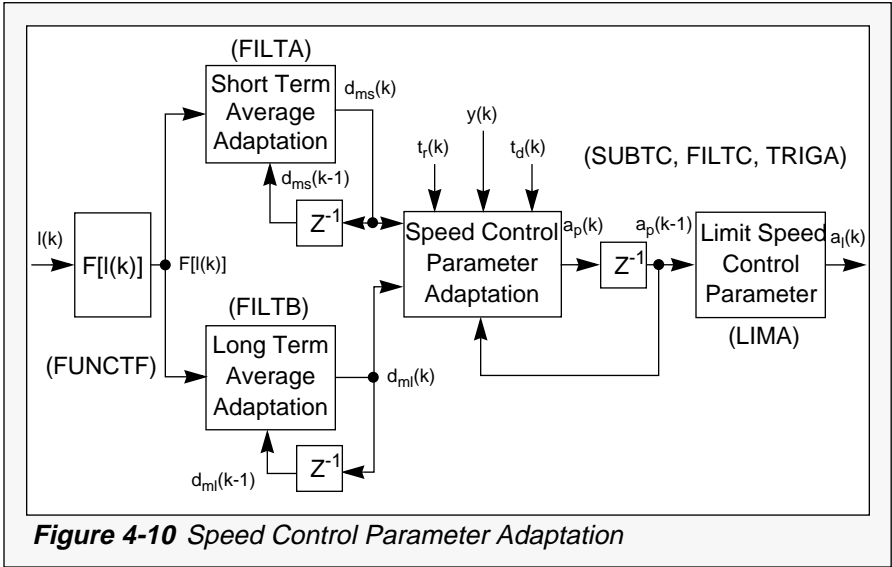
The slow (locked) scale factor  $y_l(k)$  is calculated in the routine FILTE by the equation:

$$y_l(k) = (1 - 2^{-6}) y_l(k-1) + 2^{-6} y_u(k) \quad \text{Eqn. 4-16}$$

The overall scale factor  $y(k)$  is determined in the routine MIX by the equation:

$$y(k) = a_l(k) y_u(k-1) + [1 - a_l(k)] y_l(k-1) \quad \text{Eqn. 4-17}$$

The speed control parameter  $a_l(k)$  is limited to the range  $0 < a_l(k) < 1$ . The value of  $a_l(k)$  approaches 1 for speech signals in which case the fast (unlocked) scale factor  $y_u(k)$  dominates in Eqn. 4-17. For data signals and tones however, the value of  $a_l(k)$  approaches 0 in which case the slow (locked) scale factor  $y_l(k)$  dominates in Eqn. 4-17. The value of  $a_l(k)$  is determined primarily by the rate-of-change of the difference signal, which is encoded in  $l(k)$ . The update of  $a_l(k)$  is shown in Figure 4-10.



Two measures of the difference signal value are used to update the speed control parameter,  $d_{ms}(k)$  and  $d_{ml}(k)$ . The signal  $d_{ms}(k)$  represents the “short term” average of the function  $F[I(k)]$ , while  $d_{ml}(k)$  represents the “long term” average of  $F[I(k)]$ . The value of  $F[I(k)]$  is a weighted function of  $I(k)$  and is determined in the routine FUNCTF according to Table 4-2. The difference signals  $d_{ms}(k)$  and  $d_{ml}(k)$  are calculated in the routines FILTA and FILTB respectively according to the equations:

$$d_{ms}(k) = (1 - 2^{-5}) d_{ms}(k-1) + 2^{-5} F[I(k)] \quad \text{Eqn. 4-18}$$

$$d_{ml}(k) = (1 - 2^{-7}) d_{ml}(k-1) + 2^{-7} F[I(k)] \quad \text{Eqn. 4-19}$$

The values of  $d_{ms}(k)$  and  $d_{ml}(k)$  are used to determine the “unlimited” speed control parameter  $a_p(k)$ . The value of  $a_p(k)$  is calculated in the routines SUBTC, FILTC, and TRIGA by the equation:

$$\begin{aligned}
 a_p(k) &= (1-2^{-4})a_p(k-1) + 2^{-3} && \text{if } |d_{ms}(k)-d_{ml}(k)| \geq 2^{-3}d_{ml}(k) \\
 &= (1-2^{-4})a_p(k-1) + 2^{-3} && \text{if } y(k) < 3 \\
 &= (1-2^{-4})a_p(k-1) + 2^{-3} && \text{if } t_d(k) = 1 \\
 &= 1 && \text{if } t_r(k) = 1 \\
 &= (1-2^{-4})a_p(k-1) && \text{otherwise}
 \end{aligned}
 \tag{Eqn. 4-20}$$

The value of  $a_p(k)$  tends towards 2 if the difference between  $d_{ms}(k)$  and  $d_{ml}(k)$  is large, indicating the rate-of-change of the difference signal is fast, but it tends towards 0 if the difference is small, indicating the rate-of-change is slow. The value of  $a_p(k)$  also tends towards 2 when tones are detected ( $t_d(k) = 1$ ) or in idle channel conditions ( $y(k) < 3$ ). When a tone transition is detected ( $t_r(k) = 1$ ), explicitly set to 1.

**Table 4-3**  $F[l(k)]$  Lookup Table

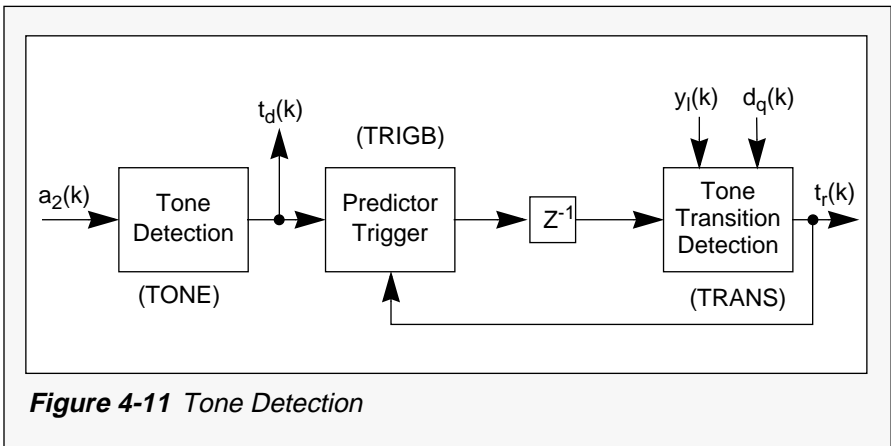
$l(k)$	7	6	5	4	3	2	1	0
$F[l(k)]$	7	3	1	1	1	0	0	0

To form the desired speed control parameter  $a_l(k)$  the parameter  $a_p(k)$  is constrained in the routine LIMA to the limits given in . This limiting has the effect of delaying a state transition start until the magnitude of the difference signal become relatively constant.

$$\begin{aligned}
 a_i(k) &= 1 && \text{if } a_p(k-1) > 1 \\
 &= a_p(k-1) && \text{if } a_p(k-1) \leq 1
 \end{aligned}
 \qquad \text{Eqn. 4-21}$$

The final step of the encoder is tone detection, shown in Figure 4-11. It is included to improve the performance of the coder in the presence of non-speech voice-band data signals that may be present on a typical analog phone line (e.g. DTMF tones and data modems.) The tone detect signal,  $t_d(k)$ , indicates the presence of a tone. When a tone is detected,  $t_d(k)$  causes the quantizer to be driven into the fast mode of adaptation. The TONE routine calculates  $t_d(k)$  by the equation:

$$\begin{aligned}
 t_d(k) &= 1 && \text{if } a_2(k) < -0.71875 \\
 &= 0 && \text{otherwise}
 \end{aligned}
 \qquad \text{Eqn. 4-22}$$



**Figure 4-11** Tone Detection



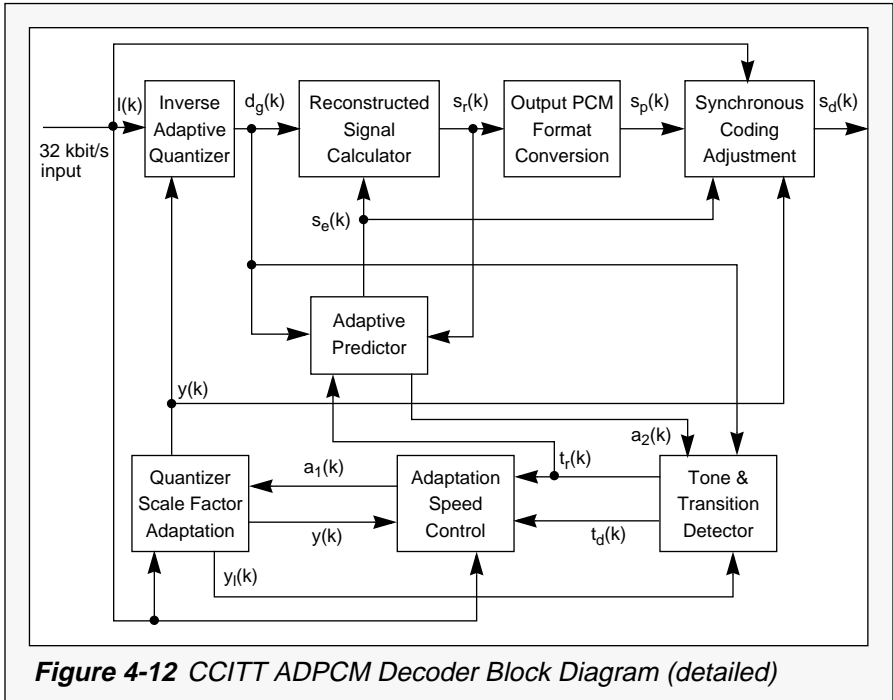
The tone detect signal,  $t_d(k)$ , also causes the tone transition detect signal,  $t_r(k)$ , to be set when a transition between tones occurs. This signal sets the predictor coefficients to 0 and the tone detect signal to 0 (in the TRIGB routine) so that the fast adaptation mode will take effect immediately. The tone transition detect signal,  $t_r(k)$ , is determined in the routine TRANS by the equation:

$$\begin{aligned}
 t_r(k) &= 1 && \text{if } a_2(k) < -0.71875 \text{ and } |dq(k)| > 24 \cdot 2 y_i(k) \\
 &= 0 && \text{otherwise}
 \end{aligned}
 \quad \text{Eqn. 4-23}$$

## 4.2 The CCITT Decoder Algorithm

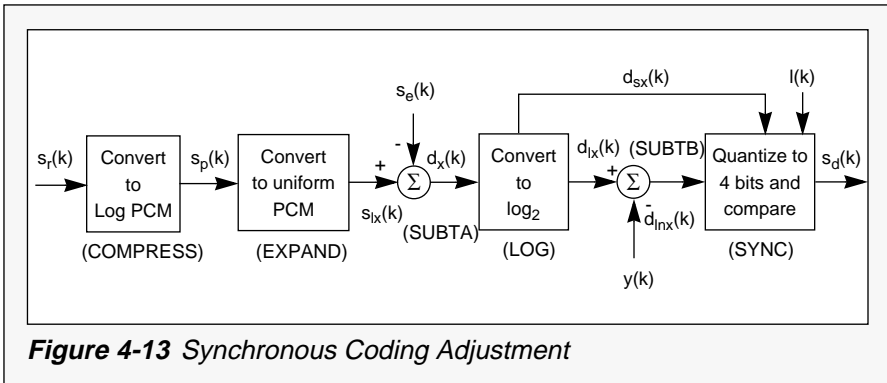
A detailed block diagram of the decoder process is shown in Figure 4-12. As mentioned previously, the CCITT ADPCM coder uses the feedback scheme and one of the properties of this scheme is that the encoder and decoder are almost identical in terms of function. The decoder portion of the CCITT algorithm uses the same routines as the encoder for the inverse quantization, linear prediction, tone detection, and adaptation functions. The input to the decoder,  $l(k)$ , is the same  $l(k)$  used in the encoder for adaptation and prediction. The decoder executes all of the same routines as the encoder (and in the same order) except for the input PCM conversion (EXPAND), the difference signal computation (SUBTA), and the adaptive quantization (LOG, SUBTB, and QUAN). These sections are normally not needed in the decoder but they are used in the

CCITT algorithm however, as explained in the following paragraphs.



The linear output of the decoder is the reconstructed signal  $s_r(k)$  as calculated in the routine ADDB. This is converted to the A-law or  $\mu$ -law PCM signal  $s_p(k)$  in the routine COMPRESS. This would normally be the final output but instead this signal is passed through a synchronous coding adjustment block, shown in Figure 4-13. The purpose of this block is to prevent cumulative distortions that may occur with “synchronous tandem codings” - multiple

ADPCM/PCM/ADPCM conversions on a transmission path. These distortions can only be prevented when the transmission paths are error-free and when no extra digital signal processing functions are performed on intermediate PCM and ADPCM signals.



**Figure 4-13** Synchronous Coding Adjustment

As noted previously, the encoder and decoder will be in the same internal "state" (all internal variables the same) assuming there are no transmission errors. The decoder then estimates the quantization that occurred in the encoder and forces the ADPCM sequence which it reconstructs to match the ADPCM sequence which it received. The decoder does so by converting the PCM signal  $s_p(k)$  back to a linear signal  $s_{ix}(k)$  in the EXPAND routine. A new difference signal  $d_x(k)$  is then calculated in the SUBTA routine by the equation:

$$d_x(k) = s_{ix}(k) - s_e(k) \quad \text{Eqn. 4-24}$$

The new difference signal  $d_x(k)$  is then converted to the normalized logarithmic signal  $d_{\ln x}(k)$  in the routines LOG and SUBTB. The same quantization as in the encoder then occurs in the routine SYNC. But this routine also does a comparison of the new coded ADPCM signal to the received ADPCM signal  $l(k)$ . The final PCM output of the decoder,  $s_d(k)$ , is determined by this comparison, defined by:

$$\begin{aligned}
 s_d(k) &= s_p(k)^+ && \text{if } d_x(k) < \text{lower interval boundary} \\
 &= s_p(k)^- && \text{if } d_x(k) \geq \text{upper interval boundary} \\
 &= s_p(k) && \text{otherwise}
 \end{aligned}$$

Eqn. 4-25

where:

$s_p(k)^+ =$  the PCM code word that represents the next more positive PCM output level

$s_p(k)^- =$  the PCM code word that represents the next more negative PCM output level





## SECTION 5

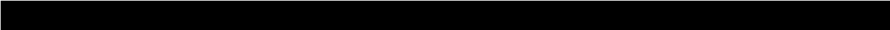
# ADPCM Implementation on the DSP56001

***“The standard version of the DSP56001 ADPCM source code implements the ADPCM algorithm exactly as specified by the CCITT.”***

---

**T**wo versions of the CCITT ADPCM algorithm described in previous sections have been implemented on the DSP56001, both in a full-duplex configuration. The assembly source code for these programs are available on the Motorola DSP bulletin board (Dr. BuB) under the names ADPCM.ASM and ADPCMNS.ASM. The code for the standard version is provided in a form that will process data in files on a host computer. The non-standard version is set up for real-time operation. The I/O interface to either version can be easily modified for other configurations.

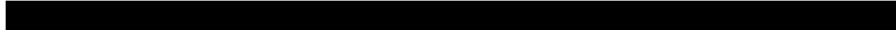
The program that implements the standard version, ADPCM.ASM, has been optimized. This optimization enables it to perform both the encode and the decode portions of the algorithm in real-time on a single DSP56001, running at 27 MHz with external program memory. The source code is set up to run the CCITT test sequences specified in Appendix II of Recommendation G.721 (1986 version) [1]. This code is a bit-for-bit compatible implementation of the CCITT specification and correctly passes all  $\mu$ -law and A-law test sequences provided by the CCITT.



ADPCMNS.ASM is the program for the non-standard version and it is a modification of the standard implementation. ADPCMNS.ASM performs the complete ADPCM algorithm in real-time on a single DSP56001, and requires less computational power than the standard version. In addition to providing a more efficient implementation of the ADPCM algorithm, this version is better suited for modification since the algorithm is programmed more directly than the CCITT standard specification allows.

For both ADPCM versions, the encoder and decoder portions of the source code are designed to be independent of the I/O interfaces so that the code can be easily modified for a variety of configurations, including single or multiple channel half-duplex configurations. For the standard version, this feature permits real-time performance on a slower speed DSP56001 or allows other simultaneous tasks to be performed on the same 27 MHz DSP56001. For the non-standard version, this feature allows an even greater variety of configurations. Further performance details for both versions are described in **SECTION 5.5 Performance Specifications**. This section details the implementation of the CCITT ADPCM algorithm on the DSP56001.

This application report provides only a basic description of the source code. For a more complete understanding of the DSP56001 code, refer to the CCITT document. Many of the details in Recommendation G.721 are not included in this document but have a significant impact on the assembly implementations, especially the standard version. In

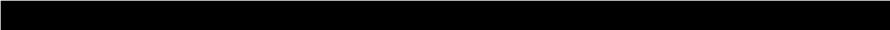


many cases, the standard code does not implement the equations given in this document in a straightforward manner due to the way the specification is written. Also, many of the comments in the source code refer to the notations used in Recommendation G.721. (See **APPENDIX A Terminology** for definitions of the basic terminology).

## 5.1 I/O Interface

The standard ADPCM source code is set up to run the CCITT test sequences on the Motorola DSP56000ADS board. The program simulates the PCM and ADPCM interfaces by using the file I/O routines on the ADS board. The file I/O routines allow programs running on the ADS board to access data in ASCII files on the host computer. These routines provide a convenient method for accessing the CCITT test files which are distributed in ASCII format. The ADS does require that the data in the test files be in a slightly different format than that provided by the CCITT. The details of this format can be found in the file ADPCM.HLP located with the ADPCM source code on the Motorola DSP bulletin board. The source code is set up to process a PCM input file to test the encoder and an ADPCM input file to test the decoder simultaneously. Two output files are written, one for the encoded ADPCM output and one for the decoded PCM output. When running the CCITT test sequences these output files can be compared to the CCITT files to verify correct operation. This procedure is



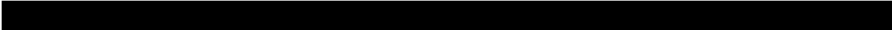


also discussed in the ADPCM.HLP file. The data in the two files being processed does not have to be related in any way since the encoder and decoder are designed to operate on two independent signals. Additionally, any file containing PCM data in ASCII hex characters may be used as input to the encoder, and likewise any file containing ADPCM data can be used as input to the decoder. It should be noted however that the file I/O routines on the ADS are not designed for high-speed data transfer so that processing data files with the DSP56001 ADPCM program will not be in real-time.

The non-standard ADPCM program includes the code required for a real-time I/O interface. The PCM channel is provided by a Motorola MC145503 CODEC connected to the DSP56001's Synchronous Serial Interface (SSI). Eight general-purpose I/O pins on the DSP56001 are used for the ADPCM channel. Four are used for parallel input and the other four are used for parallel output.

The SSI interface (both the transmit and receive) and the parallel I/O interface are assumed to be synchronous. The code is synchronized with the I/O interface by polling. No interrupts are used, although they can be added if desired. The real-time full-duplex operation of the non-standard ADPCM program has been tested on a set-up consisting of two DSP56000ADS. Further details of this test set-up can be found in the file ADPCMNS.HLP.

As mentioned previously, the I/O interface of the ADPCM programs are designed to be flexible for a variety of configurations. The standard ADPCM



code is provided with a file I/O interface to allow easy testing of the CCITT sequences, but it can be easily modified for a real-time interface. This was done to test the real-time operation of the algorithm. Only the interface portion of the source code was changed. The algorithm itself was not modified. The non-standard code already contains code for a real-time interface, but it can be modified for other configurations as well. In addition to the test set-up using CODECs for the PCM channel, this code was tested with a 16-bit linear A/D and D/A interface. In this case the PCM compression and expansion routines were removed, but again the algorithm itself was not modified.

## 5.2 Standard Implementation

The standard version of the DSP56001 ADPCM source code implements the ADPCM algorithm exactly as specified by the CCITT. The advantage of using this version is that the user can be confident that the DSP56001 implementation will perform exactly as specified by the CCITT. This includes performance with non-speech signals and in special operating conditions. The disadvantage of this version is that the specification does not always allow the equations to be implemented in an efficient manner on a general purpose digital signal processor. An example is the multiply and accumulation portion of the linear predictor. The standard specifies that the multiply be done in a floating-point

format while the accumulation be done in 16-bit fixed point format. Not only is the floating-point multiply less efficient than a native 24-bit fixed point multiply on the DSP56001, but several conversions between fixed and floating-point formats are required for each sample (see **SECTION 5.2.5 Floating-Point Conversion**). This is the most time consuming part of the algorithm but other parts of the specification also do not permit efficient implementation on a programmable microprocessor.

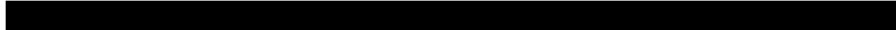
The standard implementation was written with the following two objectives:

- to adhere with the CCITT specification in order to maintain bit-for-bit compatibility with the CCITT test sequences
- to obtain a full-duplex solution with real-time operation on a single DSP56001

Knowing that the standard version was written with these two primary objectives may help to clarify the way the algorithm was implemented.

### **5.2.1 Code Structure**

The assembly program for the standard version of the ADPCM algorithm is structured as two main routines, the encoder (transmit) and the decoder (receive), plus one subroutine for initialization. After the initialization routine, the encoder routine is executed, followed by the decoder. The code then alternates between the encoder and decoder indefinitely. A flow diagram of the encoder and decoder is shown in Figure 5-1. This shows the order in which



the various portions of the algorithm are executed. The order of execution of the individual CCITT routines along with their execution speed is given in **SECTION 5.4 Performance Specifications**.

The encoder and decoder routines are designed to operate as independent code segments. They do assume that appropriate variables are stored in data memory and that appropriate pointers have been set. In particular, address registers r1, r2, r6, and r7 should contain the appropriate memory addresses prior to executing the encoder and decoder sections. Registers r3 and r5 should contain constant address values used for table lookup. Registers r0 and r4 do not need to be initialized since they are used as general purpose registers. The encoder and decoder are not set up as subroutines in the program. If interrupts are used for data I/O, they can easily be made into subroutines or interrupt routines. However, one routine should not interrupt the other routine until it is completely finished executing.

No subroutines are used within the encoder or the decoder so that optimal speed can be obtained. The code has also been optimized to take advantage of the DSP56001's architecture as much as possible. This causes the various CCITT routines in the code to "overlap" in many cases, meaning that variables and data values for one routine may be read from memory while the previous routine is still executing. In one case, the XOR and UPB routines are actually combined into a single section of code.

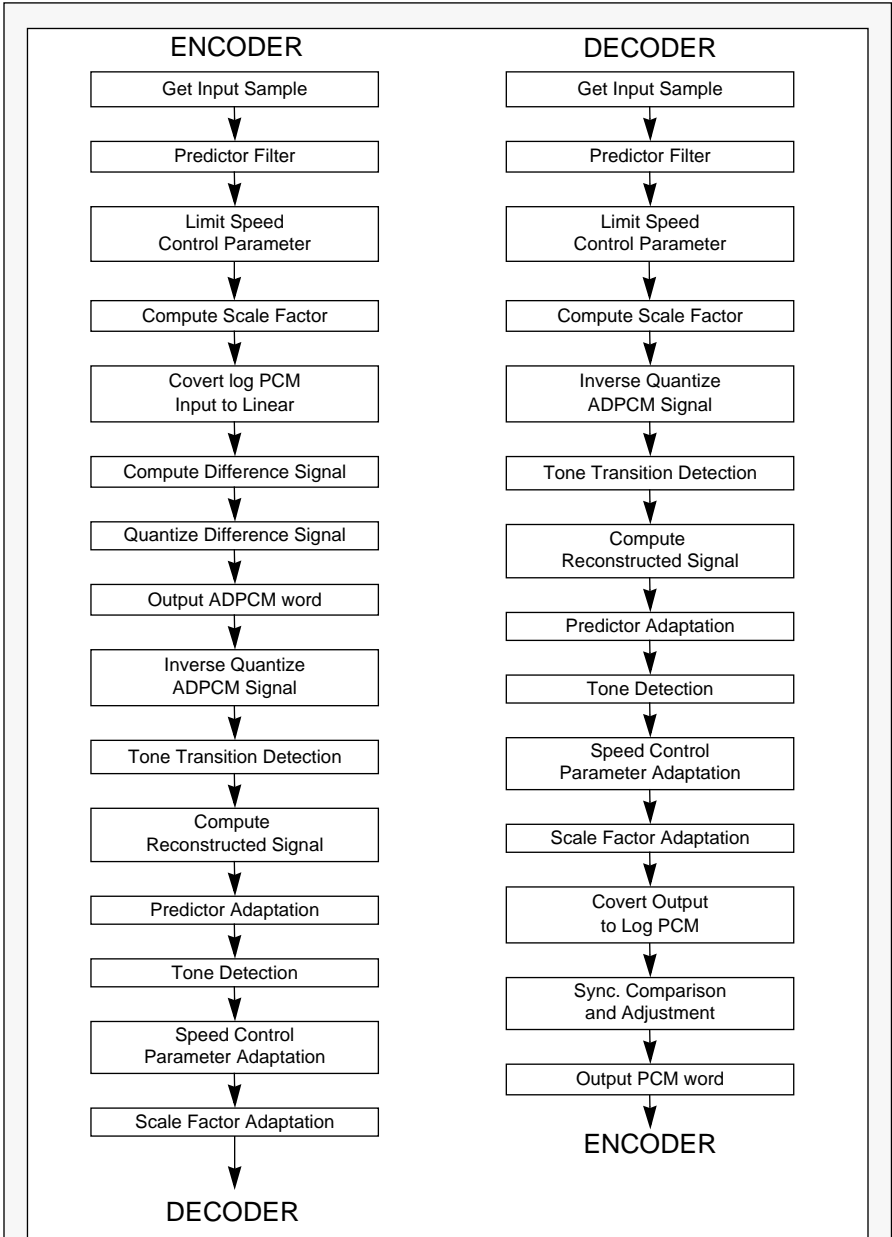
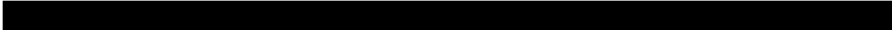
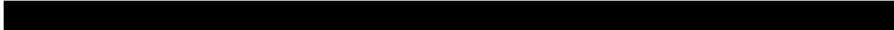


Figure 5-1 Code Flow Diagram



This optimization makes the code more difficult to follow in some cases, however, extra comments were added to clarify most of these cases. Further discussion of this optimization technique is presented in **SECTION 5.4 Optimization Techniques**.

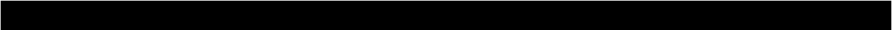
## 5.2.2 Initialization

The initialization subroutine accomplishes three main tasks:

1. initializing the DSP56001
2. initializing program variables and lookup tables
3. initializing data buffer pointers and modulo registers

This subroutine also includes any I/O interface configuration that is necessary. Figure 5-2 shows the memory map of the internal data RAM. The encoder and decoder algorithms each require several variables to be stored in memory. The DSP56001 code also requires several other temporary storage locations. Most of the variables used by the algorithm are stored in data memory below address \$40 so that the short immediate addressing mode can be used when accessing them. Data locations that are accessed with an addressing register are stored at higher locations in data memory since they do not need to use immediate addressing modes.

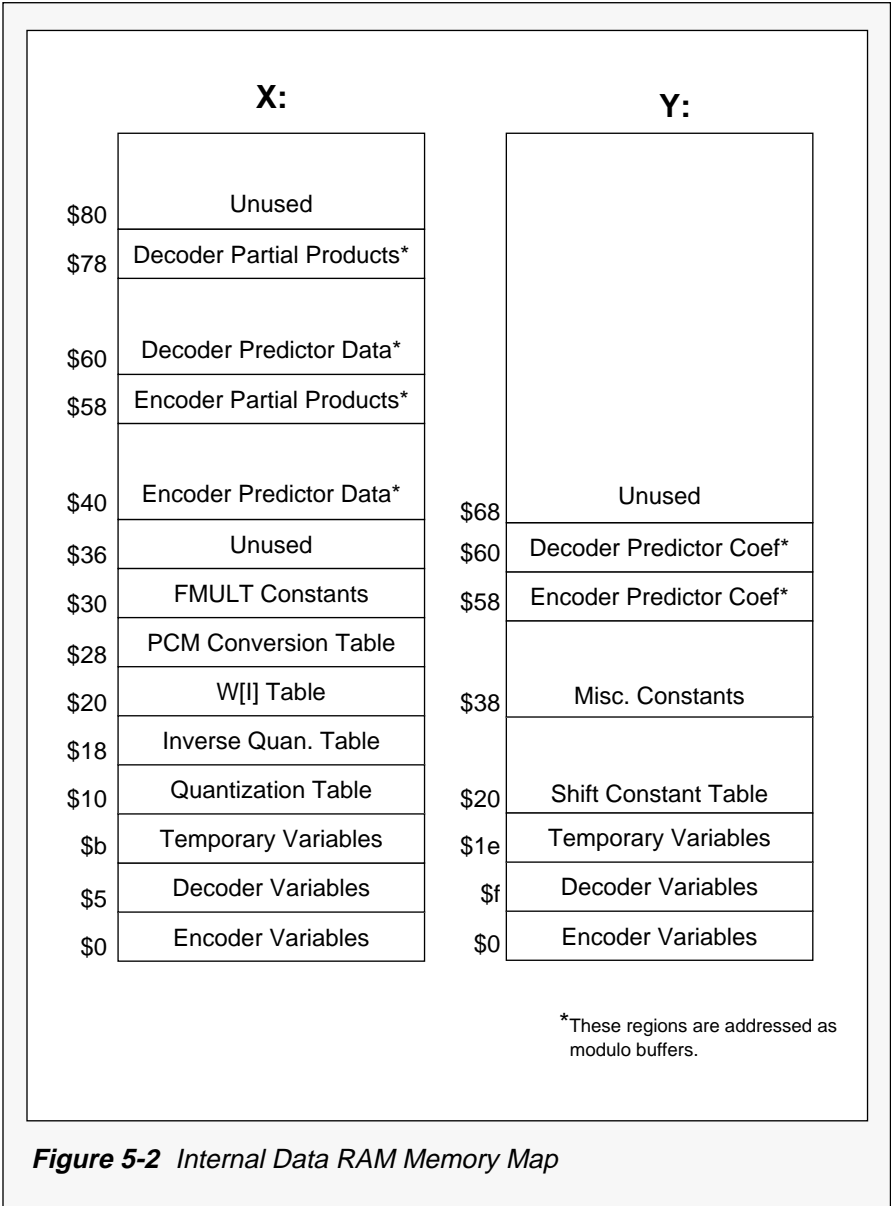
The DSP56001 initialization consists of enabling the on-chip, factory programmed data ROM tables and setting the DSP56001's Bus Control Register (BCR) for zero wait state external program memory access. The  $\mu$ -law and A-law ROM table in X memory is



needed for the log PCM conversion routine. The zero wait states for external program memory are needed so the algorithm will run in real time.

The program initialization is done in the subroutine INIT. First, all internal X and Y data RAM is cleared, all variables that require specific values are initialized, and then all lookup tables are copied from their load-time locations in program memory to their run-time locations in data RAM. Next, the pointers to the receive (decode) and transmit (encode) data buffers are initialized. These buffers hold the delayed values of  $d_q(k)$  and  $s_r(k)$  used in the linear predictor filter. These are the only true modulo buffers used in the assembly code in the sense that the newest delayed values replace the oldest delayed values without actually moving the other delayed values. The INIT routine initializes the sign and mantissa locations in these buffers since the code assumes a certain range of legal values in these locations. The INIT routine also initializes other variables including the variable LAW. It determines whether the  $\mu$ -law or A-law format is chosen. The program defaults to setting LAW to zero to select  $\mu$ -law for the PCM format. The code can be changed to set LAW to any non-zero value which will select the A-law format. It can also be easily modified to select  $\mu$ -law or A-law based on an external input.

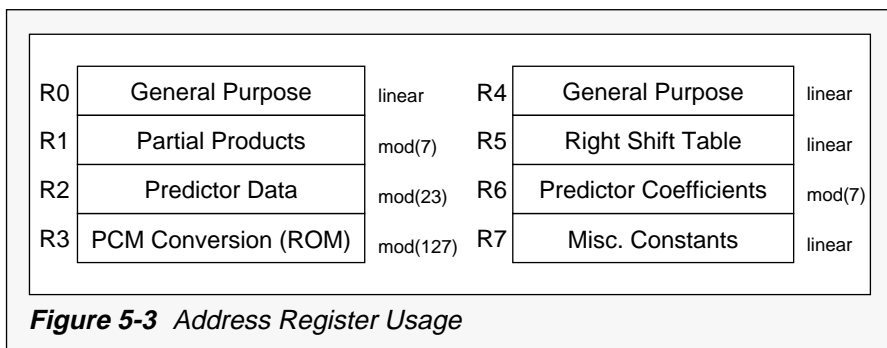
The remaining portions of the INIT routine initialize the addressing registers and modifier registers that are used to access the data memory. Six of the variable storage areas (indicated by \* in Figure 5-2) are addressed using modulo pointers. These six areas, three each for the encoder and decoder, are used by the linear predictor filter.



**Figure 5-2** Internal Data RAM Memory Map

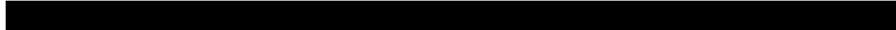


The complete addressing register assignments along with the associated addressing modes are shown in Figure 5-3. Six of the addressing registers are reserved for particular functions, and the remaining two are used for other general purpose tasks requiring addressing registers. The INIT routine can also contain any initialization needed for a hardware interface such as the SSI port. No initialization is needed to use the file I/O routines on the ADS so I/O initialization is not included in the standard code.



### 5.2.3 PCM Format Conversion

The ADPCM algorithm uses several different types of numeric formats. Conversion between these formats is required in several places in the assembly code. The log PCM format conversion is one of these instances. Two different routines are used for converting between  $\mu$ -law or A-law PCM samples and linear (uniform) samples. EXPAND converts an



8-bit log PCM sample to a linear 14-bit two's-complement representation that is suitable for numeric operations. COMPRESS performs the opposite conversion. The DSP56001 ADPCM implementation supports both  $\mu$ -law and A-law format conversion.

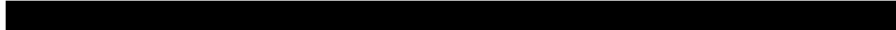
The routine EXPAND performs the conversion by using the internal  $\mu$ -law and A-law ROM tables on the DSP56001. EXPAND uses register r3 as the pointer into the lookup table. Register r3 is set during program initialization to either the  $\mu$ -law table base or the A-law table base and remains set to this value while the program is running. Register n3 is used as an offset into the ROM table. The assembly code for EXPAND is identical for both formats and is only dependent on the base pointer stored in r3 so separate routines are not needed. To obtain optimal speed, the COMPRESS routine requires separate code sections for the  $\mu$ -law and A-law conversion. In this routine, the variable LAW is tested for each sample. If only one format is used, the variable LAW may be eliminated and one section of COMPRESS may be removed to save program memory. The SYNC routine discussed in **SECTION 5.2.11** also requires separate code segments for each log PCM format, so similar program memory savings can be obtained there.

The EXPAND and COMPRESS routines are modified versions of routines given in the Motorola applications brief "Logarithmic/Linear Conversion Routines for the DSP56000/1" [6]. They were chosen based on maximum execution speed. Complete descriptions of each routine can be found in the above applications brief.

## 5.2.4 Logarithmic Conversion

Another conversion required in the ADPCM algorithm is between a linear and a base 2 logarithmic format. The quantizer and the inverse quantizer achieve their adaptive characteristic by use of a scale factor. The scale factor itself and many of the variables used to calculate it are in a base 2 logarithmic form. The total number of bits used for these variables differs but they all share a common form of a mixed number. The numeric operations are performed on these numbers assuming an integer exponent portion combined with a fractional mantissa portion in one mixed number. The scale factor adaptation does not require a specific conversion to this format but a conversion is required in the quantizer and inverse quantizer. The routine LOG in the quantizer converts the difference signal to the base 2 logarithmic form so that it can be modified by the scale factor. In the inverse quantizer, the routine ANTILOG converts the quantized difference signal back to the linear form after it has been readjusted by the scale factor. In fact, the ADPCM codeword is based on the log of the difference signal  $d(k)$  rather than the difference signal itself.

Figure 5-4 illustrates the conversion process in a version of the LOG routine. The input D is in a 16-bit two's complement format. The first step is to convert this number to a sign magnitude representation saving the sign in register y1. After checking the magnitude for a non-zero value it is then normalized to determine the exponent and mantissa. The iterative NORM instruction is used for this conversion. This instruction will shift the magnitude left one bit for



each iteration until a 0 is in bit 23 and a 1 is in bit 22 (this is the normalized fraction format on the DSP56001). For each left shift the value in r0 is decremented. Once the magnitude has been normalized successive iterations will do no further adjustments. Fourteen iterations are performed since the maximum that the magnitude can be shifted is 14 bits, assuming the magnitude is non-zero. Fourteen iterations of the NORM is not needed in all cases but taking time to test after each iteration would cause the worst case delay to be longer. After the normalization process is finished the normalized mantissa will be in accumulator *a* and the associated exponent will be in register r0.

The remaining instructions combine the exponent and mantissa into a mixed number. The truncation of the mantissa to seven bits is performed by using a mask instead of actually shifting. This technique is common throughout the code. The process of combining the exponent and mantissa also shows the technique of shifting by multiplication. The exponent is moved from r0 to x1 where it will be in the four LSBs but it needs to be left justified to bits 22-19 which are the four MSBs of the DSP56001's fractional format. A shift constant is read from the shift constant table in Y memory and is multiplied with the exponent. The result is that the exponent is effectively shifted left 19 bits. A shift constant is also used to shift the mantissa right by three bits. In this example the mantissa is shifted right and combined with the exponent in a single MAC instruction. This shift technique is described in further detail in **SECTION 5.4**. The resulting log signal DL is a mixed number with

four integer bits and seven fractional bits with an implied radix point. Note that this logarithmic format is similar to the mixed number format discussed in [9].

```

;*****
; LOG
;
; Convert difference signal from the linear to the log domain
;
; Input: D = siii iiii | iiii iiii. | 0000 0000 (16TC) in accum A
;
; Outputs:
; DL   = 0iii i.fff | ffff 0000 | 0000 0000 (1LSM) in accum A
; DS   = sXXX XXXX | XXXX XXXX | 0000 0000          (1TC) in Y1
;
;*****

        MOVE     #$000E, R0                ;Get exp bias (14)
        MOVE     X:Y_T,B                  ;Get Y
        ABS      A A,Y1                    ;Find DQM=|D|, save DS to Y1
        JNE      <NORMEXP_T               ;Check for DQM=0
        CLR      A (R7) +                  ;If DQM=0 set DL=0
        JMP      <SUBTB_T

NORMEXP_T
        REP      #14                        ;If DQM!=0, do norm iteration
        NORM     R0, A                      ;14 times to find MSB of DQM

; A1 = 01?? ???? | ???? ???? | 0000 0000 = normalized DQM (A2=A0=0)
; R0 = 0000 0000 | 0000 eeee = exponent of normalized DQM

; Get rid of leading "1" in normalized DQM
; Truncate mantissa to 7 bits and combine with exponent

        MOVE     Y:(R7)+,X1                ;Get mask K6 ($3F8000)
        AND      X1,A      Y:LSHFT-19,X0  ;Truncate MANT, get EXP shift

; A1 = 00mm mmmmm | m000 0000 | 0000 0000 (A2=A0=0)

        MOVE     R0,X1                      ;Move EXP to X1
        MPY     X0,X1,A      A,X1 Y:(R7) +,Y0 ;Shift EXP<<19,save MANT to X1,
                                                ;get mask K7 ($100000)
        MOVE     A0,A                        ;Move EXP to A1

; X1= 00mm mmmmm | m000 0000 | 0000 0000
; A1= 0eee e000 | 0000 0000 | 0000 0000 (A2=A0=0)

        MAC     Y0,X1,A                      ;shift MANT>>3 & combine with EXP

; A1= 0eee e.mmm | mmmmm 0000 |0000 0000 (A2=A0=0)
; = 0iii i.fff | ffff 0000 |0000 0000 (A2=A0=0)

```

**Figure 5-4** Linear to Log Conversion Routine



The routine ANTILOG in the inverse quantizer performs the opposite conversion. It does so by splitting the exponent and mantissa apart and then shifting the mantissa right again according to the exponent.

### 5.2.5 Floating-Point Conversion

The other type of conversion required in the ADPCM algorithm is a floating-point conversion. The CCITT specifies that the multiplications in the linear predictor filter be done in a specific floating-point format. After each multiplication, the result must be converted back into fixed-point format before it is accumulated with the other partial products. The data inputs to the predictor filter are the delayed values of  $d_q(k)$  and  $s_r(k)$ . For each input sample, these values are converted to floating-point and then stored in this form so they do not have to be converted again. The coefficients of the predictor filter  $a_i(k)$  and  $b_i(k)$  are updated in fixed-point form for each sample so they must be converted from linear to floating-point form for each sample. Since there are six zeros and two poles in the predictor filter, the overall requirement is ten fixed-point to floating-point conversions and eight floating-point to fixed-point conversions. Clearly, this conversion process has a major impact on the execution speed of the overall algorithm, so this process must execute as fast as possible. Sixteen of these conversions are performed in the FMULT routine and the other two are performed in the FLOATA and FLOATB routines so that much of the speed emphasis is placed

on the FMULT routine. Again, the overall goal is the minimum worst case execution time.

The floating-point format used in this algorithm consists of four exponent bits, six mantissa bits (with an explicit leading 1), and one sign bit for a total of eleven bits (11FL). As mentioned, the values of  $d_q(k)$  and  $s_r(k)$  are stored in the floating-point format.

```

;*****
; FLOATA
;
; Converts the quantized difference signal from 15-bit signed magnitude to
; floating pt. format (11FL - sign, exp, and mant stored separately)
;
; Inputs:
; DQ = siii iiii | iiii iii.0 | 0000 0000 (15SM) in accum A
;
; Outputs:
; DQ0 = (11FL)
; DQ0EXP = X:(R2) = 0000 0000 | 0000 0000 | 0000 eeee
; DQ0MANT = X:(R2+1) = 01mm mmm0 | 0000 0000 | 0000 0000
; DQ05 = X: (R2+2) = sXXX XXXX | XXXX XXXX | 0000 0000
;
;*****
; R2 points to predictor data buffer - DQ0 will overwrite previous SR2

        MOVE     X:DQ_T, Y0           ;Get DQS
        MOVE     Y:DQMAG,A           ;Get MAG=DQMAG
        TST      A #$000E,R0         ;Check MAG, get exponent bias (14)
        JNE     <NORMDQ_T           ;Test MAG
        MOVE     #<$40, A             ;If MAG=0 set MANT=100000,
        MOVE     #0,R0               ; and EXP=0
        JMP      <TRUNCdq_T

NORMDQ_T
        REP      #13                 ;If MAG!=0 do NORM iteration 13
        NORM    R0,A                ; times to find MSB of MAG

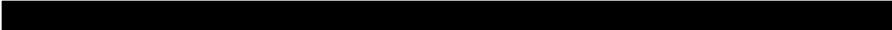
; A1 = 01?? ???? | ???? ???0 | 0000 0000 = normalized MAG (A2=A0=0)
; R0 = 0000 0000 | 0000 eeee = exponent of normalized MAG

TRUNCdq_T  MOVE     #<$7E,X0         ;Get mask
           AND      X0,A R0,X:(R2)+ ;Truncate MANT to 6 bits,
           ; save EXP to DQ1EXP

; A1 = 01mm mmm0 | 0000 0000 | 0000 0000 (A2=A0=0)
           MOVE     A1,X:(R2)+       ;Save MANT to DQ1MANT
           MOVE     Y0,X:(R2)+       ;Save DQ to DQ1S

```

**Figure 5-5** Linear to Floating-Point Conversion Routine



The coefficients do not need to be stored in this format since they are used immediately after they are converted in the FMULT routine. Further details of the FMULT routine are given in the adaptive predictor section, however, a version of FLOATA is shown in Figure 5-5 to illustrate the conversion process. Clearly, the process of normalizing the input value is very similar to that in the logarithmic conversion routine LOG. The main difference is that the sign, exponent, and mantissa components are not combined but are stored separately in the data buffer, using register r2 as a pointer. Note that this storage form was chosen because it requires less data manipulation and shifting. Also notice that the complete value of the two's complement number is stored as the sign. Only the sign of the value is important so the sign does not need to be separated from the rest of the number.

### 5.2.6 Difference Signal Quantization

After the EXPAND conversion routine converts the input signal  $s(k)$  to the two's complement signal  $s_1(k)$ , the routine SUBTA subtracts the signal estimate  $s_e(k)$  from this value to form the difference signal  $d(k)$ . The computation in SUBTA only requires aligning radix points and subtracting. The adaptive quantization of  $d(k)$  is not as straightforward. To perform the quantization  $d(k)$  must be normalized by the scale factor  $y(k)$ . As noted previously, the scaling and quantization is performed in base 2 log format. This conversion in the routine LOG is described in **SECTION 5.2.4**.



After conversion the log signal  $d_l(k)$  is scaled in the routine SUBTB and quantized in the routine QUAN. These two routines are shown in Figure 5-6. SUBTB simply truncates the scale factor  $y(k)$  and then subtracts this value from  $d_l(k)$ . The quantization of this normalized value  $d_{ln}(k)$  in the QUAN routine is done by a table search. The boundary values of the eight quantization regions shown in Table 4-1 are stored in the table QUANTAB in data memory. These values are read from the table using register r0 as a pointer and then compared with  $d_{ln}(k)$  until the correct range is found. When the range is found, an offset from the starting address of QUANTAB is subtracted from the last value in r0. This process produces the correct magnitude of  $l(k)$  given in Table 4-1.

```

;*****
;
;          SUBTB
;
; Scale log version of difference signal by subtracting the scale factor
;
; DLN = DL - Y
;
; Inputs:
; DL = 0iii i.fff | ffff 0000 | 0000 0000 (11SM) in accum B
; Y = 0iii i.fff | ffff ff00 | 0000 0000 (13SM) in accum A
;
; Output:
; DLN = sii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
;
;*****

SUBTB_T  MOVE    Y:(R7),X0      ;Get mask K8 ($7FF000)
         AND     X0,B          ;Truncate Y to 11 bits (Y>>2)
         SUB     B,A           ;Find DLN = DL - Y

;*****

```

**Figure 5-6** Difference Signal Scaling and Quantization (sheet 1 of 2)

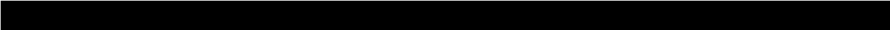
```

;      QUAN
; Quantize difference signal in log domain
;
;      log2 |D(k) - Y(k)| |I(k)|
;      -----|-----
;      [3.12, + inf) | 7
;      [2.72, 3.12) | 6
;      [2.34, 2.72) | 5
;      [1.91, 2.34) | 4
;      [1.38, 1.91) | 3
;      [0.62, 1.38) | 2
;      [-0.98, 0.62) | 1
;      (-inf, -0.98) | 0
; Inputs:
; DLN = siii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
; DS = sxxx xxxx | xxxx xxxx | 0000 0000 (1TC) in reg Y1
; Output:
; I = siii 0000 | 0000 0000 | 0000 0000 (ADPCM format) in accum A
;*****
; Quantization table in X memory
; QUANTAB DC $F89000 ;-0.98
;          DC $050000 ;0.62
;          DC $0B2000 ;1.38
;          DC $0F6000 ;1.91
;          DC $12C000 ;2.34
;          DC $15D000 ;2.72
;          DC $190000 ;3.12
;          DC $7FFFFFF ;15.99
;
;          MOVE #QUANTAB,R0 ;Get quantization table base
;          MOVE #>QUANTAB+2,X1 ;Get offset for quan. conversion
;          MOVE X:(R0)+,X0 ;Get 1st quan. table value
TSTDNL_T
;          CMP X0,A X:(R0)+,X0 ;Compare to DLN, get next value
;          JGE <TSTDNL_T ;If value<DLN try next range
;          MOVE R0,A
;          SUB X1,A Y:LSHFT-20,X0 ;When range found subtract pointer
; ; from base to get IMAG=II

; A1 = 0000 0000 | 0000 0000 | 0000 0iii (A2=A0=0)
;          MOVE A1,X1
;          MPY X0,X1,A Y1,B ;Shift IMAG <<20, result is
; ; in A0, move DS into B
;          MOVE A0,A
; A1 = 0iii 0000 | 0000 0000 | 0000 0000 (A2=A0=0)
;          MOVE A1,X:IMAG ;Save IMAG
;          TST A #<$F0,X0 ;Check IMAG, get invert mask
;          JEQ <INVERT_T ;If IMAG=0 invert bits
;          TST B ; else check DS
;          JPL <IOUT_T ;If DS=1 don't invert IMAG
INVERT_T EOR X0,A ;If DS=0 or IMAG=0 invert IMAG
IOUT_T MOVE A1,A ;Adjust sign extension

```

**Figure 5-6** Difference Signal Scaling and Quantization (sheet 2 of 2)



The magnitude of  $I(k)$  is shifted to the MSBs of register  $a1$  and is then combined with the sign value  $d_s(k)$  which was previously saved in register  $y1$  in the LOG routine. The ADPCM word  $I(k)$  is in a sign magnitude type format. If  $d_s(k)$  is negative then the four MSBs of the accumulator are inverted, setting the sign bit to 1 and inverting the magnitude bits. If the sign is positive the magnitude of  $I(k)$  is not changed leaving the sign set to 0. A special case occurs when the magnitude of  $I(k)$  is 0. In this case the bits are inverted even if the sign is positive. This means that an all zero word is not legal and will never be transmitted. This is why the quantizer is referred to as a 15-level quantizer. It should be noted however that transmission errors can cause an all zero word to be received by the decoder so this case must be taken into account in the inverse quantization.

## 5.2.7 Inverse Quantization

The inverse quantization of the ADPCM sample  $I(k)$  is performed in the routines RECONST and ADDA. These routines are shown in Figure 5-7. The RECONST routine uses a table lookup to find  $d_{lnq}(k)$  — the quantized version of  $d_{ln}(k)$ . After removing the sign of  $I(k)$  the magnitude is inverted if necessary and is then shifted to the three LSBs of the 24-bit word. This magnitude is then moved to the offset register  $n4$  where it is used as an offset to find one of eight values stored in the lookup table IQQUANTAB (defined in Table 4-1). The scale factor  $y(k)$  is added to the result to find the denormalized value  $d_{ql}(k)$ .

```

;*****
;
;          RECONST
;
; Reconstruct quantized difference signal in the log domain
;
;          |I(K)|      | log2 |DQ(k)| - Y(k)
;          -----
;          7          |      3.32
;          6          |      2.91
;          5          |      2.52
;          4          |      2.13
;          3          |      1.66
;          2          |      1.05
;          1          |      0.031
;          0          |      - inf
;
; Inputs:
;   I = iiii 0000 | 0000 0000 | 0000 0000 (ADPCM format) in accum A
;
; Output:
;   DQLN = siii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
;   DQS = sXXX 0000 | 0000 0000 | 0000 0000 (1TC) in reg Y1
;
;*****
;
; Inverse quantization table in X memory
;
; IQUNTAB   DC   $800000          ;-16      |I|=0
;           DC   $004000          ;0.031   |I|=1
;           DC   $087000          ;1.05    |I|=2
;           DC   $0D5000          ;1.66    |I|=3
;           DC   $111000          ;2.13    |I|=4
;           DC   $143000          ;2.52    |I|=5
;           DC   $175000          ;2.91    |I|=6
;           DC   $1A9000          ;3.32    |I|=7
;
;           MOVE   #<$F0,X1
;           MOVE   A,Y1           A,X:I_R           ;Save DQS (sign of I) to Y1
;           EOR    X1,A           Y:RSHFT+20,Y0     ;Invert bits of I
;           TMI    Y1,A           ;If ^IS=1 use I, else use ^I
; A1 = 0iii 0000 | 0000 0000 | 0000 0000
;           MOVE   A1,X0
;           MOVE   A1,X:IMAG           ;Save |I|
;           MPY   X0,Y0,A         #IQUNTAB,R4       ;shift IMAG>>20
; A1 = 0000 0000 | 0000 0000 | 0000 0iii (A2=A0=0)
;           MOVE   A1,N4           ;Load IMAG as offset into IQUAN table
;           MOVE   X:Y_R,B         ;Get Y
;           MOVE   X:(R4+N4),A      ;Lookup DQLN

```

**Figure 5-7** Inverse Quantization and Scaling of ADPCM Codeword  
(sheet 1 of 2)

```

;*****
;  ADDA
; Add scale factor to log version of quantized difference signal
;
; DQL = DQLN + Y
;
; Inputs:
; Y = 0iii i.fff | ffff ff00 | 0000 0000 (13SM) in accum B
; DQLN = siii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
; Output:
; DQL = siii i.fff | ffff 0000 | 0000 0000 (12TC) in accum A
;
;*****
      MOVE     Y:(R7)+,Y0      ;Get mask K8 ($7FF000)
      AND      Y0,B           ;Truncate Y to 11 bits (Y<<2)
      ADD      B,A            ;Find DQL=DQLN+(Y<<2)

```

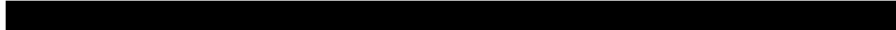
**Figure 5-7** *Inverse Quantization and Scaling of ADPCM Codeword*  
(sheet 2 of 2)

This logarithmic value is converted back into linear form in the routine ANTILOG to find the result of the overall inverse quantization procedure  $d_q(k)$ , the quantized version of the difference signal.

The quantization and inverse quantization procedures can serve as illustrations of one way of implementing an adaptive quantization in a waveform coder. The adaptation of  $y(k)$ , discussed in **SECTION 5.2.10**, addresses the adaptive characteristic of the scale factor but the scaling and quantization process described here can still be used no matter how the adaptation is performed.

### 5.2.8 Adaptive Predictor

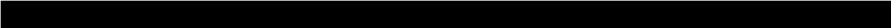
The adaptive predictor portion of the ADPCM algorithm is implemented in two main sections. The



first section is the prediction filter itself, shown in Figure 4-5. This section consists of the routines FMULT and ACCUM. The filter uses delayed data and coefficient values so FMULT and ACCUM are the first two routines executed in the encoder and the decoder. The second section consists of the reconstructed signal calculation and the adaptation of the predictor coefficients. These routines are executed after the inverse quantization in both the encoder and the decoder.

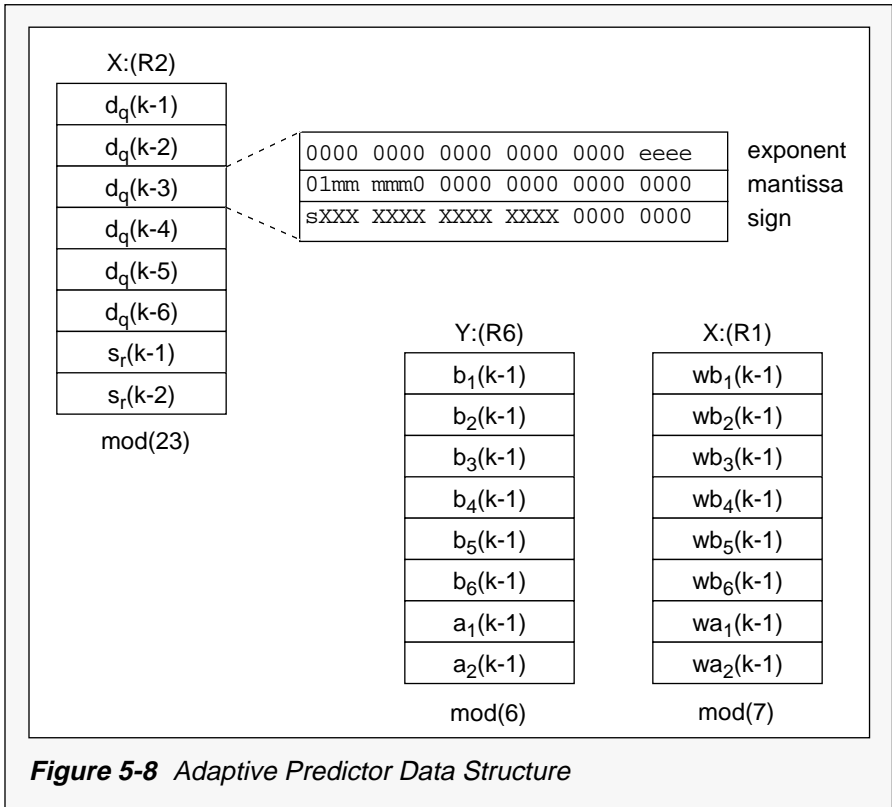
The adaptive predictor is the most computationally intensive portion of the ADPCM implementation on the DSP56001. One of the main reasons for this is the floating-point multiplies that are required in the FMULT routine, as was discussed in the floating-point conversion section. The FMULT routine is set up as a hardware DO loop that is executed eight times, two for the poles and six for the zeros. For each tap of the filter the two's-complement coefficient must be converted to the floating-point format before it is multiplied with the delayed data value. After the multiplication each partial product must be converted to the fixed point format. Overall eight fixed-point to floating-point conversions and eight floating-point to fixed point conversions are required in the FMULT routine. The flow description within each loop of FMULT is as follows:

1. Convert the 16-bit two's complement coefficient to a 13-bit magnitude and a 1-bit sign.
2. Convert the 13-bit magnitude to a 4-bit exponent and a 6-bit mantissa.

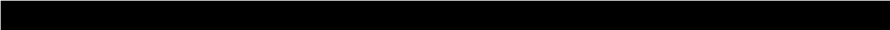
- 
3. Add the exponents of the coefficient and the data to find the 5-bit exponent of the partial product.
  4. Multiply the mantissas of the coefficient and the data and truncate the results to find the 8-bit mantissa of the partial product.
  5. Convert the exponent and the mantissa of the partial product to a 15-bit magnitude.
  6. Exclusive-OR the signs of the coefficient and the data to find the sign of the partial product.
  7. Convert the 15-bit magnitude and 1-bit sign of the partial product to a 16-bit two's complement number.

Since the FMULT routine requires a large percentage of the processing time of the overall algorithm it is desired to have the worst case execution speed of this routine as short as possible. To reduce data movement overhead, several constants that are needed by the routine are stored in the table CONST in data memory. These constants are addressed using register r0 so parallel moves can be taken advantage of whenever possible. This technique is discussed in **SECTION 5.4 Optimization Techniques**. Another aspect of this routine that has a significant effect on the speed is the way the coefficients, data, and partial products are stored in data memory. A description of this buffer structure is shown in Figure 5-8. The variables  $w_{a_i}(k)$  and  $w_{b_i}(k)$  in this figure represent the partial products for the pole and zero taps of the filter. To eliminate extra overhead the predictor data buffer that contains the delayed floating point values of  $d_q(k)$  and  $s_r(k)$  is

set up so that the sign, mantissa, and exponent are stored separately. Storing the data in this form uses more data memory but the routine requires much less computation when using these values than if they were combined into one word. The address register r2 is used as a pointer to this buffer so when an element is needed an address register addressing mode can be used. These modes allow the parallel bus structure of the DSP56001 to be taken advantage of whenever possible.








The data buffer structure also allows the efficient offset addressing modes of the DSP56001 to be used when consecutive values of one component are required. For instance, the signs of each delayed  $d_q(k)$  value are needed in the XOR/UPB routine. In this routine the offset register n3 is set to 3 so that when one sign is read from the buffer, register r3 is automatically post-incremented by 3 to point to the next sign. The modulo pointer feature of the DSP56001 also reduces execution speed since less software overhead is required to update the pointer. The data buffer for the prediction filter is actually a modified form of the usual modulo buffer structure on the DSP56001 since two new values,  $s_r(k-1)$  and  $d_q(k-1)$ , are added to the buffer for each sample. The new  $s_r(k-1)$  overwrites the current  $d_q(k-6)$  and the new  $d_q(k-1)$  overwrites the current  $s_r(k-2)$ . The coefficient buffer and partial product buffer are addressed using modulo pointers for efficiency but the physical locations of each component do not change.

The implementation of the ACCUM routine that adds the partial products is more straightforward than the FMULT routine. The  $wb_i(k)$  partial products for the six zeros are accumulated first to obtain the partial signal estimate  $s_{ez}(k)$ . Then the  $wa_i(k)$  partial products for the two poles are accumulated with the zeros to form the final signal estimate  $s_e(k)$ . Even though FMULT and ACCUM account for a large percentage of the overall execution speed they only implement Eqn. 4-2 and Eqn. 4-3 of the CCITT algorithm.

The second section of the adaptive predictor code is executed after the inverse quantizer. Calculating Eqn. 4-4 to determine the reconstructed signal  $s_r(k)$  is done in the routine ADDB. This routine requires a format conversion since  $d_q(k)$  is in a sign magnitude format. Following the ADDB routine is the adaptation of the predictor coefficients which includes Eqn. 4-5 through Eqn. 4-12. Due to the way the specification is written the implementation of Eqn. 4-5 through Eqn. 4-8 to update  $a_1(k)$  and  $a_2(k)$  is slightly more complicated than Figure 4-6 and Figure 4-7 indicate. The ADDC routine basically implements Eqn. 4-7 but it also calculates part of Eqn. 4-5 and Eqn. 4-6 that use the signal  $p(k)$ . The routine first calculates  $p(k)$  and then saves it to memory after delaying the previous values of  $p(k)$  and  $p(k-1)$ . The routine then calculates the variables PKS1 and PKS2 that represent the multiplication of the  $\text{sgn}[x]$  functions in Eqn. 4-5 and Eqn. 4-6. The  $\text{sgn}[x]$  function that is calculated in this routine does not have the values shown in Eqn. 4-12. Instead a 0 in the sign bit represents a positive number while a 1 in the sign bit represents a negative number. The variable SIGPK is used to distinguish the special case of  $\text{sgn}[p(k)] = 0$ . The routine UPA1 calculates Eqn. 4-5 in a slightly rearranged form:

$$a_1(k) = a_1(k-1) - [2^{-8} \cdot a_1(k-1)] + \text{gain} \quad \text{Eqn. 5-1}$$

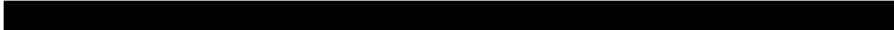
The gain portion of Eqn. 5-1 represents the second half of Eqn. 4-5 and is determined by testing the variables PKS1 and SIGPK. The routine UPA2 cal-



calculates Eqn. 4-6 to update  $a_2(k)$  in a similar manner. This routine is more complicated since it also has to calculate the value of  $f(a_1)$  in Eqn. 4-8. After updating  $a_1(k)$  and  $a_2(k)$ , the routines LIMD and LIMC limit these values. The routine LIMC uses a constant upper and lower limit defined in Eqn. 4-9 for the comparison. It makes use of the conditional transfer instructions of the DSP56001 as discussed in **SECTION 5.4 Optimization Techniques**. The LIMD routine is very similar but it must calculate the upper and lower limits before the comparison.

The  $b_i(k)$  coefficients are updated in the routines XOR and UPB which are combined in a single code segment. The XOR routine accounts for the  $\text{sgn}[x]$  multiplication in Eqn. 4-11 that uses the delayed values of  $d_q(k)$ . The calculation of Eqn. 4-11 is similar to that of Eqn. 4-5 and Eqn. 4-6 except that the calculation must be done six times for each of the six  $b_i(k)$  coefficients. In this routine the special case of  $\text{sgn}[d_q(k)] = 0$  is checked only once since it applies to all six calculations. If this case is found a DO loop that does not add the second half of Eqn. 4-11 is executed. If this case is not present a DO loop that does the full calculation is executed, but execution time is saved since the test for the special case does not have to be performed for each stage of the loop.

After the coefficients have been updated they are passed through the predictor trigger routine. Since this routine also affects the tone detect signal  $t_r(k)$ , they are all updated at once in the tone detection section of the algorithm. The final step of the adaptive prediction section is the conversion of the



quantized difference signal  $d_q(k)$  and the reconstructed signal  $s_r(k)$  to the floating-point format. This is done in the routines FLOATA and FLOATB respectively. After these values have been converted they are stored in the predictor data buffer. Further details of the floating-point conversion are found in **SECTION 5.2.5 Floating Point Conversion**.

### 5.2.9 Tone Detection

The tone detection portion of the ADPCM algorithm is implemented in three routines; TONE, TRIGB, and TRANS. The TONE routine, executed after the predictor adaptation, checks the pre-limited version of the coefficient  $a_2(k)$  for a threshold. If it is below that threshold the tone detection variable  $t_d(k)$  is set to 1. The TRIGB routine affects the signal  $t_d(k)$  as well as the predictor coefficients.

TRIGB tests the transition detect signal  $t_d(k)$ , and if it is set then the tone detect signal and all of the predictor coefficients are set to zero. The transition detect signal  $t_d(k)$  is set in the TRANS routine. This routine occurs after a delay block so it is executed after the inverse quantizer (since it needs the value of  $d_q(k)$ ) but before the predictor adaptation. The TRANS routine first checks the delayed signal of  $t_d(k)$ . If  $t_d(k)$  is set indicating a tone is present, TRANS compares the values of  $d_q(k)$  and  $y_l(k)$  according to Eqn. 4-23 to detect a transition from a tone to a non-stationary signal. The comparison requires a format conversion since  $d_q(k)$  is in sign magnitude format and  $y_l(k)$  is in logarithmic format. If the comparison threshold is met then  $t_d(k)$  will be set to drive the adaptation to the fast mode immediately.

### 5.2.10 Scale Factor Adaptation

The scale factor adaptation consists of two main sections, the adaptation of the scale factor, shown in Figure 4-9, and the speed control parameter adaptation, shown in Figure 4-10. The adaptation of the speed control parameter comprises seven routines. The first routine, FUNCTF, maps the magnitude of the ADPCM sample to one of four values specified in Table 4-2. The next two routines, FILTA and FILTB, use the result of this mapping,  $F[l(k)]$ , to track a short term and a long term average of the difference signal. As was the case with the adaptation of the predictor coefficients, the FILTA routine implements Eqn. 4-18 in a slightly different form:

$$d_{ms}(k) = d_{ms}(k-1) + 2^{-5} [F[l(k)] - d_{ms}(k-1)] \quad \text{Eqn. 5-2}$$

The FILTB routine implements Eqn. 4-19 in a similar manner using  $d_{ml}(k)$ . The next three routines implement Equation (4-20) that determines the unlimited speed control parameter  $a_p(k)$  from these averages and other inputs. Eqn. 4-20 sets  $a_p(k)$  to one of three values. The SUBTC routine checks for the three cases in which the factor of  $2^{-3}$  is added. SUBTC sets the variable AX to 1 if either of these three conditions is met, otherwise it is set to 0. The routine FILTC calculates the factor  $(1-2^{-4}) a_p(k-1)$  and then adds AX to this value. Notice that AX is actually set to a value that can be added directly so that extra shifting is not required. The following routine, TRIGA, tests the final condition based on a

transition detection. If  $t_r(k)$  is equal to 1 then the value of  $a_p(k)$  is set to 1, otherwise TRIGA leaves  $a_p(k)$  as the value set by FILTC. The final routine, LIMA, limits the delayed value of  $a_p(k)$  according to Eqn. 4-21 to form the final speed control parameter  $a_l(k)$ . Since LIMA uses the delayed value of  $a_p(k)$  this routine is executed after FMULT and ACCUM routines near the beginning of the algorithm. The output of LIMA,  $a_l(k)$  is fed directly to the scale factor adaptation routine MIX.

The update of the scale factor  $y(k)$  is performed in five routines that are executed immediately following the speed control parameter update. The first of these routines, FUNCTW, performs a mapping of  $l(k)$  that is based on Table 4-2. It is similar to the FUNCTF routine. The output of FUNCTW is the signal  $W[l(k)]$  that is used to update the unlocked scale factor  $y_u(k)$ . This update is performed in the FILTD routine. It implements Eqn. 4-14 in a rearranged form that is similar to that of Eqn. 5-2. This value is limited in the routine LIMB according to Eqn. 4-15. The limited value of  $y_u(k)$  is also used in Eqn. 4-16 to update the locked scale factor  $y_l(k)$ . This is done in the routine FILTE. The final update of the scale factor  $y(k)$  is done in the routine MIX. Since this follows a delay block, it is performed immediately after LIMA near the beginning of the algorithm. The MIX routine combine the three inputs  $y_u(k-1)$ ,  $y_l(k-1)$ , and  $a_l(k)$  according to Eqn. 4-17. Like previous equations it is executed in a slightly different form:

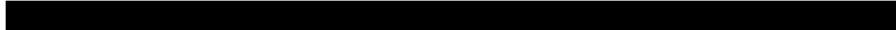
$$y(k) = a_l(k) [y_u(k-1) - y_l(k-1)] + y_l(k-1) \quad \text{Eqn. 5-3}$$

### 5.2.11 Decoder Synchronization

As mentioned in **SECTION 4.2 The CCITT Decoder Algorithm**, the decoder is almost identical to the encoder. The differences include the log PCM conversion and the synchronization block shown in Figure 4-13. The log PCM conversion routine COMPRESS is discussed in **SECTION 5.2.3**. Most of the synchronization section is identical to the first section of the encoder. The routines EXPAND, SUBTA, LOG, and SUBTB are the same as those discussed previously. The SYNC routine is similar to the QUAN routine but instead of transmitting the resulting ADPCM word, it compares it to the received ADPCM word. If they are the same then  $s_p(k)$ , the log PCM value of the reconstructed signal, becomes the output of the decoder,  $s_d(k)$ . If they are not the same then the next more positive or negative PCM word becomes the output as described in Eqn. 4-25. Most of the computation in the SYNC routine is used to check the boundary cases of the PCM word. Also, it has two separate code sections, one for the  $\mu$ -law case and the other for the A-law case.

## 5.3 Non-Standard Implementation

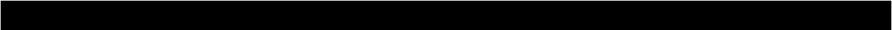
The main objective of the non-standard ADPCM program is to provide a more straightforward and efficient implementation of the ADPCM algorithm than the CCITT standard specification allows. The non-standard code implements the same algorithm



as the standard version but eliminates many of the details described in Recommendation G.721. Two key features have been removed; one being the use of floating-point multiplies in the adaptive prediction filter. Instead, the non-standard code implements the filter in fixed-point arithmetic using the 24-bit multiplier and 56-bit accumulator on the DSP56001. The removal of this feature alone greatly improves the execution speed of the code and also reduces the code complexity. Another key feature of the standard ADPCM code that was removed is the synchronous coding adjustment block in the decoder. As noted previously, this block is included to prevent cumulative distortions when multiple ADPCM encodes and decodes are performed on the same channel. Since many applications do not require this block it has been removed in the non-standard version. Other minor details have also been removed to make individual routines as efficient as possible.

In addition to improving the execution speed of the algorithm, the non-standard version also reduces memory size requirements. The standard ADPCM code is not able to implement code segments as subroutines because of execution speed requirements. Since the execution speed of the non-standard routine is much faster, code segments that are common to the encoder and the decoder can be shared. This feature allows the run-time portion of the ADPCM algorithm to fit into the DSP56001's on-board program RAM, so that no high-speed external RAM is required for real-time operation of a single full-duplex channel. Even with





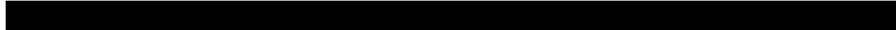
the extra overhead of subroutines, real-time performance can still be obtained on a 20 MHz DSP56001. Further performance details are given in **SECTION 5.5 Performance Specifications**.

One objective of the non-standard version is to provide a direct, readable implementation of the algorithm that can be easily modified. Many optimization techniques used in the standard implementation make modifying a part of the algorithm difficult without affecting other parts of the code. The non-standard code is written so that the subroutines are as independent as possible. Even within subroutines the individual ADPCM routines are separated so that they do not overlap.

This section provides further details that are particular to the non-standard implementation. Since the basic form of the code is very similar to the standard version, this section focuses on differences between the standard and the non-standard code.

### **5.3.1 Code Structure**

Like the standard code, the non-standard program is organized as two main routines; the encoder and the decoder, plus an initialization subroutine. The encoder and decoder are still implemented as independent code segments, but the structure of the code within the encoder and decoder routines differs from the standard version. The most noticeable difference is that the ADPCM routines are coded as a set of subroutines that are shared by the encoder and the decoder. This requires more processor



overhead to call the routines, but it provides much greater savings of program memory. The savings are increased if multiple encodes and decodes are used. The encoder and decoder are still organized as independent segments even though they share common subroutines.

The execution flow of the algorithm blocks within the non-standard code is similar to the one shown in Figure 5-1 with some exceptions. The exceptions are:

- the synchronous coding adjustment in the decoder has been removed
- the PCM output occurs immediately after the reconstructed signal calculation

For this version, only registers r0, r2, r3, and r6 are reserved and each ADPCM routine functions efficiently as possible as an independent routine. Several of the optimization techniques used in the standard version were removed to keep the routines independent. Many of these optimization techniques, discussed in **SECTION 5.4**, can be added to the non-standard code to improve the execution speed.

### **5.3.2 Initialization**

The initialization subroutine on the non-standard code performs the same basic tasks as the standard version. Since the non-standard version has a real-time I/O interface, the INIT routine also includes initialization of the SSI port used for the PCM interface and the general-purpose I/O pins used for the ADPCM interface.

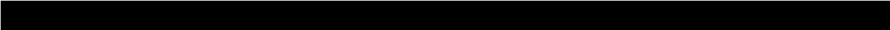
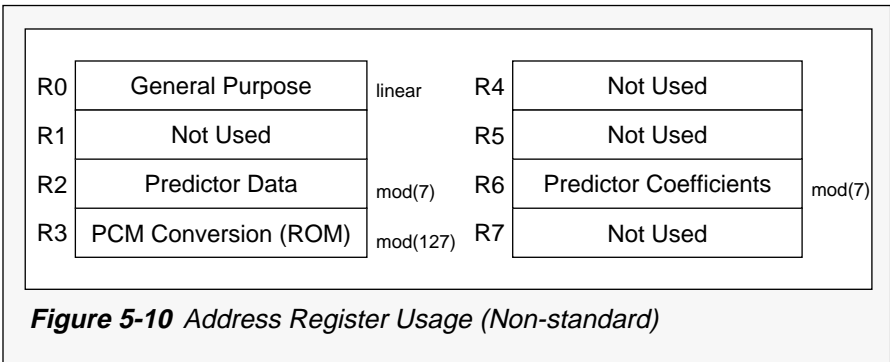
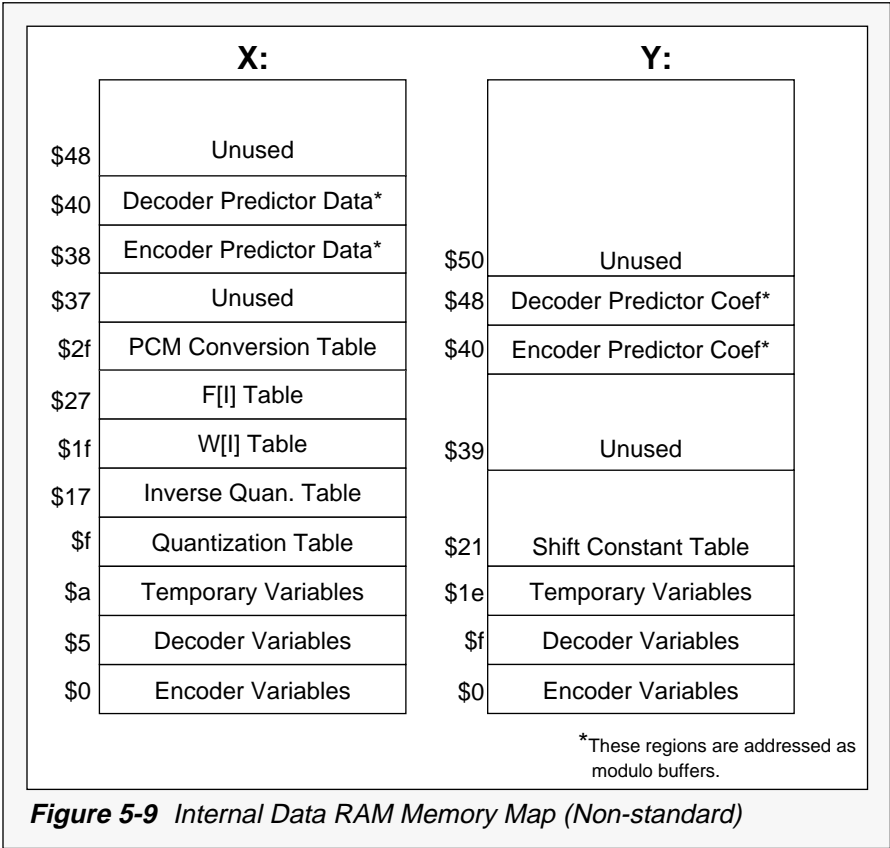


Figure 5-9 shows the memory map of the internal data RAM. In general, the arrangement of the variables and lookup tables is similar to the standard version. A notable difference between the two is the total storage requirement. A reason for this is that the predictor data buffers no longer require data to be stored in floating point format, so each buffer is only eight words long instead of 24. Also, since the predictor filter is implemented as one routine, the partial products of each multiply no longer need to be stored. Another reduction results from the elimination of the FMULT constant storage and the miscellaneous constant storage. Most of these constants are no longer needed since many are related to details of the CCITT specification. To reduce code complexity the remaining miscellaneous constants are stored as immediate data in program memory. Another difference from the standard version is the addition of a lookup table for the F[] variable calculated in the routine FUNCTF.

Figure 5-10 shows how the non-standard version uses addressing registers. Registers r2, r3, and r6 function the same as in the standard code. The predictor data buffer is now addressed as a modulo 7 block instead of a modulo 23 block since the floating point format is no longer used. The ADPCM code no longer uses registers r1, r4, r5, and r7. Only register r0 is retained as a general purpose register. The use of register r3 can be removed if desired since the algorithm uses it only in the EXPAND routine. Register r0 can be used in this routine instead but it must be set to the correct PCM table base, based on the variable LAW, each time the CONVERT subroutine is called.

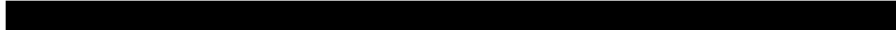


### 5.3.3 Format Conversions

The standard and non-standard programs use several different numeric formats. As noted previously, the change having the most major impact is the removal of the floating-point format. The floating-point conversions used in three routines, FMULT, FLOATA, and FLOATB were eliminated. The FMULT routine was combined with the ACCUM routine to form one common routine. The FLOATB routine that converts the reconstructed signal to floating-point in the standard version simply writes it directly to the predictor data buffer in the non-standard version. The floating-point conversion in the FLOATA routine that converts the quantized difference signal has also been removed, but this routine must still do a conversion from the sign magnitude format to the two's-complement format before the value is stored in the data buffer.

The size of variables is another main difference between the standard and the non-standard versions that has a large impact on the code. The full 24/56-bit precision of the DSP56001 is used whenever possible. The truncation of data values to shorter lengths that is specified in the CCITT document in many places is avoided when unnecessary. Not only does this improve the precision of calculations, it also leads to a more efficient implementation.

The other numeric format conversions used in the non-standard ADPCM code, the PCM and logarithmic conversions, are basically the same as in the standard version. The full A-law and  $\mu$ -law PCM



conversion routines are provided in the code, but since the ADPCM algorithm itself operates on linear input data these routines can be removed if an interface to linear data is desired. The logarithmic format must still be retained since it is inherent in many of the ADPCM equations. The quantization and inverse quantization blocks are also basically the same as in the standard version.

### **5.3.4 Adaptive Predictor**

The adaptive predictor filter in the non-standard code is implemented as a single routine called FMULT/ACCUM as shown in Figure 5-11. This routine is a much simpler implementation of the filter than the FMULT and ACCUM routines used in the standard version. The entire filter requires only nine words of program memory compared to the 60 words required by the standard version. This routine is shared by the encoder and the decoder in the non-standard version so the code savings are actually doubled.

In addition to improved execution speed and reduced program memory size, this implementation also reduces data memory size. The data storage structure is shown in Figure 5-12. Since the FMULT and ACCUM routines are combined the partial products do not need to be stored. This eliminates the need for the partial product buffers. Additional data memory is saved since the delayed reconstructed signal values and the delayed quantized difference signal values do not need to be stored in

floating-point format. Instead these values are stored in the predictor data buffer as 24-bit two's-complement numbers. Also note that the predictor coefficients have been extended to 24 bits to make full use of the data sizes on the DSP56001.

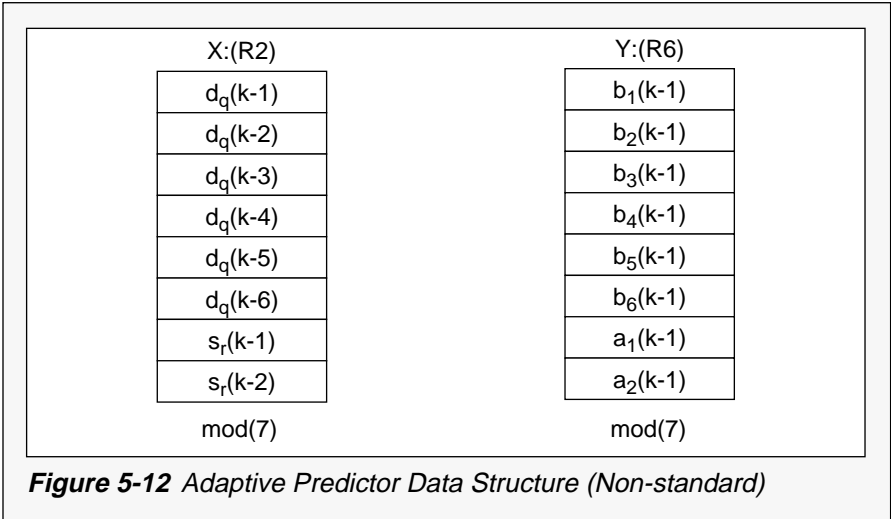
```

;*****
;          FMULT/ACCUM
;
; Perform adaptive prediction filter using 29-bit fixed point
; multiply and 56-bit accumulate
;
; SEZ(k) = [B1(k-1) * DQ(k-1)] + ... + [B6(k-1) * DQ(k-6)]
;          = WB1 + WB2 + ... + WB6
;
; SE(k) = SEZ(k) + [A1(k-1) * SR(k-1)] + [A2(k-1) * SR(k-2)] = SEZ + WA1 + WA2
;
; Inputs:
;      SRn = X:(R2) = siii iiii | iiii iiii. | ffff ffff (24TC)
;          (DQ in same format as SR)
;
;      An = Y:(R6) = si.ff ffff | ffff ffff | ffff ffff (24TC)
;          (Bn in same format as An)
;
; Outputs:
;      SEZ = siii iiii | iiii iiii. | ffff ffff (24TC)
;      SE  = siii iiii | iiii iiii. | ffff ffff (24TC)
;*****

PRDICT  CLR  A           X:(R2)+,X0   Y:(R6)+,Y0   ;Get DQ1 & B1
        REP  #6
        MAC  X0,Y0,A     X:(R2)+,X0   Y:(R6)+,Y0   ;Find SEZ
        TFR  A,B
        ASL  B
        MAC  X0,Y0,A     X:(R2)+,X0   Y:(R6)+,Y0   ;Adjust radix pt.
        MAC  X0,Y0,A
        ASL  A           ;Accum, get SR2 & A2
        RTS             ;Find SE
                    ;Adjust radix pt.

```

**Figure 5-11 Adaptive Prediction Filter**

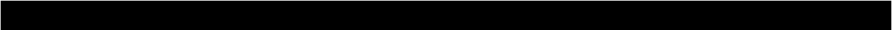


The update of the predictor filter coefficients is largely the same as in the standard version except that the XOR/UPB routine has been modified to use less program memory. Instead of using two separate loops, (one with the exclusive-or calculation and one without), only one loop is used because the XOR function is done for each coefficient.

## 5.4 Optimization Techniques

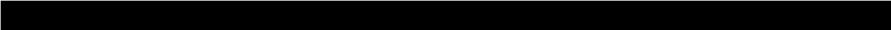
The implementation of the standard CCITT ADPCM coder on the DSP56001 uses several optimization





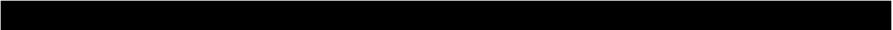
techniques to obtain real-time performance. The goal is to obtain the minimum worst case execution speed for the entire algorithm. A description of these techniques serves as an aid to understanding the assembly code since many of these techniques make the code harder to follow. A key factor in optimizing any assembly code is complete knowledge of the part's architecture, both from a hardware and software standpoint; plus equal knowledge of the algorithm being implemented. Such knowledge is also helpful for the ADPCM example. The following is a list of optimization techniques used in the ADPCM version:

1. The algorithm does not use subroutines, as noted in **SECTION 5.2.1**. The overhead of passing parameters and calling subroutines requires a significant portion of the total execution time, prohibiting real-time performance. Eliminating subroutines also allows exploitation of the DSP56001's parallelism as much as possible since less data movement is required. The disadvantage of this approach is that more program memory is required to duplicate common routines. This can be a problem if more external memory is required but in the ADPCM code it was found that even with subroutines the amount of required code would not fit in the on-chip program RAM.
2. In this application memory is sacrificed whenever gains in execution speed can be obtained. This may not work in all cases depending on the memory configuration. A key factor in using this technique is the multiplexed external bus of the DSP56001. The external bus will allow one external access per instruction cycle (assuming zero wait states) with no



penalty in execution speed. In this application external data memory is not used so there will not be a speed delay when accessing external program memory.

3. REP and DO instructions are very useful for saving program memory locations and in general are very efficient instructions, however they do take extra cycles to set up the loop registers. In many cases in the ADPCM code instructions that could be performed with a REP or DO loop are instead repeated in the code multiple times. An example is the iterative NORM instruction used for the logarithmic and floating-point conversions. Instead of using a "REP #14" preceding the NORM instruction, 14 separate NORM instructions are used. For a single case the savings are very little but when adding up the savings throughout the algorithm this allows several cycles of execution time to be saved for each sample. In the case of FMULT which is executed eight times, this allows a total of 16 instruction cycles to be saved for each sample.
  
4. In many cases gains can be obtained by examining instruction encoding. For instance, immediate operands that are greater than 8 bits (or 12 bits in some cases) require an extra word of storage in program memory and also cause the instruction to take an extra cycle to fetch the operand. This is also the case for immediate data that is less than eight bits but not left-justified. The ADPCM algorithm needs several operands of this type, 32 to be exact. Instead of addressing them as immediate operands they are moved to a constant table in data memory and addressed with register r7. This does not cost extra memory since these constants require an extension word in program memory in any case. The advantage of this technique is that an extra



instruction cycle is saved by not having to fetch an instruction extension word. For the data in the table in Y memory this results in a savings of 32 cycles per sample for the entire algorithm. This technique is also used in the FMULT routine that uses its own constant table so even more savings per sample are added. In addition to the instruction fetch savings, this allows more parallelism to be exploited since the most efficient parallel addressing modes on the DSP56001 require the use of addressing registers. The disadvantages of this technique are that a dedicated register is required to address them and code is harder to follow and modify because of the added complexity. To help alleviate the complexity of the ADPCM code these constants are referred to as Kxx in the comments so that what is actually being read from the constant table is easier to identify.

5. Since the DSP56001 is a fractional-based architecture, many operations are more efficient when the data is left-justified. An example of this is the 8-bit immediate operand storage mentioned above. Another example is the NORM instruction that deals with the left-most bits in an accumulator. Data that is already left-justified as much as possible will require less shifting for normalization and therefore fewer NORM iterations. For this reason data is kept left-justified throughout the code as much as possible.
6. Parallel moves are taken advantage of whenever possible. This makes the assembly code more difficult to understand and modify since values may be fetched from memory long before they are actually used, in many cases while the previous routine is being executed. In many applications this can save considerable time especially when multiple loops of a code segment are executed.

Dual parallel moves on the DSP56001 usually require the use of an addressing register for accessing memory. Whenever possible pointers are used instead of immediate addresses.

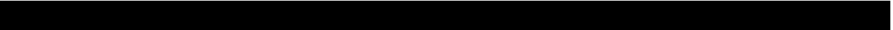
7. Knowledge of efficient test instructions can result in savings when decisions are necessary. For example, many cases in the CCITT specification refer to single bit variables. Whenever possible the single bit is stored in the sign bit of the 24-bit word whether it represents a sign or not. This allows instructions to test the sign of the word to determine if the variable is set or not.
8. Some of the most efficient instructions on the DSP56001 are the conditional transfer instructions. These are taken advantage of in several places in the ADPCM source code. These include the routines used for limiting such as the LIMC routine. In this routine a value is tested for both an upper and a lower limit. When this is required the following code segment can be used:

```
; Lower limit in x0
; Upper limit in xl
; Value to be limited in a

      cmp      x0,a      <parallel move>
      tlt      x0,a
      cmp      xl,a      <parallel move>
      tgt      xl,a
```

This code segment can be very efficient since each instruction executes in a single cycle and the CMP instructions can have parallel moves associated with them. When it is possible to use these transfer instructions the savings over using branching instructions can be great.

9. As with virtually all microprocessors, JMP instructions on the DSP56001 should be avoided whenever possible. These instructions take a minimum of two instruction cycles to execute due to



the instruction pipeline and do not allow parallel moves. Conditional jump instructions also take a minimum of two instruction cycles even if the jump is not taken. As mentioned above, conditional jumps can be avoided in many cases by using conditional transfer instructions instead.

10. Many different data formats are encountered in the ADPCM assembly code. Some of these have been discussed previously. Trying to adjust these different formats to the DSP56001's fractional data format is not practical. Instead, these formats are allowed to "float" freely within the 24-bit data word, or the 56-bit data word in the accumulators. The goal is to find the most efficient format that requires the least amount of data manipulation. An example is the many shifts that are specified in the CCITT standard. In many of these cases the shift is not actually performed. Instead, the value is truncated by using a mask constant and an AND instruction.
11. When shifting data cannot be avoided the lack of a single-cycle, multi-bit hardware shifter on the DSP56001 can be a problem. This is another example of where complete knowledge of the architecture of the part is a key factor in code optimization. The DSP56001 does not have a hardware shifter but it does have a single-cycle hardware multiplier. Since shift operations are actually just a multiplication by a power of two, the multiplier can be used as a shifter.

A basic description of this technique is presented in the Motorola DSP application report "Fractional and Integer Arithmetic Using the DSP56000..." [9]. This application report describes two techniques for doing multi-bit shifts on the DSP56000/1. The most straightforward approach uses a REP or DO in-

struction followed by a shift instruction. As noted previously, this process is iterative and requires overhead for the DO or REP. A faster way calls for multiplying the operand by a shift constant. If the amount of shift is always the same, the constant can be explicitly coded in the instruction sequence. If however, the shift amount is not always the same there is a convenient method of getting the appropriate shift constant. This method uses a lookup table for determining the shift constant. The table should be of the following form:

```

                org      x:
rshift         equ      *-1
                dc      $400000      ;>>1 or <<23
                dc      $200000      ;>>2 or <<22
                dc      $100000      ;>>3 or <<21
                * * *
                dc      $000002      ;>>22 or <<2
                dc      $000001      ;>>23 or <<1
lshift         equ      *
```

This table can be put in either X or Y memory. To perform an arbitrary right shift the value of rshift should be loaded into one of the address registers as a table base. The amount of the shift will then be loaded into the corresponding offset register so that the appropriate shift constant can be read from the table in memory. The technique is similar for an arbitrary left shift although the offset will be negative. For either a left or right shift the integer shift amount should be between 1 and 23. The following is an example of a right shift:

```

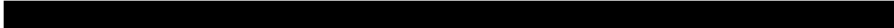
move    #rshift,r0      ;Set r0=table base
move    b0,n0           ;Load shift amount as offset
move    a1,x1           ;Set data up for shift
move    x:(r0+n0),x0    ;Lookup shift constant
mpy     x0,x1,a         ;Shift data right
                        ;Result is shifted into a0
```

In this example, the 24-bit number to be shifted is in a1 and the amount of the shift is in the LSBs of b0. The base of the right shift lookup table rshift is loaded into r0 and the amount of the shift is loaded into n0 as an offset into the table. The shift constant is found by using the Indexed by Offset addressing mode. The table base in r0 is not changed so the r0 does not have to be loaded again for another shift unless r0 is used elsewhere. To accomplish the shift the shift constant is put in x0 and the 24-bit number that is to be shifted is put in x1. The MPY instruction performs the shift. The result is found in a1 and a0. It is also sign extended into a2. This method is faster than the REP or DO method because the actual shift takes only one instruction cycle. Data movement is the only overhead and it can often be done in parallel with other operations. The only other extra time needed is the one extra instruction cycle required by the Indexed by Offset mode.

The following is an example of a left shift:

```
move    #lshift,r0      ;Set r0=table base
neg     b              a1,x1 ;Find negative shift amount,
                          ; set data up for shift
move    b0,n0          ;Load s.a. as negative offset
move    x:(r0+n0),x0   ;Lookup shift constant
mpy     x0,x1,a        ;Shift data left
                          ;Result is shifted into a1
```

This example is very similar to the right shift except that a negative offset is used. The shift amount is negated before it is loaded as an offset, since the Indexed by Offset mode can only use a positive offset. Again the result is found in a0 and a1 and is



sign extended into a2. The left shift requires the same amount of data movement as the right shift but also needs an extra ALU operation to negate the shift amount. In the example shown, the execution time is the same as the right shift.

The APDCM code uses this technique in several forms. The shift table is in Y memory and uses the labels RSHFT and LSHFT. In some cases the table is addressed like the above examples but with register r5 as the pointer. When the immediate short addressing mode can be used, the table is addressed using this mode instead of the address pointer. In addition, some shift constants are addressed as immediate operands in the instruction word. In all cases the actual shift is performed in the same manner as the above examples.

As mentioned, many of these optimization techniques improve execution speed at the expense of memory usage or code complexity. Since execution speed is not as high a priority in the non-standard version, several of these techniques are not used. The result is greater memory savings and lesser code complexity. Some techniques, such as shifting by multiplication, are still used in some cases however. As with almost any program there are always many trade-offs to be considered. If a particular application requires greater speed from the non-standard version, the optimization techniques can be added to the code following the examples shown in the standard version.



## 5.5 Performance Specifications

The memory usage for both implementations of the DSP56001 ADPCM algorithm is shown in Table 5-1. Both programs use only internal X and Y memory so no external data memory is required. External program memory is required for the standard version, however. As noted previously, this memory must be zero-wait state memory for a 27 MHz DSP56001 (45 ns access time) in order for the standard code to run in real-time. Table 5-1 shows a total of 611 words of program memory for the non-standard code indicating some external memory is required. However, most of the last 132 words of the code is taken up by the INIT routine and the load-time constant table storage. The remaining few words are part of the COMPRESS routine. Most applications will not require simultaneous use of both  $\mu$ -law and A-law compression. If either the  $\mu$ -law or A-law portion of the INIT routine is removed then the entire "run-time" portion of the non-standard code (excluding the INIT routine) will reside in internal memory. If a host processor is attached to the DSP56001 then no external memory will be required. Otherwise, slower memories can be used during the initialization process.

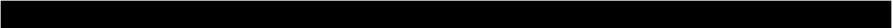
**Table 5-1** Memory Usage

	PROGRAM		DATA	
	Standard	Non-standard	Standard	Non-standard
Internal	447	479	118 (X)	70 (X)
External	783	132		
Total	1230	611	222	142

**Note:** All values are for 24 bit words.

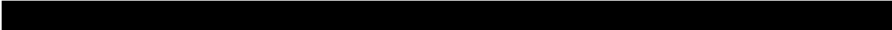
Table 5-2 shows the order of execution of the routines and the worst-case processing time in instruction cycles for each routine. A routine is defined as the code between commented sections even though, in the case of the standard version, some processing for that function may not be included in this code. These worst-case times are calculated based on worst-case branches and delays in each routine. Note that an instruction cycle (Icycle) is defined as two clock cycles on the DSP56001. For a 27 MHz DSP56001 an instruction cycle is 74.1 ns. Sampling at a rate of 8 kHz gives 125 microseconds to do both an encode and a decode. This translates into 1687 instruction cycles on 27 MHz DSP56001.

The calculations for the standard version in Table 5-2 indicate a total of 1707 instruction cycles for full-duplex operation (both the encoder and decoder). It has been found, however, that there is very little possibility of the maximum delay occurring in all of the routines in the encoder and decoder simultaneously.



No samples of the CCITT test sequences were observed to exceed the real-time limit and no indications of this occurrence were found in the real-time test set-up. If an extra margin is desired, the input clock can be increased to 27.5 MHz. Also the synchronization block in the decoder is not necessary for the ADPCM algorithm itself. This includes the routines EXPAND, SUBTA, LOG, SUBTB, and SYNC. This block is included for synchronization of multiple PCM/ADPCM/PCM conversions on a single channel. If only one PCM/ADPCM/PCM conversion is used the deletion of this block should not significantly affect the output speech quality. In this case the worst case execution time will be 1589 instruction cycles. Note that these routines are necessary to correctly pass the CCITT test sequences.

The non-standard version takes a total of 984 instruction cycles for both the encode and decode, indicating full-duplex operation is possible on a 20.5 MHz DSP56001. As noted, the non-standard implementation removed some of the optimizations used in the standard version to conserve program memory. If desired, these optimizations can be added to the non-standard version to improve performance. At least 100 instruction cycles can be saved from both the encoder and the decoder. This would allow 2 full-duplex channels on a 27 MHz DSP56001.



**Table 5-2 Code Execution Times (page 1 of 2)**

ENCODER			DECODER		
	Standard	Non-standard		Standard	Non-standard
FMULT (x8)	341	-	FMULT (x8)	341	-
ACCUM	13	14	ACCUM	13	14
LIMA	4	4	LIMA	4	4
MIX	14	9	MIX	14	9
EXPAND	10	14	RECONST	13	10
SUBTA	3	2	ADDA	3	1
LOG	22	28	ANTILOG	25	28
SUBTB	3	1	TRANS	34	32
QUAN	36	41	ADDB	8	6
RECONST	7	10	ADDC	19	11
ADDA	3	1	XOR	-	-
ANTILOG	25	28	UPB (x8)	76	72
TRANS	34	32	UPA2	27	25
ADDB	8	6	LIMC	6	5
ADDC	19	11	UPAI	12	14
XOR	-	-	LIMD	8	8
UPB (x8)	76	72	FLOATA	22	7
UPA2	27	25	FLOATB	27	1
LIMC	6	5	TONE	5	3
UPAI	12	14	TRIGB	13	15

**Table 5-2 Code Execution Times (page 2 of 2)**

ENCODER			DECODER		
	Standard	Non-standard		Standard	Non-standard
LIMD	8	8	FUNCTF	14	5
FLOATA	22	7	FILTA	6	4
FLOATB	27	1	FILTB	5	3
TONE	5	3	SUBTC	11	11
TRIGB	13	15	FILTC	5	3
FUNCTF	14	5	TRIGA	3	3
FILTA	6	4	FUNCTW	7	5
FILTB	5	3	FILTD	5	4
SUBTC	11	11	LIMB	4	7
FILTC	5	3	FILTE	9	4
TRIGA	3	3	COMPRESS	33	39
FUNCTW	7	5	EXPAND	10	-
FILTD	5	4	SUBTA	3	-
LIMB	4	7	LOG	22	-
FILTE	8	4	SUBTB	3	-
misc.	4	116	SYNC	80	-
	-	-	misc.	7	117
TOTAL	810	513	TOTAL	897	471

**Note:** All numbers are in lcycles and are for worst case delays.

## APPENDIX

# Terminology

***“... the suffix ‘\_T’ refers to those labels associated with the encoder (transmit) and the suffix ‘\_R’ refers to those associated with the decoder (receive).”***

---

**T**he DSP56001 assembly code and this application note use several different symbols describing the ADPCM implementation. Most of these symbols are derived from the G.721 specification [1]. The following symbols correspond to the variable types defined in the G.721 specification:

SM = signed magnitude value

TC = two's complement value

FL = floating point value

A number preceding one of these symbols shows the number of total bits in a particular variable (e.g. 14TC represents a 14-bit two's complement number). See Table 3 of Recommendation G.721 (Reference 1) for the full binary representation of each variable, including the location of the radix point. The contents of registers or memory locations at certain points in the code are also detailed bit for bit.

This application note uses the following terminology:

- . = location of implied radix point
- i = integer bit
- f = fraction bit
- s = sign bit
- m = mantissa bit
- e = exponent bit
- 1 = bit is always 1
- 0 = bit is always 0
- X = bit value is unknown but is not significant

An **exception** to the above list is the PCM word where:

- p = sign bit
- s = segment bit
- q = quantization level bit

Note that when labels are used to refer to variables or program locations in the assembly code the suffix “\_T” refers to those labels associated with the encoder (transmit) and the suffix “\_R” refers to those associated with the decoder (receive).

Example:

```
; y = 0iiii i.fff | ffff ff00 | 0000 0000 (13sm)
Y_T              DS          1           ;Quantizer scale factor
```



The above example shows the memory allocation for a variable — the scale factor  $y(k)$ . It is defined as a 13-bit signed magnitude number with four integer bits and nine fractional bits. The value stored in memory is always stored in the 24-bit format with the implied radix point between bits 18 and 19.

Example:

```
; A1 = 01mm mmm0 | 0000 0000 | 0000 0000 (A2=A0=0)
; B1 = 0000 0000 | 0000 0000 | 0000 eeee (B2=B0=0)
```

At the point where these comments appear in the code; the register a1 always contains a 1 in bit 22, five other mantissa bits, and all other bits set to 0. Register b1 always contains four exponent bits in bits 0 through 3 with all other bits set to 0. Registers a0, a2, b0, and b2 are always set to 0. ■



## REFERENCES

1. CCITT Recommendation G.721, "32 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)", Study Group XVIII - Report R 26(C), August 1986.
2. D. O'Shaughnessy, *Speech Communication — Human and Machine*, Addison-Wesley, 1987.
3. A.V. Oppenheim and R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
4. N.S. Jayant and P. Noll, *Digital Coding of Waveforms*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
5. CCITT Recommendation G.711, "Pulse Code Modulation (PCM) of Frequencies", *CCITT Red Book*, October, 1984.
6. "Logarithmic/Linear Conversion Routines for DSP56000/1", Motorola, Inc., DSP Operation Technical Brief.
7. L. R. Rabiner and R.W. Schaffer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
8. N. Benvenuto et al., "The 32-kbit/s ADPCM Coding Standard", *AT&T Technical Journal*, Vol. 65, No. 5, September/October 1986, pp. 12-22.
9. "Fractional and Integer Arithmetic Using the DSP56000 Family of General Purpose Digital Signal Processors", Motorola, Inc., DSP Division Technical Report APR3/D.
10. CCITT Recommendation G.712, "Performance Characteristics of PCM Channels Between 4-Wire Interfaces at Voice Frequencies", *CCITT Red Book*, October, 1984.

