INTRODUCTION

In the past, adding speech recording and playback capability to a product meant using a digital signal processor or a specialized audio chip. Now, using a simplified Adaptive Differential Pulse Code Modulation (ADPCM) algorithm, these audio capabilities can be added to any PICmicro device. This application note will cover the ADPCM compression and decompression algorithms, performance comparison of all PICmicro devices, and an application using a PIC16C72 microcontroller.

DEFINITION OF TERMS

step size - value of the step used for quantization of analog signals and inverse quantization of a number of steps.
quantization - the digital form of an analog input signal is represented by a finite number of steps.
adaptive quantization - the step size of a quantizer is dramatically changed with time in order to adapt to a changing input signal.
inverse quantizer - a finite number of steps is converted into a digital representation of an analog signal.

THEORY OF OPERATION

The ADPCM algorithm takes advantage of the high correlation between consecutive speech samples, which enables future sample values to be predicted. Instead of encoding the speech sample, ADPCM encodes the difference between a predicted sample and the speech sample. This method provides more efficient compression with a reduction in the number of bits per sample, yet preserves the overall quality of the speech signal. The implementation of the ADPCM algorithm provided in this application note is based on the Interactive Multimedia Association’s (IMA) Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems revision 3.00. The ITU (formerly CCITT) G.721 ADPCM algorithm is well known and has been implemented on many digital signal processors such as the TMS320 family from Texas Instruments and the ADSP-2100 family from Analog Devices. ITU G.721 uses floating point arithmetic and logarithmic functions which are not easily implemented in the 8-bit microcontroller world. The IMA Reference algorithm significantly reduces the mathematical complexity of ITU G.721 by simplifying many of the operations and using table lookups where appropriate.

COMPRESSION

The input, \( s_i \), to the encoder routine must be 16-bit two’s complement speech data. The range of allowable values for \( s_i \) is 32767 to -32768. Figure 1 shows a block diagram for ADPCM compression and Appendix A has a listing of the function `ADPCMEncoder()`. The predicted sample, \( s_p \), and the quantizer step size index are saved in a structure for the next iteration of the encoder. Initially, the quantizer step size index and the predicted sample \( (s_p) \) are set to zero. The encoder function takes a 16-bit two’s complement speech sample and returns an 8-bit number containing the 4-bit sign-magnitude ADPCM code.

FIGURE 1: ADPCM ENCODER BLOCK DIAGRAM

The predicted sample, \( s_p \), is subtracted from the linear input sample, \( s_i \), to produce a difference, \( d \). Adaptive quantization is performed on the difference, resulting in the 4-bit ADPCM value, \( t \). The encoder and decoder both update their internal variables based on this ADPCM value. A full decoder is actually embedded within the encoder. This ensures that the encoder and decoder are synchronized without the need to send any additional data. The embedded decoder is shown within the dotted lines of Figure 1. The embedded decoder uses the ADPCM value to update the inverse quantizer, which produces a dequantized version, \( d_q \), of the difference, \( d \). The implementation of the ADPCM algorithm presented in this application note uses a fixed predictor instead of an adaptive predictor which reduces the amount of data memory and instruction cycles required.
The adaptive predictor of ITU G.721 adjusts according to the value of each input sample using a weighted average of the last six dequantized difference values and the last two predicted values. So at this point, the dequantized difference, $d_{pq}$, is added to the predicted sample, $s_p$, to produce a new predicted sample, $s_r$. Finally the new predicted sample, $s_r$, is saved into $s_p$.

The following (Table 1) is a step-by-step description of the ADPCMEncoder() function from Appendix A.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>ADPCMEncoder takes a 16-bit signed number (speech sample, 32767 to -32768) and returns an 8-bit number containing the 4-bit ADPCM code (0-15)</td>
</tr>
<tr>
<td></td>
<td>char ADPCMEncoder(long signed sample)</td>
</tr>
<tr>
<td>2.</td>
<td>Restore the previous values of predicted sample ($s_p$) and the quantizer step size index</td>
</tr>
<tr>
<td></td>
<td>predsample = state.prevsample;</td>
</tr>
<tr>
<td></td>
<td>index = state.previndex;</td>
</tr>
<tr>
<td>3.</td>
<td>Find the quantizer step size ($q$) from a table lookup using the quantizer step size index</td>
</tr>
<tr>
<td></td>
<td>step = StepSizeTable[index];</td>
</tr>
<tr>
<td>4.</td>
<td>Compute the difference ($d$) between the actual sample ($s_i$) and the predicted sample ($s_p$)</td>
</tr>
<tr>
<td></td>
<td>diff = sample - predsample;</td>
</tr>
<tr>
<td>5.</td>
<td>Set the sign bit of the ADPCM code ($t$) if necessary and find the absolute value of difference ($d$)</td>
</tr>
<tr>
<td></td>
<td>if(diff &gt;= 0)</td>
</tr>
<tr>
<td></td>
<td>code = 0;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>code = 8;</td>
</tr>
<tr>
<td></td>
<td>diff = - diff;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>6.</td>
<td>Save quantizer step size ($q$) in a temporary variable</td>
</tr>
<tr>
<td></td>
<td>tempstep = step;</td>
</tr>
<tr>
<td>7.</td>
<td>Quantize the difference ($d$) into the ADPCM code ($t$) using the quantizer step size ($q$)</td>
</tr>
<tr>
<td></td>
<td>if(diff &gt;= tempstep)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td></td>
<td>diff -= tempstep;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>tempstep &gt;&gt;= 1;</td>
</tr>
<tr>
<td></td>
<td>if(diff &gt;= tempstep)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td></td>
<td>diff -= tempstep;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>tempstep &gt;&gt;= 1;</td>
</tr>
<tr>
<td></td>
<td>if(diff &gt;= tempstep)</td>
</tr>
<tr>
<td></td>
<td>code</td>
</tr>
<tr>
<td>8.</td>
<td>Inverse quantize the ADPCM code ($t$) into a predicted difference ($d_{pq}$) using the quantizer step size ($q$)</td>
</tr>
<tr>
<td></td>
<td>diffq = step &gt;&gt; 3;</td>
</tr>
<tr>
<td></td>
<td>if(code &amp; 4)</td>
</tr>
<tr>
<td></td>
<td>diffq += step;</td>
</tr>
<tr>
<td></td>
<td>if(code &amp; 2)</td>
</tr>
<tr>
<td></td>
<td>diffq += step &gt;&gt; 1;</td>
</tr>
<tr>
<td></td>
<td>if(code &amp; 1)</td>
</tr>
<tr>
<td></td>
<td>diffq += step &gt;&gt; 2;</td>
</tr>
</tbody>
</table>
9. Fixed predictor computes new predicted sample \( (s_r) \) by adding the old predicted sample \( (s_p) \) to the predicted difference \( (d_q) \):

\[
\text{if}(\text{code} \& 8) \\
\quad \text{predsample} -= \text{diffq}; \\
\text{else} \\
\quad \text{predsample} += \text{diffq};
\]

10. Check for overflow of the new predicted sample \( (s_r) \). \( s_r \) is a signed 16-bit sample, must be in the range of 32767 to -32768:

\[
\text{if}(\text{predsample} > 32767) \\
\quad \text{predsample} = 32767; \\
\text{else if}(\text{predsample} < -32768) \\
\quad \text{predsample} = -32768;
\]

\[
\text{index} += \text{IndexTable}[\text{code}];
\]

11. Find the new quantizer step size index \( (q) \) by adding the previous index and a table lookup using the ADPCM code \( (t) \):

\[
\text{if}(\text{index} < 0) \\
\quad \text{index} = 0; \\
\text{if}(\text{index} > 88) \\
\quad \text{index} = 88;
\]

12. Check for overflow of the new quantizer step size index:

\[
\text{if}(\text{index} < 0) \\
\quad \text{index} = 0; \\
\text{if}(\text{index} > 88) \\
\quad \text{index} = 88;
\]

13. Save the new predicted sample \( (s_r) \) and quantizer step size index for next iteration:

\[
\text{state.prevsample} = \text{predsample}; \\
\text{state.previndex} = \text{index};
\]

14. Return the ADPCM code \( (t) \):

\[
\text{return} \ (\text{code} \& \ 0x0f);
\]

This function requires five 16-bit variables and two 8-bit variables. Some optimizations can be made to the code in Appendix A such as combining steps 7 and 8. Appendix C gives the output listing of the MPC compiler for a PIC16CXX device using the optimized encoder algorithm and the decoder algorithm from the next section.

### DECOMPRESSION

The input into the decoder, \( t \), must be an 8-bit number containing the 4-bit ADPCM data in sign-magnitude format. The range of allowable values for \( t \) is 0 to 15, where 7 = 0x07 and -7 = 0x0F. Figure 2 shows a block diagram for ADPCM decompression and Appendix B has a listing of the function `ADPCMDecoder()`. The predicted sample, \( s_p \), and the quantizer step size index are saved in a structure for the next iteration of the decoder. Initially, the quantizer step size index and the predicted sample \( (s_p) \) are set to zero. This function takes a 4-bit sign-magnitude ADPCM code and returns the 16-bit two's complement speech sample.

This decoder is the same as the one used in the encoder routine. It uses the ADPCM value to update the inverse quantizer, which produces a difference, \( d_q \). The difference, \( d_q \), is added to the predicted sample, \( s_p \), to produce the output sample, \( s_r \). The output sample, \( s_r \), is then saved into the predicted sample, \( s_p \), for the next iteration of the decoder.

### FIGURE 2: ADPCM DECODER BLOCK DIAGRAM

![ADPCM Decoder Block Diagram](image)
The following (Table 2) is a step-by-step description of the `ADPCMDecoder()` function from Appendix B.

**TABLE 2: ADPCMDecoder() STEP-BY-STEP FUNCTIONS**

1. ADPCMDecoder takes an 8-bit number containing the 4-bit ADPCM code (0-15) and returns a 16-bit signed number (speech sample, 32767 to -32768)

   ```c
   signed long ADPCMDecoder( char code )
   ```

2. Restore the previous values of predicted sample \( (s_p) \) and quantizer step size index

   ```c
   predsample = state.prevsample;
   index = state.previndex;
   ```

3. Find the quantizer step size \( (q) \) from a table lookup using the quantizer step size index

   ```c
   step = StepSizeTable[index];
   ```

4. Inverse quantize the ADPCM code \( (t) \) into a predicted difference \( (d_q) \) using the quantizer step size \( (q) \)

   ```c
   if(code & 4) diffq += step;
   if(code & 2) diffq += step >> 1;
   if(code & 1) diffq += step >> 2;
   ```

5. Fixed predictor computes new predicted sample \( (s_r) \) by adding the old predicted sample \( (s_p) \) to the predicted difference \( (d_q) \)

   ```c
   if(code & 8) predsample -= diffq;
   else predsample += diffq;
   ```

6. Check for overflow of the new predicted sample \( (s_r) \), \( s_r \) which is a signed 16-bit sample, must be in the range of 32767 to -32768

   ```c
   if(predsample > 32767) predsample = 32767;
   else if(predsample < -32768) predsample = -32768;
   ```

7. Find the new quantizer step size \( (q) \) by adding the previous index and a table lookup using the ADPCM code \( (t) \)

   ```c
   index += IndexTable[code];
   ```

8. Check for overflow of the new quantizer step size index

   ```c
   if(index < 0) index = 0;
   if(index > 88) index = 88;
   ```

9. Save the new predicted sample \( (s_r) \) and quantizer step size index for next iteration

   ```c
   state.prevsample = predsample;
   state.previndex = index;
   ```

10. Return the new sample \( (s_r) \)

    ```c
    return ( predsample );
    ```

This function requires three 16-bit variables and one 8-bit variable. Appendix C gives the listing output of the MP-C compiler of the decoder algorithm and the optimized encoder algorithm.
IMA ADPCM REFERENCE ALGORITHM

The IMA, specifically the Digital Audio Technical Working Group, is a trade association with representatives from companies such as Compaq®, Apple Computer®, Crystal Semiconductor®, DEC®, Hewlett-Packard®, Intel®, Microsoft®, Sony®, and Texas Instruments® to name a few. This group is working towards a standard that defines the exchange of high quality audio data between computing platforms. The algorithm from Intel/DVI® (Digital Video Interactive) has been selected as the standard due to its audio dynamic range and low data rate. The recommended digital audio exchange formats are given in Table 3.

The algorithms that are implemented in this application note were derived from the IMA ADPCM Reference algorithm. The data format is 8.0 kHz, mono, 4-bit ADPCM. Essentially, the compression and decompression use an adaptive quantization with fixed prediction. The adaptive quantization are based on a table lookup first developed by Intel/DVI for the IMA. Appendix D gives the information about IMA and the supporting documentation for IMA ADPCM.

### TABLE 3: DIGITAL AUDIO EXCHANGE FORMATS

<table>
<thead>
<tr>
<th>Sampling Rate</th>
<th>Mono/Stereo</th>
<th>Data Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0 kHz</td>
<td>mono</td>
<td>8-bit μ-Law PCM</td>
<td>CCITT G.711 Standard</td>
</tr>
<tr>
<td></td>
<td>mono</td>
<td>8-bit A-Law PCM</td>
<td>CCITT G.711 Standard</td>
</tr>
<tr>
<td></td>
<td>mono</td>
<td>4-bit ADPCM</td>
<td>DVI Algorithm</td>
</tr>
<tr>
<td>11.025 kHz</td>
<td>mono/stereo</td>
<td>8-bit Linear PCM</td>
<td>Macintosh® &amp; MP-C Standard</td>
</tr>
<tr>
<td></td>
<td>mono/stereo</td>
<td>4-bit ADPCM</td>
<td>DVI Algorithm</td>
</tr>
<tr>
<td>22.05 kHz</td>
<td>mono/stereo</td>
<td>8-bit Linear PCM</td>
<td>Macintosh &amp; MPC Standard</td>
</tr>
<tr>
<td></td>
<td>mono/stereo</td>
<td>4-bit ADPCM</td>
<td>DVI Algorithm</td>
</tr>
<tr>
<td>44.10 kHz</td>
<td>mono/stereo</td>
<td>16-bit Linear PCM</td>
<td>CD-DA Standard</td>
</tr>
<tr>
<td></td>
<td>mono/stereo</td>
<td>4-bit ADPCM</td>
<td>DVI Algorithm</td>
</tr>
</tbody>
</table>

### PERFORMANCE

Table 4 shows the performance comparison of the ADPCM compression and decompression routines for the PIC16C5X, PIC16CXXX, and PIC17CXXX family of devices assuming 80 additional instruction cycles for overhead. Any device without a PWM module will have to increase the operating frequency to generate a software PWM. Table 4 also provides the minimum external operating frequency required to run the routines. The C code from Appendix C was used. The input/output data rate of the speech samples is 8.0 kHz. The minimum operating frequency is calculated as follows:

\[
\text{Minimum External Operating Frequency} = \frac{8000 \times \# \text{ of Total Instruction Cycles} \times 4}{16000}
\]

For example, the ADPCM encoding using the PIC16CXXX family would require an 8000 x (225 + 80) x 4 = 9.760 MHz external crystal.

### TABLE 4: PERFORMANCE COMPARISON TABLE

<table>
<thead>
<tr>
<th>Device</th>
<th>Encode/Decode</th>
<th># of Instruction Cycles</th>
<th>Minimum External Operating Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C5X</td>
<td>Encode</td>
<td>273</td>
<td>11.296 MHz</td>
</tr>
<tr>
<td></td>
<td>Decode</td>
<td>208</td>
<td>9.216 MHz</td>
</tr>
<tr>
<td>PIC16CXXX</td>
<td>Encode</td>
<td>225</td>
<td>9.760 MHz</td>
</tr>
<tr>
<td></td>
<td>Decode</td>
<td>168</td>
<td>7.936 MHz</td>
</tr>
<tr>
<td>PIC17CXXX</td>
<td>Encode</td>
<td>199</td>
<td>8.928 MHz</td>
</tr>
<tr>
<td></td>
<td>Decode</td>
<td>161</td>
<td>7.712 MHz</td>
</tr>
</tbody>
</table>

Compaq is a registered trademark of Compaq Computer.
Apple Computer and Macintosh are a registered trademarks of Apple Computer, Inc.
Crystal Semiconductor is a registered trademark of Crystal Semiconductor.
DEC is a registered trademark of Digital Equipment Corporation.
Hewlett-Packard is a registered trademark of Hewlett-Packard Company.
Intel and DVI are registered trademarks of Intel Corporation.
Microsoft is a registered trademark of Microsoft Corporation.
Sony is a registered trademark of Sony Corporation.
Table 5 illustrates the amount of program and data memory that the ADPCM algorithms consume for the various PICmicro devices. The program memory numbers may change slightly depending on the specific device being used. The table memory column shows how much program memory is used to store the two lookup tables used by the ADPCM algorithm.

**TABLE 5: DEVICE MEMORY CONSUMED**

<table>
<thead>
<tr>
<th>Device</th>
<th>Program Memory (words)</th>
<th>Data Memory (bytes)</th>
<th>Table Lookup (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C5X</td>
<td>Encode 273</td>
<td>13</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>Decode 205</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>PIC16CXXX</td>
<td>Encode 220</td>
<td>13</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>Decode 162</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>PIC17CXXX</td>
<td>Encode 203</td>
<td>13</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>Decode 164</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**APPLICATION**

The hardware for this application note implements only the decompression algorithm, but the compression algorithm is also included in the firmware. A block diagram is shown in Figure 3 and the schematic is provided in Appendix E. The complete source code listing, including a block diagram, is given in Appendix F.

The board uses a PIC16C72, rated for 20 MHz operation, to control the speech decoding and output. The A/D (Analog-to-Digital) converter of the PIC16C72 is not used in this design. Two 27C512A EPROMs from Microchip Technology are used to store up to 32.768 seconds of ADPCM data. Each EPROM holds 65536 bytes. Each byte in the EPROM contains two ADPCM values. One second of speech requires 8000 ADPCM codes (8 kHz sample rate). Therefore, each EPROM holds (65536 x 2) / 8000 = 16.384 seconds of speech. A 16-bit up counter is used to clock data out of the EPROMs. This method uses only two I/O lines to control the counter and eleven I/O lines to read the EPROM data.

Speech regeneration is accomplished by using the Capture/Compare/PWM (CCP) module of the PIC16C72. The PWM module is configured for a period of 32 kHz. This allows each sample of the speech signal to be output for four PWM periods. The period is calculated by using the following equation from Section 10.3 of the PIC16C7X Data Sheet (DS30390). From the following calculation, PR2 is set to a value of 155.

\[
\text{PWM period} = [PR2 + 1] * 4 * \text{Tosc} * (\text{TMR2 prescale value})
\]

\[
\frac{1}{32 \text{ kHz}} = [PR2 + 1] * 4 * (1 / 20 \text{ MHz}) * 1
\]

\[
31.25 \mu s = [PR2 + 1] * 4 * 50 \text{ ns} * 1
\]

156.25 = PR2 + 1

155.25 = PR2

The CCP module has the capability of up to 10-bit resolution for the PWM duty cycle. The maximum resolution of the duty cycle that can be achieved for a 32 kHz period can be calculated using the following equation from Section 10.3 of the PIC16C7X Data Sheet.

\[
\text{PWM duty cycle} = \frac{\text{DC}<10:0> * \text{Tosc} * (\text{TMR2 prescale value})}{\text{PWM period}}
\]

where DC<10:0> = 2^x where x = bits of resolution

\[
\frac{1}{32 \text{ kHz}} = 2^x * (1 / 20 \text{ MHz}) * 1
\]

\[
31.25 \mu s = 2^x * 50 \text{ ns} * 1
\]

625 = 2^x

log(625) = log(2^x)

log(625) = x * log(2)

9.3 = x

A PWM duty cycle with up to 9-bits of resolution may be used with a period of 32 kHz. The upper 9-bits of each speech sample are used to set the PWM duty cycle (CCPR1L = sample<15:9>, CCP1CON<5:4> = sample<8:7>). This PWM speech signal is passed through a 4th-order Butterworth low pass filter with a corner frequency of 4 kHz. The low pass filter converts the PWM into an analog voltage level. Finally, the analog voltage level is amplified by a National Semiconductor LM386N-3 before entering the speaker.

Every 125 μs (8 kHz) one ADPCM code must be converted into a speech sample for output to the PWM. This frame of 125 μs relates to 625 instruction cycles for an external crystal frequency of 20 MHz. The ADPCM decode routine, EPROM reads and writes to the PWM must be accomplished within 625 instruction cycles.

**FIGURE 3: APPLICATION HARDWARE BLOCK DIAGRAM**

```
PIC16C72
```

```
16-bit Counter
```

```
27C512A
```

```
Speaker
```

```
Power Supply
```

```
Butterworth LPF
```

```
Amplifier
```

```
2
```

```
3
```

```
8
```

```
27C512A
```

```
2
```

```
3
```

```
8
```

```
16
```

```
27C512A
```

```
2
```

```
3
```

```
8
```

```
16
```

```
27C512A
```

```
2
```

```
3
```

```
8
```

```
16
```
To implement audio recording, the following changes to the application hardware would have to be made:

- Replace the EPROMs with RAM and
- Add a microphone with input filtering and
- Use the on-board A/D converter of the PIC16C7X devices for 8-bit recording or
- Use an external 12- to 16-bit A/D converter for better quality recording

The following is the sequence of events to record and store speech data. A timer would be used to set the sample rate. When the timer overflowed an A/D conversion would be started.

1. Start an A/D conversion when the timer overflows.
2. Read the sample and call the routine ADPCMEncoder() .
3. Store the result in the upper 4-bits of a temporary variable.
4. Start an A/D conversion when the timer overflows.
5. Read the sample and call the routine ADPCMEncoder() .
6. Store the result in the lower 4-bits of the temporary variable.
7. Write the temporary variable to the SRAM.
8. GOTO 1.

Typical conversion times for the PIC16C7XX devices are 40 µs. Assuming that the external crystal frequency is 20 MHz and the sample rate is 8 kHz, the encoder routine takes approximately 45 µs (225 instruction cycles x 200 ns) to complete. This leaves approximately 40 µs or 200 instruction cycles to read the A/D converter, write to the SRAM and complete any other tasks. An external A/D converter that is faster could be used to increase the amount of time for processing other tasks.

COMPUTER PROGRAM

The IMA Reference ADPCM algorithm was written for a PC using Borland C++ version 3.1. This program uses the same routines that are used in Appendix C. A raw sound file is recorded on the PC using a 16-bit sound card. Speech is recorded at 8.0 kHz in 16-bit mono mode. This raw speech file is processed by the encoder part of the program which creates a file with the ADPCM values. This file can now be burned into EPROMs for use with the application hardware. The decoder part of the program can take this file and create a new 16-bit raw speech file. The 16-bit sound card on the PC can play this file to ensure that the ADPCM compression/decompression routines are working properly. The PC program is given in Appendix F. It is composed of three files: pcspeech.c, pcadpcm.c, and pcadpcm.h.

CONCLUSION

The final results of the application hardware using the PIC16C72 are:

- Compression and Decompression
  - 906 words of Program Memory used
  - 44% of total Program Memory
  - 24 bytes of Data Memory used
  - 19% of total Data Memory
- Decompression only
  - 686 words of Program Memory used
  - 33% of total Program Memory
  - 22 bytes of Data Memory used
  - 17% of total Data Memory
- ~250 Instruction cycles to decode one ADPCM code
  - 40% of frame (250/625)
- Hardware used
  - CCP1 configured for PWM (9-bit duty cycle, 32 kHz period speech output)
  - Timer2 to set the 8.0 kHz output sample rate
  - 13 I/O lines to read data from the EPROMs

References:

5. 32-kbits/s ADPCM with the TMS32010, Digital Signal Processing Applications with the TMS320 Family, SPRA012, Jay Reimer, Mike McMahan, and Masud Arjmand, Texas Instruments, 1986.
APPENDIX A: ADPCMEncoder() FUNCTION

/* Table of index changes */
const int IndexTable[16] = {
    0xff, 0xff, 0xff, 0xff, 0x02, 0x04, 0x06, 0x08,
    0xff, 0xff, 0xff, 0xff, 0x02, 0x04, 0x06, 0x08
};

/* Quantizer step size lookup table */
const long StepSizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};
signed long diff;               /* Difference between sample and predicted sample */
long step;                      /* Quantizer step size */
signed long predsample;         /* Output of ADPCM predictor */
signed long diffq;              /* Dequantized predicted difference */
int index;                      /* Index into step size table */

char ADPCMEncoder( signed long sample )
{
    int code;               /* ADPCM output value */
    int tempstep;           /* Temporary step size */

    /* Restore previous values of predicted sample and quantizer step size index */
    predsample = state.prevsample;
    index = state.previndex;
    step = StepSizeTable[index];

    /* Compute the difference between the actual sample (sample) and the predicted sample (predsample) */
    if(diff >= 0)
    {
        code = 0;
        diff = -diff;
    }
    else
    {
        code = 8;
        diff = -diff;
    }

    /* Quantize the difference into the 4-bit ADPCM code using the quantizer step size */
    tempstep = step;
    if( diff >= tempstep )
    {
        code |= 4;
    }
    return code;
}
diff -= tempstep;
}
tempstep >>= 1;
if( diff >= tempstep )
{
    code |= 2;
    diff -= tempstep;
}
tempstep >>= 1;
if( diff >= tempstep )
    code |= 1;

/* Inverse quantize the ADPCM code into a predicted difference
   using the quantizer step size */
diffq = step >> 3;
if( code & 4 )
    diffq += step;
if( code & 2 )
    diffq += step >> 1;
if( code & 1 )
    diffq += step >> 2;

/* Fixed predictor computes new predicted sample by adding the
   old predicted sample to predicted difference */
if( code & 8 )
    predsample -= diffq;
else
    predsample += diffq;

/* Check for overflow of the new predicted sample */
if( predsample > 32767 )
    predsample = 32767;
else if( predsample < -32768 )
    predsample = -32768;

/* Find new quantizer steps size index by adding the old index
   to a table lookup using the ADPCM code */
index += IndexTable[code];

/* Check for overflow of the new quantizer step size index */
if( index < 0 )
    index = 0;
if( index > 88 )
    index = 88;

/* Save the predicted sample and quantizer step size index for
   next iteration */
state.prevsample = predsample;
state.previndex = index;

/* Return the new ADPCM code */
return ( code & 0x0f );
APPENDIX B:  ADPCMDecoder() FUNCTION

/* Table of index changes */
const int IndexTable[16] = {
  0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8,
  0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8
};

/* Quantizer step size lookup table */
const long StepSizeTable[89] = {
  7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
  19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
  50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
  130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
  337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
  876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
  2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
  5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
  15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

long step;                      /* Quantizer step size */
signed long predsample;         /* Output of ADPCM predictor */
signed long diffq;              /* Dequantized predicted difference */
int index;                      /* Index into step size table */

**************************************************************************
*   ADPCMDecoder - ADPCM decoder routine                              *
**************************************************************************
*   Input Variables:                                                     *
*       char code - 8-bit number containing the 4-bit ADPCM code          *
*   Return Variable:                                                     *
*       signed long - 16-bit signed speech sample                        *
**************************************************************************
signed long ADPCMDecoder(char code )
{
    /* Restore previous values of predicted sample and quantizer step  *
      size index                                                        *
    */
    predsample = state.prevsample;
    index = state.previndex;

    /* Find quantizer step size from lookup table using index           *
    */
    step = StepSizeTable[index];

    /* Inverse quantize the ADPCM code into a difference using the       *
      quantizer step size                                               *
    */
    diffq = step >> 3;
    if( code & 4 )
        diffq -= step;
    if( code & 2 )
        diffq -= step >> 1;
    if( code & 1 )
        diffq -= step >> 2;

    /* Add the difference to the predicted sample                       *
    */
    if( code & 8 )
        predsample -= diffq;
    else
        predsample += diffq;

    /* Check for overflow of the new predicted sample                   *
    */

}`
if( predsample > 32767 )
    predsample = 32767;
else if( predsample < -32768 )
    predsample = -32768;

/* Find new quantizer step size by adding the old index and a
table lookup using the ADPCM code */
index += IndexTable[code];

/* Check for overflow of the new quantizer step size index */
if( index < 0 )
    index = 0;
if( index > 88 )
    index = 88;

/* Save predicted sample and quantizer step size index for next
iteration */
state.prevsample = predsample;
state.previndex = index;

/* Return the new speech sample */
return( predsample );
}
APPENDIX C: OPTIMIZED ADPCMEncoder(), ADPCMDecoder(), AND TABLE LISTING

/*****************************************************************************/
* Filename:   ADPCM.C                                                     *
***************************************************************************/
*    Author:     Rodger Richey                                               *
*    Company:    Microchip Technology Incorporated                           *
*    Revision:   0                                                           *
*    Date:       1-9-96                                                      *
*    Compiled using Bytecraft Ltd. MPC version BC.193                        *
***************************************************************************/
*    This file contains the ADPCM encode and decode routines. This          *
*    routines were obtained from the Interactive Multimedia Association's    *
*    Reference ADPCM algorithm. This algorithm was first implemented by     *
*    Intel/DVI.                                                              *
***************************************************************************/
/* Table of index changes */
const int IndexTable[16] = {0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8,
                             0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8};

const long StepSizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 21, 23, 25, 28, 31, 34, 37,
    41, 45, 50, 55, 60, 66, 73, 80, 88, 97, 107, 118, 130, 143, 157,
    173, 190, 230, 253, 279, 307, 337, 371, 408, 449, 494, 544,
    598, 658, 724, 796, 876, 963, 1060, 1166, 1282, 1411, 1552,
    1707, 1878, 2066, 2272, 2499, 2749, 3024, 3327, 3660, 4026,
    4428, 4871, 5358, 5894, 6484, 7132, 7845, 8630, 9493, 10442,
    11487, 12635, 13899, 15289, 16818, 18500, 20350, 22385,
    24623, 27086, 29794, 32767};

/* Quantizer step size lookup table */
0016 0782 ADDWF PCL
0017 3407 RETLW 07h
0018 3400 RETLW 00h
0019 3408 RETLW 08h
001A 3400 RETLW 00h
001B 3409 RETLW 09h
001C 3400 RETLW 00h
001D 340A RETLW 0Ah
001E 3400 RETLW 00h
001F 340B RETLW 0Bh
0020 3400 RETLW 00h
0021 340C RETLW 0Ch
0022 3400 RETLW 00h
0023 340D RETLW 0Dh
0024 3400 RETLW 00h
0025 340E RETLW 0 Eh
0026 3400 RETLW 00h
0027 3410 RETLW 10h
0028 3400 RETLW 00h
0029 3411 RETLW 11h
002A 3400 RETLW 00h
002B 3413    RETLW  13h
002C 3400    RETLW  00h
002D 3415    RETLW  15h
002E 3400    RETLW  00h
002F 3417    RETLW  17h
0030 3400    RETLW  00h
0031 3419    RETLW  19h
0032 3400    RETLW  00h
0033 341C    RETLW  1Ch
0034 3400    RETLW  00h
0035 341F    RETLW  1Fh
0036 3400    RETLW  00h
0037 3422    RETLW  22h
0038 3400    RETLW  00h
0039 3425    RETLW  25h
003A 3400    RETLW  00h
003B 3429    RETLW  29h
003C 3400    RETLW  00h
003D 342D    RETLW  2Dh
003E 3400    RETLW  00h
003F 3432    RETLW  32h
0040 3400    RETLW  00h
0041 3437    RETLW  37h
0042 3400    RETLW  00h
0043 343C    RETLW  3Ch
0044 3400    RETLW  00h
0045 3442    RETLW  42h
0046 3400    RETLW  00h
0047 3449    RETLW  49h
0048 3400    RETLW  00h
0049 3450    RETLW  50h
004A 3400    RETLW  00h
004B 3458    RETLW  58h
004C 3400    RETLW  00h
004D 3461    RETLW  61h
004E 3400    RETLW  00h
004F 346B    RETLW  6Bh
0050 3400    RETLW  00h
0051 3476    RETLW  76h
0052 3400    RETLW  00h
0053 3482    RETLW  82h
0054 3400    RETLW  00h
0055 348F    RETLW  8Fh
0056 3400    RETLW  00h
0057 349D    RETLW  9Dh
0058 3400    RETLW  00h
0059 34AD    RETLW  ADh
005A 3400    RETLW  00h
005B 34BE    RETLW  BEh
005C 3400    RETLW  00h
005D 34D1    RETLW  D1h
005E 3400    RETLW  00h
005F 34E6    RETLW  E6h
0060 3400    RETLW  00h
0061 34FD    RETLW  FDh
0062 3400    RETLW  00h
0063 3417    RETLW  17h
0064 3401    RETLW  01h
0065 3433    RETLW  33h
0066 3401    RETLW  01h
0067 3451    RETLW  51h
0068 3401    RETLW  01h
0069 3473    RETLW  73h
006A 3401    RETLW  01h
006B 3498    RETLW  98h
006C 3401    RETLW  01h
006D 34C1    RETLW  C1h
006E 3401    RETLW  01h
006F 34EE    RETLW  EEh
0070 3401    RETLW 01h
0071 3420    RETLW 20h
0072 3402    RETLW 02h
0073 3456    RETLW 56h
0074 3402    RETLW 02h
0075 3492    RETLW 92h
0076 3402    RETLW 02h
0077 34D4    RETLW D4h
0078 3402    RETLW 02h
0079 341C    RETLW 1Ch
007A 3403    RETLW 03h
007B 340C    RETLW 0Ch
007C 3403    RETLW 03h
007D 34C3    RETLW C3h
007E 3403    RETLW 03h
007F 3424    RETLW 24h
0080 3404    RETLW 04h
0081 348E    RETLW 8Eh
0082 3404    RETLW 04h
0083 3402    RETLW 02h
0084 3405    RETLW 05h
0085 3483    RETLW 83h
0086 3405    RETLW 05h
0087 3410    RETLW 10h
0088 3406    RETLW 06h
0089 34AB    RETLW ABh
008A 3406    RETLW 06h
008B 3456    RETLW 56h
008C 3407    RETLW 07h
008D 3412    RETLW 12h
008E 3408    RETLW 08h
008F 34E0    RETLW E0h
0090 3408    RETLW 08h
0091 34C3    RETLW C3h
0092 3409    RETLW 09h
0093 34BD    RETLW BDh
0094 340A    RETLW 0Ah
0095 34D0    RETLW D0h
0096 340B    RETLW 0Bh
0097 34FF    RETLW FFh
0098 340C    RETLW 0Ch
0099 344C    RETLW 4Ch
009A 340E    RETLW 0Eh
009B 34BA    RETLW BAh
009C 340F    RETLW 0Fh
009D 344C    RETLW 4Ch
009E 3411    RETLW 11h
009F 3407    RETLW 07h
00A0 3413    RETLW 13h
00A1 34EE    RETLW EEh
00A2 3414    RETLW 14h
00A3 3406    RETLW 06h
00A4 3417    RETLW 17h
00A5 3454    RETLW 54h
00A6 3419    RETLW 19h
00A7 34DC    RETLW DCh
00A8 341B    RETLW 1Bh
00A9 34A5    RETLW A5h
00AA 341E    RETLW 1Eh
00AB 34B6    RETLW B6h
00AC 3421    RETLW 21h
00AD 3415    RETLW 15h
00AE 3425    RETLW 25h
00AF 34CA    RETLW CAh
00B0 3428    RETLW 28h
00B1 34DF    RETLW DFh
00B2 342C    RETLW 2Ch
00B3 345B    RETLW 58h
00B4 3431    RETLW 31h
signed long diff; /* Difference between sample and predicted sample */

long step; /* Quantizer step size */

signed long predsample; /* Output of ADPCM predictor */

signed long diffq; /* Dequantized predicted difference */

int index; /* Index into step size table */

char ADPCMEncoder( signed long sample )
{
    /* Restore previous values of predicted sample and quantizer step size index */
    predsample = state.prevsample;
    index = state.previndex;

    int code; /* ADPCM output value */

    /* Restore previous values of predicted sample and quantizer step size index */
    predsample = state.prevsample;
    index = state.previndex;
/* Compute the difference between the actual sample (sample) and the predicted sample (predsample) */
diff = sample - predsample;

00E4 0832    MOVF   32,W
00E5 0237    SUBWF  37,W
00E6 00AE    MOVWF  2E
00E7 0833    MOVF  33,W
00E8 1C03    BTFSS  STATUS,C
00E9 3E01    ADDLW  01h
00EA 0238    SUBWF  38,W
00EB 00AF    MOVWF  2F
00EC 1BAF    BTFSC  2F,7
00ED 28F0    GOTO  00F0h
00EE 01B9    CLRF   39
00EF 28F9    GOTO  00F9h

else
{
00F0 3008    MOVLW  08h
00F1 00B9    MOVWF  39
00F2 09AF    COMF  2F
00F3 09AE    INCF  2E
00F4 1D03    BTFSS  STATUS,Z
00F5 28F9    GOTO  00F9h
00F6 1283    BCF  STATUS,RP0
00F7 0AAF    INCF  2F
00F8 1283    BCF  STATUS,RP0
00FA 0830    MOVF  30,W
00FB 00B4    MOVWF  34
00FC 0831    MOVF  31,W
00FD 00B5    MOVWF  35
00FE 0D35    RLF  35,W
00FF 0830    MOVF  30,W
0100 00B4    MOVWF  34
0101 0D35    RLF  35,W
0102 0CB5    RRF  35
0103 0CB4    RRF  34
0104 0D35    RLF  35,W
0105 0CB5    RRF  35
0106 0CB4    RRF  34
0107 0830    MOVF  30,W
0108 022E    SUBWF  2E,W
0109 3080    MOVWL  80h
010A 062F    XORWF  2F,W
010B 00A9    MOVWF  29
010C 3080    MOVWL  80h
010D 0631    XORWF  31,W
010E 1C03    BTFSS  STATUS,C
010F 3E01    ADDLW  01h
0110 0229    SUBWF  29,W
0111 1C03    BTFSS  STATUS,C
0112 2921    GOTO  0121h

if( diff >= step )
0113 1283    BCF  STATUS,RP0  code |= 4;
0114 1539    BSF  39,2
0115 0830    MOVF  30,W  diff -= step;
0116 02AE    SUBWF  2E
0117 0831    MOVF  31,W
0118 1C03    BTFSS  STATUS,C
0119 3E01    ADDLW  01h
011A 02AF    SUBWF  2F
011B 0830    MOVF  30,W  diffq -= step;
011C 07B4    ADDWF  34
011D 0831    MOVF  31,W
011E 1803    BTFS C  STATUS,C
011F 3E01    ADDLW  01h
0120 07B5 ADDWF 35

0121 1283 BCF STATUS,RP0 step >>= 1;
0122 0D31 RLF 31,W
0123 0CB1 RRF 31
0124 0CB0 RRF 30
0125 0830 MOVF 30,W if( diff >= step )
0126 022E SUBWF 2E,W
0127 3080 MOVLW 80h
0128 062F XORWF 2F,W
0129 00A9 MOVWF 29
012A 3080 MOVLW 80h
012B 0631 XORWF 31,W
012C 1C03 BTFS STATUS,C
012D 3E01 ADDLW 01h
012E 0229 SUBWF 29,W
012F 1C03 BTFS STATUS,C
0130 293F GOTO 013Fh
0131 0131 1283 BCF STATUS,RP0 code |= 2;
0132 14B9 BSF 39,1
0133 0830 MOVF 30,W diff -= step;
0134 02AE SUBWF 2E
0135 0831 MOVF 31,W
0136 1C03 BTFS STATUS,C
0137 3E01 ADDLW 01h
0138 02AF SUBWF 2F
0139 0830 MOVF 30,W diffq += step;
013A 07B4 ADDWF 34
013B 0831 MOVF 31,W
013C 1803 BTFS STATUS,C
013D 3E01 ADDLW 01h
013E 07B5 ADDWF 35

013F 1283 BCF STATUS,RP0 step >>= 1;
0140 0D31 RLF 31,W
0141 0CB1 RRF 31
0142 0CB0 RRF 30
0143 0830 MOVF 30,W if( diff >= step )
0144 022E SUBWF 2E,W
0145 3080 MOVLW 80h
0146 062F XORWF 2F,W
0147 00A9 MOVWF 29
0148 3080 MOVLW 80h
0149 0631 XORWF 31,W
014A 1C03 BTFS STATUS,C
014B 3E01 ADDLW 01h
014C 0229 SUBWF 29,W
014D 1C03 BTFS STATUS,C
014E 2957 GOTO 0157h
014F 014F 1283 BCF STATUS,RP0 code |= 1;
0150 1439 BSF 39,0
0151 0830 MOVF 30,W diffq += step;
0152 07B4 ADDWF 34
0153 0831 MOVF 31,W
0154 1803 BTFS STATUS,C
0155 3E01 ADDLW 01h
0156 07B5 ADDWF 35

/* Fixed predictor computes new predicted sample by adding the old predicted sample to predicted difference */
0157 1283 BCF STATUS,RP0 if( code & 8 )
0158 1DB9 BTFS 39,3
0159 2961 GOTO 0161h
015A 0834 MOVF 34,W predsamp += diffq;
015B 02B2 SUBWF 32
015C 0835 MOVF 35,W
015D 1C03    BTFSS  STATUS,C
015E 0834    MOVF  34,W
015F 02B3    SUBWF  33
0160 2967    GOTO   0167h
0161 3E01    ADDLW  01h
0162 07B2    ADDWF  32
0163 0835    MOVF  35,W
0164 1803    BTFSC  STATUS,C
0165 3E01    ADDLW  01h
0166 07B3    ADDWF  33

/* Check for overflow of the new predicted sample */
if( predsample > 32767 )
    0167 3001    MOVLW  01h
0168 0732    ADDWF  32,W
0169 00A8    MOVWF  28
016A 3080    MOVLW  80h
016B 0633    XORWF  33,W
016C 00A7    MOVWF  27
016D 3000    MOVLW  00h
016E 1803    BTFSC  STATUS,C
016F 3001    MOVLW  01h
0170 0727    ADDWF  27,W
0171 0428    IORWF  28,W
0172 1B03    BTFSS  STATUS,Z
0173 1C03    BTFSS  STATUS,C
0174 29B7    GOTO   017Bh
0175 30FF    MOVLW  FFh
0176 1283    BCF    STATUS,RP0
0177 00B2    MOVWF  32
0178 307F    MOVLW  7fh
0179 00B3    MOVWF  33
017A 2987    GOTO   0187h
/* Find new quantizer stepsize index by adding the
old index to a table lookup using the ADPCM code */
017B 3080    MOVLW  80h
017C 1283    BCF    STATUS,RP0
017D 0633    XORWF  33,W
017E 00A7    MOVWF  27
017F 3000    MOVLW  00h
0180 0227    SUBWF  27,W
0181 1803    BTFSC  STATUS,C
0182 2987    GOTO   0187h
0183 1283    BCF    STATUS,RP0
0184 01B2    CLRWF  32
0185 3080    MOVLW  80h
0186 00B3    MOVWF  33

0187 1BB6    BTFSF  36,7
0188 1283    BCF    STATUS,RP0
0189 0839    MOVWF  39,W
018A 2005    CALL   0005h
018B 1283    BCF    STATUS,RP0
018C 07B6    ADDWF  36

/* Check for overflow of the new quantizer step size
index */
018D 1BB6    BTFSF  36,7
018E 01B6    CLRWF  36
018F 3080    MOVLW  80h
0190 1283    BCF    STATUS,RP0
0191 0636    XORWF  36,W
0192 00A7    MOVWF  27
0193 3008    MOVLW  D8h
0194 0227    SUBWF  27,W
0195 1D03    BTFSF  STATUS,Z
0196 1C03    BTFSF  STATUS,C
0197 29B9    GOTO   019Bh
0198 3058    MOVLW  58h
0199 1283    BCF    STATUS,RP0
019A 00B6    MOVWF  36

/* Save the predicted sample and quantizer step size
index for next iteration */

signed long ADPCMDecoder(char code)
{
    return (code & 0x0f);
}

/***********************************************************/
/*    ADPCMDecoder - ADPCM decoder routine                */
/***********************************************************/

/* Input Variables: */
/* char code - 8-bit number containing the 4-bit ADPCM code */
/* Return Variable: */
/* signed long - 16-bit signed speech sample */
/***********************************************************/

/* Restore previous values of predicted sample and quantizer step size index */
predsample = state.prevsample;
index = state.previndex;

/* Find quantizer step size from lookup table using index */
diffq = step >> 3;

/* Inverse quantize the ADPCM code into a difference using the quantizer step size */
if( code & 4 )
    GOTO 01D3h
diffq += step;

if( code & 2 )
    diffq += step >> 1;

if( code & 1 )
    diffq += step >> 2;

/* Add the difference to the predicted sample */
if( code & 8 )
    predsample -= diffq;
else
    predsample += diffq;

/* Check for overflow of the new predicted sample */
if( predsample > 32767 )
    /* Add the difference to the predicted sample */
/* Find new quantizer step size by adding the old index and a table lookup using the ADPCM code */
index += IndexTable[code];

/* Check for overflow of the new quantizer step size index */
if( index < 0 )
    index = 0;
else if( index > 88 )
    index = 88;

/* Save predicted sample and quantizer step size index for next iteration */
state.prevsample = (short)predsample;
state.previndex = index;
return( predsample );
APPENDIX D: INTERACTIVE MULTIMEDIA ASSOCIATION INFORMATION

The IMA ADPCM Reference Algorithm is contained in the Digital Audio Doc-Pac. The Doc-Pac contains the following information:

- **About IMA Digital Audio** - describes the IMA's activities in digital audio
- **IMA Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems** (version 3.0). This document contains the official technical recommendations including the ADPCM algorithm
- **IMA Digital Audio Special Edition Proceedings** This document contains a detailed notes of the evaluation process, code fragments and testing methodologies.
- Floppy disk with code fragments.
- Contact information for companies supporting IMA ADPCM.

The IMA can be reached at:
Interactive Multimedia Association
48 Maryland Avenue, Suite 202
Annapolis, MD 21401-8011 USA
Tel: (410)-626-1380
Fax: (410)-263-0590
http://www.ima.org
ADPCM using a PIC16C72

Each EPR OM holds 65536 bytes. Each byte holds two ADPCM codes. Every 8000 ADPCM codes is 1 second. Therefore, each EPROM holds 16,534 seconds (65536 x 2 x 8000 = 16,534).

4th Order Butterworth LPF @ 4 kHz

Selects the analog voltage +5V or +9V using a 20k pot.

Speaker Amplifier

Voltage Inverter

Power Jack

© 1997 Microchip Technology Inc.
APPENDIX F: APPLICATION FIRMWARE BLOCK DIAGRAM AND LISTING

Initialize the PIC16C72

Enable EPROM #1, Timer2 Interrupt, PEIE, and GIE

Start Pressed?

Clear Address, ADPCM structure, and enable Timer2 and PWM

Yes

Read EPROM #1 and decode the upper 4-bits

Send upper 9-bits to PWM

Decode lower 4-bits of byte read from EPROM #1

Send upper 9-bits to PWM

Increment Address and EPROM Address

Address overflow?

Yes

Reset Address and EPROM Address

Disable EPROM #1 and enable EPROM #2

No

Make PORTA all digital outputs

Set CE, OE1, and OE2 high

Set CS, SCK, and DIN high

Set ECLK high and RST low

Set CCP for 32 kHz period

Read EPROM #2 and decode the upper 4-bits

Send upper 9-bits to PWM

Decode lower 4-bits of byte read from EPROM #2

Send upper 9-bits to PWM

Increment Address and EPROM Address

Address overflow?

Yes

Reset EPROM Address, disable EPROMs and PWM

No
#pragma option v
#include <16c72.h>
#pragma option +l;
#define MAXROM 2043

0FFB 0005 0FFB
#pragma memory ROM [MAXROM] @ 0x05;

0020 0060
#pragma memory RAM [0x96]   @ 0x20;

00A0 0060
#pragma memory RAM [0x32]   @ 0xA0;

#pragma option +l;
#include "adpcm.h"

struct ADPCMState {
    signed long prevsample;       /* Predicted sample */
    int         previndex; /*Index into step size table*/
};
/* Function prototype for the ADPCM Encoder routine */
signed long ADPCMDecoder( char );

/* Function prototype for the ADPCM Decoder routine */
char ADPCMEncoder( signed long );

// Defines for PORTA
#define CE   0   // Chip Enable for EPROMS
#define OE1  1   // Output Enable for EPROM #1
#define OE2  2   // Output Enable for EPROM #2

// Defines for PORTC
#define STRT 0   // Start Button input
#define PWM  2   // PWM output
#define RST  6   // EPROM counter reset
#define ECLK 7   // EPROM counter clock

// defines for PORTD
#define ADPCM 3  // ADPCM output

// Defines for PORTC
#define STRT 0   // Start Button input
#define PWM  2   // PWM output
#define RST  6   // EPROM counter reset
#define ECLK 7   // EPROM counter clock

#include "adpcm.c"

//************************\\
// Insert Appendix C here  \\
//*************************

/*****************************************************************************
*    Init72 - This function initializes PORTA, PORTC, and PWM              *
*****************************************************************************/
void Init72(void)
{
 0247 3008    MOVLW  08h               OPTION = 8;
 0248 1683    BSF    STATUS,RP0
 0249 0081    MOVWF  TMR0
 024A 3007    MOVLW  07h               ADCON1 = 7;    // Make PORTA all digital
 024B 009F    MOVWF  ADCON0
 024C 1283    BCF    STATUS,RP0        PORTA = 7;     // Set CE,OE1,OE2
 024D 0085    MOVWF  PORTA
 024E 1683    BSF    STATUS,RP0        TRISA = 0;     // Make PORTA all outputs
 024F 0185    CLRF   PORTA
 0250 3040    MOVLW  40h               PORTC = 0x40;  // Set ECLK, all outputs
 0251 1283    BCF    STATUS,RP0
 0252 0087    MOVWF  PORTC
 0253 3005    MOVLW  05h               TRISC = 0x05; // Clear RST output, STRT input
 0254 1683    BSF    STATUS,RP0
 0255 0087    MOVWF  PORTC
 0256 309B    MOVLW  9Bh               PR2 = 155;     // Set PWM for 32kHz period
 0257 0092    MOVWF  T2CON
 0258 1283    BCF    STATUS,RP0
 0259 0195    CLR    CCPR1L
 025A 3018    MOVLW  18h               T2CON = 0x18; // Set Timer2 postscaler to 4
 025B 0092    MOVWF  T2CON
 025C 0008    RETURN                   return;
}

/*****************************************************************************
*    main - This function controls everything, kinda like a god.             *
*****************************************************************************/

* Input Variables:
void main(void) {
    char code;              // EPROM byte
    signed long sample;    // decoded sample
    unsigned long pwmout;  // temporary variable for PWM

    CALL 0247h            // Initialize the PIC16C72
    MOVLW 01h              Wait = 1;
    BCF STATUS,RP0
    MOVWF 2A
    BCF PORTC,6            PORTC.RST = 1;     // Reset the EPROM counter
    NOP();
    NOP();
    BCF PORTC,6            PORTC.RST = 0;
    NOP();
    NOP();
    PORTA.OE1 = 1;         // Enable EPROM #1
    NOP();
    NOP();
    PORTA.CE = 0;
    NOP();
    NOP();
    PIR1.TMR2IF = 0;       // Enable Timer2 Interrupts
    PIE1.TMR2IE = 1;
    PIR1.TMR2IE = 1;
    INTCON.PEIE = 1;       // Enable PEIE and GIE
    INTCON.GIE = 1;

    while(1) {
        while(PORTC.STRT);  // Wait for Start button to be
        while(!PORTC.STRT); // pressed then released
        Address = 0;        // Clear EPROM address
        CLRF 25
        CLRF 26
        CLRF 2B
        state.prevsample = 0; // Clear ADPCM structure
        CLRF 2C
        CLRF 2D
        state.previndex = 0;

        TRISC.PWM = 0;       // Make PWM an output
        T2CON.TMR2ON = 1;    // Enable Timer2
        CCP1CON = 0x0f;     // Enable PWM

        do                 // Decode upper 4 bits of code
            code = PORTB;
            MOVF PORTB,W
            MOVWF CCP1CON
        
        CCP1CON = (code>>4)&0x0f;

        MOVWF 2B
        SWAPF 2B
        CCP1CON = 0x0f;     // Enable PWM
        CCP1CON = 0x0f;

    }
028D 21A5    CALL   01A5h
028E 1283    BCF   STATUS,RP0
028F 00BC    MOVWF  3C
0290 0804    MOVF   FSR,W
0291 00BD    MOVWF  3D
0292 1283    BCF   STATUS,RP0
0293 082A    MOVF   2A,W
0294 3800    IORLW  00h
0295 1D03    BTFSS STATUS,Z
0296 2A92    GOTO   0292h
0297 3001    MOVWL  01h  while(Wait);  // Wait for 8kHz to output
0298 1283    BCF   STATUS,RP0
0299 00AA    MOVWF  2A
029A 083C    MOVF   3C,W  pwmout = sample; // Write to PWM
029B 00BE    MOVWF  3E
029C 083D    MOVF   3D,W
029D 00BF    MOVWF  3F
029E 1FB0    BTFSS 3D,7  if(sample<0)   // Add offset to
029F 2A44    GOTO   02A4h
02A0 3080    MOVWL  80h
02A1 023F    MOVF   3F,W
02A2 00BF    MOVWF  3F
02A3 2A62    GOTO   02A6h  else
02A4 3080    MOVWL  80h  pwmout += 0x8000; // sample
02A5 07BF    ADDWF  3F
02A6 083F    MOVF   3F,W  CCP1CON = (pwmout>>9)&0x007f; // Write 7 bits
02A7 00A7    MOVWF  27
02A8 0CA7    RRF    27
02A9 307F    MOVLW  7Fh
02AA 0527    ANDWF  27,W
02AB 0095    MOVWF  CCP1CON
02AC 30CF    MOVWL  CFh  // to CCP1CON
02AD 0597    ANDWF  CCP1CON
02AE 183F    BTFSC  3F,0  if((pwmout&0x0100)  // Write 2 bits
02AF 1697    BSF    CCP1CON,5  CCP1CON.5 = 1;  // to CCP1CON
02B0 1283    BCF   STATUS,RP0  if((pwmout&0x0080)  // to get 9-bit
02B1 1BBE    BTFSC  3E,7  CCP1CON.4 = 1;  // PWM
02B2 1617    BSF    CCP1CON,4
02B3 300F    MOVLW  0Fh  // Decode lower 4 bits of code
02B4 1283    BCF   STATUS,RP0
02B5 053B    ANDWF  3B,W
02B6 118A    BCF   PCLATH,3
02B7 21A5    CALL   01A5h
02B8 118A    BCF   PCLATH,3
02B9 1283    BCF   STATUS,RP0
02BA 00BC    MOVWF  3C
02BB 0804    MOVF   FSR,W
02BC 00BD    MOVWF  3D
02BD 1283    BCF   STATUS,RP0
02BE 082A    MOVF   2A,W
02BF 3800    IORLW  00h
02C0 1D03    BTFSS STATUS,Z
02C1 2A92    GOTO   0292h
02C2 3001    MOVWL  01h  while(Wait);  // Wait for 8kHz to output
02C3 1283    BCF   STATUS,RP0
02C4 00AA    MOVWF  2A
02C5 083C    MOVF   3C,W  pwmout = sample; // Write to PWM
02C6 00BE    MOVWF  3E
02C7 083D    MOVF   3D,W
02C8 00BF    MOVWF  3F
02C9 1FB0    BTFSS 3D,7  if(sample<0)   // Add offset to
02CA 2AFC    GOTO   02CFh
02CB 3080    MOVWL  80h  pwmout += 0x8000; // sample
02CC 023F    SUBWF  3F,W
02CD 00BF MOVWF 3F
02CE 2AD1 GOTO 02D1h
02CF 3080 MOVLO 80h
02D0 07BF ADDWF 3F
02D1 083F MOVF 3F,W
02D2 00A7 MOVWF 27
02D3 0CA7 RRF 27
02D4 307F MOVLO 7Fh
02D5 0527 ANDWF 27,W
02D6 0995 MOVWF CCP1R1
02D7 30CF MOVLO CFh
02D8 0597 ANDWF CCP1CON
02D9 183F BTFSC 3F,0
02DA 1697 BSF CCP1CON,5
02DB 1283 BCF STATUS,RP0
02DC 1BBE BTFSC 3E,7
02DD 1617 BSF CCP1CON,4
02DE 1283 BCF STATUS,RP0
02DF 0AA5 INCF 25
02E0 1903 BTFSC STATUS,2
02E1 0AA6 INCF 26
02E2 1787 BSF PORTC,7
02E3 0000 NOP
02E4 0000 NOP
02E5 0000 NOP
02E6 0000 NOP
02E7 0000 NOP
02E8 0826 MOVF 26,W
02E9 0425 IORWF 25,W
02EA 118A BCF PCLATH,3
02EB 1D03 GOTO 0284h
02ED 1283 BCF STATUS,RP0
02EE 1405 BSF PORTA,0
02EF 0000 NOP
02F0 0000 NOP
02F1 1485 BSF PORTA,1
02F2 0000 NOP
02F3 0000 NOP
02F4 1705 BSF PORTA,6
02F5 0000 NOP
02F6 0000 NOP
02F7 1305 BCF PORTA,6
02F8 0000 NOP
02F9 0000 NOP
02FA 1105 BCF PORTA,2
02FB 0000 NOP
02FC 0000 NOP
02FD 1005 BCF PORTA,0
02FE 0000 NOP
02FF 0000 NOP
0300 01A5 CLRF 25
0301 01A6 CLRF 26
0302 0000 NOP
0303 0000 NOP
0304 1283 BCF STATUS,RP0
do {  // Decode upper 4 bits of code
0305 0806 MOVFW PORTB,W
0306 083B MOVWF 3B
0307 083B MOVF 3B,W
0308 00A7 MOVWF 27
0309 1283 BCF STATUS,RP0
code = PORTB;
030A 0806 MOVFW PORTB,W
030B 083B MOVWF 3B
030C 083B MOVF 3B,W
030D 00A7 MOVWF 27
030E 1283 BCF STATUS,RP0
AN643

0309 0EA7    SWAPF 27
030A 300F    MOVWF 0Fh
030B 0527    ANDWF 27,W
030C 390F    ANDLW 0Fh
030D 21A5    CALL 01A5h
030E 1283    BCF STATUS,RP0
030F 00BC    MOVWF 3C
0310 0804    MOVF FSR,W
0311 00BD    MOVWF 3D
0312 1283    BCF STATUS,RP0 while(Wait); // Wait for 8kHz
0313 082A    MOVF 2A,W
0314 3800    IORLW 00h
0315 1D03    BTFSS STATUS,Z
0316 2B12    GOTO 0312h
0317 3001    MOVVL 01h Wait = 1;
0318 1283    BCF STATUS,RP0
0319 00AA    MOVWF 2A
031A 083C    MOVF 3C,W
031B 00BE    MOVWF 3E
031C 083D    MOVF 3D,W
031D 00BF    MOVWF 3F
031E 1FBD    BTFSS 3D,7 if(sample<0) // Add offset to
031F 2B24    GOTO 0324h
0320 3080    MOVVL 80h
0321 023F    SUBWF 2A,W
0322 3800    IORLW 00h
0323 1D03    BTFSS STATUS,Z
0324 2B3D    GOTO 033Dh
0325 00AA    MOVWF 2A
0326 082A    MOVF 2A,W
0327 00A7    MOVWF 27
0328 0CA7    RRF 27
0329 307F    MOVVL 7Fh
032A 0527    ANDWF 3B,W
032B 118A    BCF PCLATH,3
032C 21A5    CALL 01A5h
032D 00BC    MOVWF 3C
032E 0804    MOVF FSR,W
032F 00BD    MOVWF 3D
0330 1283    BCF STATUS,RP0 while(Wait); // Wait for 8kHz
0331 082A    MOVF 2A,W
0332 3800    IORLW 00h
0333 1D03    BTFSS STATUS,Z
0334 2B3D    GOTO 033Dh
0335 00AA    MOVWF 2A
0336 083C    MOVF 3C,W
0337 00BE    MOVWF 3E
0338 083D    MOVF 3D,W
0339 00BF    MOVWF 3F
033A 1FBD    BTFSS 3D,7 if(sample<0) // Add offset to
034D 00BF  MOVWF  3F
034E 083F  MOVLW  80h                     pwmout -= 0x8000; // sample
034F 0080  MOVLW  80h                     pwmout += 0x8000;
0350 07BF  ADDWF  3F
0351 0351h GOTO   0351h                  else
0352 00A7  MOVF  3F
0353 00BF  MOVWF  3F
0354 0307F MOVLW  7Fh
0355 0527  ANDWF  27
0356 0095  MOVLW CCP1CON
0357 30CF  MOVLW CFh
0358 0597  ANDWF CCP1CON
0359 183F  BTFS  PORTC,7
0360 1903  BTFSC  STATUS,Z
0361 118A  BCF  CCP1CON
0362 1283  BCF  STATUS,RP0
0363 0000  NOP
0364 0000  NOP
0365 0826  MOVF  26,W                 } while(Address);  // Has Address overflowed?
0366 0425  IORWF  25,W
0367 118A  BCF  PCLATH,3
0368 1D03  BTFSS  STATUS,Z
0369 2B04  GOTO   0304h
036A 1283  BCF  STATUS,RP0
036B 1112  BCF  T2CON,2
036C 0197  CLRPF CCP1CON
036D 1683  BSF  STATUS,RP0
036E 1507  BSF  PORTC,2
036F 1283  BCF  STATUS,RP0
0370 1405  BSF  PORTA,0
0371 0000  NOP
0372 0000  NOP
0373 1705  BSF  PORTA,6
0374 0000  NOP
0375 0000  NOP
0376 1505  BSF  PORTA,2
0377 0000  NOP
0378 0000  NOP
0379 1705  BSF  PORTA,6
037A 0000  NOP
037B 0000  NOP
037C 1305  BCF  PORTA,6
037D 0000  NOP
037E 0000  NOP
037F 1085  BCF  PORTA,1
0380 0000  NOP
0381 0000  NOP
0382 1005  BCF  PORTA,0
0383 0272h GOTO   0272h
0384 0008  RETURN               }

/*******************************************************************************
*  __INT - This is the interrupt service routine. Only Timer 2 overflow      *
*  is implemented.                                                          *
*******************************************************************************/
Input Variables:

None

Return Variables:

None

******************************************************************************/

0004 2B85    GOTO   0385h    void __INT(void)
0385          {
0385 1283    BCF    STATUS,RP0    if(PIR1.TMR2IF)    // Timer2 overflow interrupt
0386 1C8C    BTFSS  PIR1,1
0387 2B8A    GOTO   038Ah
0388          {
0388 01AA    CLRF   2A    Wait = 0;
0389 108C    BCF    PIR1,1    PIR1.TMR2IF = 0;    // Clear flag
038A 0009    RETFIE    return;
038B          }

ROM USAGE MAP

0000 to 0002    0004 to 038A
Total ROM used 038A

Errors    :    0
Warnings :    0
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pcadpcm.h"

/**********************
* Usage - this routine prints a how to message for the pcspeech prgm *
************************/

void Usage(void)
{
    printf("ADPCM Encoder/Decoder -- usage:\n");
    printf("Encoder = pcspeech e infile outfile\n");
    printf("Decoder = pcspeech d infile outfile\n");
    exit(1);
}

/**********************
* main - controls file I/O and ADPCM calls *
************************/

void main(
    int   argc,
    char  **argv)
{
{  int       which;
  short    sample;
unsigned char   code;
int         n;
struct ADPCMstate state;
FILE   *fpin;
FILE    *fpout;

state.prevsample=0;
state.previndex=0;

/* Determine if this is an encode or decode operation */
if(argc < 1)
  Usage();
else if( strcmp(argv[1],"e")==0 || strcmp(argv[1],"E")==0 )
  which = 0;
else if( strcmp(argv[1],"d")==0 || strcmp(argv[1],"D")==0 )
  which = 1;
argc--;
argv++;

/* Open input file for processing */
if(argc < 1)
  Usage();
else if( (fpin=fopen(argv[1],"rb"))==NULL )
{
  printf("ADPCM Encoder/Decoder\n");
  printf("ERROR: Could not open %s for input\n",argv[1]);
  exit(1);
}
argc--;
argv++;

/* Open output file */
if(argc < 1)
{
  fclose(fpin);
  Usage();
}
else if( (fpout=fopen(argv[1],"wb"))==NULL )
{
  fclose(fpin);
  printf("ADPCM Encoder/Decoder\n");
  printf("ERROR: Could not open %s for output\n",argv[1]);
  exit(1);
}

// ADPCM Decoder selected
if(which)
{
  printf("ADPCM Decoding in progress\n");
/* Read and unpack input codes and process them */
while (fread(&code, sizeof (char), 1, fpin) == 1)
  {
    // Send the upper 4-bits of code to decoder
    sample = ADPCMDecoder((code>>4)&0x0f, &state);
    // Write sample for upper 4-bits of code
    fwrite(&sample, sizeof(short), 1, fpout);
    // Send the lower 4-bits of code to decoder
    sample = ADPCMDecoder(code&0x0f, &state);
    // Write sample for lower 4-bits of code
    fwrite(&sample, sizeof(short), 1, fpout);
  }
}
```c
// ADPCM Encoder selected
else
{
    printf("ADPCM Encoding in progress\n");
    /* Read input file and process */
    while (fread(&sample, sizeof(short), 1, fpin) == 1)
    {
        // Encode sample into lower 4-bits of code
        code = ADPCMEncoder(sample,&state);
        // Move ADPCM code to upper 4-bits
        code = (code << 4) & 0xf0;
        // Read new sample from file
        if(fread(&sample,sizeof(short),1,fpin)==0)
        {
            // No more samples, write code to file
            fwrite(&code,sizeof(char),1,fpout);
            break;
        }
        // Encode sample and save in lower 4-bits of code
        code |= ADPCMEncoder(sample,&state);
        // Write code to file, code contains 2 ADPCM codes
        fwrite(&code, sizeof (char), 1, fpout);
    }
}
fclose(fpin);
fclose(fpout);
}

PCADPCM.H

/*******************************************************************************
*       Filename:  PCADPCM.H                                                 *
* *******************************************************************************
*       Author:         Rodger Richey                                        *
*       Title:         Senior Applications Engineer                         *
*       Company:        Microchip Technology Incorporated                    *
*       Revision:       0                                                    *
*       Date:         1-11-96                                              *
*       Compiled using Borland C+ Version 3.1                                *
*******************************************************************************/
struct ADPCMstate {
    short prevsample; /* Predicted sample */
    int previndex; /* Index into step size table */
};

/* Function prototype for the ADPCM Encoder routine */
char ADPCMEncoder(short , struct ADPCMstate *);

/* Function prototype for the ADPCM Decoder routine */
int ADPCMDecoder(char , struct ADPCMstate *);

PCADPCM.C

/*******************************************************************************
*       Filename:  PCADPCM.C                                                 *
* *******************************************************************************
*       Author:         Rodger Richey                                        *
*       Title:         Senior Applications Engineer                         *
*       Company:        Microchip Technology Incorporated                    *
*       Revision:       0                                                    *
*       Date:         1-11-96                                              *
*       Compiled using Borland C+ Version 3.1                                *
*******************************************************************************/
```

---

© 1997 Microchip Technology Inc.
/* Include files: */
* stdio.h - Standard input/output header file
* pcadpcm.h - ADPCM related information header file (Rev0)
*
* This file contains the ADPCM encode and decode routines. These
* routines were obtained from the Interactive Multimedia Association's
* Reference ADPCM algorithm. This algorithm was first implemented by
* Intel/DVI.
*
 ***************************************************************************/
#include <stdio.h>
#include "pcadpcm.h"

/* Table of index changes */
signed char IndexTable[16] = {
-1, -1, -1, -1, 2, 4, 6, 8,
-1, -1, -1, -1, 2, 4, 6, 8,
};

/* Quantizer step size lookup table */
int StepSizeTable[89] = {
7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

/**************************************************************************** *
* ADPCMEncoder - ADPCM encoder routine                                    *
*****************************************************************************/
char ADPCMEncoder( short sample , struct ADPCMstate *state )
{
    int code; /* ADPCM output value */
    int diff; /* Difference between sample and the predicted sample */
    int step; /* Quantizer step size */
    int predsample; /* Output of ADPCM predictor */
    int diffq; /* Dequantized predicted difference */
    int index; /* Index into step size table */

    /* Restore previous values of predicted sample and quantizer step size
     * index */
    predsample = (int)(state->prevsample);
    index = state->previndex;
    step = StepSizeTable[index];

    /* Compute the difference between the actual sample (sample) and the
     * the predicted sample (predsample) */
    if(diff > 0)
        code = 0;
else
{
  code = 8;
  diff = -diff;
}

/* Quantize the difference into the 4-bit ADPCM code using the
   the quantizer step size */
/* Inverse quantize the ADPCM code into a predicted difference
   using the quantizer step size */
diffq = step >> 3;
if( diff >= step )
{
  code |= 4;
  diff -= step;
  diffq += step;
}
step >>= 1;
if( diff >= step )
{
  code |= 2;
  diff -= step;
  diffq += step;
}
step >>= 1;
if( diff >= step )
{
  code |= 1;
  diffq += step;
}
/* Fixed predictor computes new predicted sample by adding the
   old predicted sample to predicted difference */
if( code & 8 )
preds = -diffq;
else
  pred = diffq;
/* Check for overflow of the new predicted sample */
if( pred > 32767 )
pred = 32767;
else if( pred < -32767 )
pred = -32767;
/* Find new quantizer stepsize index by adding the old index
   to a table lookup using the ADPCM code */
index += IndexTable[code];
/* Check for overflow of the new quantizer step size index */
if( index < 0 )
  index = 0;
if( index > 88 )
  index = 88;
/* Save the predicted sample and quantizer step size index for
   next iteration */
state->prevsample = (short)pred;
state->previndex = index;
/* Return the new ADPCM code */
return ( code & 0x0f );
}

int ADPCMDecoder(char code, struct ADPCMstate *state)
{
    int step; /* Quantizer step size */
    int predsample; /* Output of ADPCM predictor */
    int diffq; /* Dequantized predicted difference */
    int index; /* Index into step size table */

    /* Restore previous values of predicted sample and quantizer step size index */
predsample = (int)(state->prevsample);
index = state->previndex;

    /* Find quantizer step size from lookup table using index */
    step = StepSizeTable[index];

    /* Inverse quantize the ADPCM code into a difference using the quantizer step size */
    if( code & 4 )
        diffq += step;
    if( code & 2 )
        diffq += step >> 1;
    if( code & 1 )
        diffq += step >> 2;

    /* Add the difference to the predicted sample */
    if( predsample > 32767 )
        predsample = 32767;
    else if( predsample < -32767 )
        predsample = -32767;

    /* Find new quantizer step size by adding the old index and a table lookup using the ADPCM code */
    index += IndexTable[code];

    /* Check for overflow of the new quantizer step size index */
    if( index < 0 )
        index = 0;
    if( index > 88 )
index = 88;

/* Save predicted sample and quantizer step size index for next iteration */
state->prevsample = (short)predsample;
state->previndex = index;

/* Return the new speech sample */
return( predsample );
}
Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, FilterLab, KEELOQ, microID, MPLAB, PIC, PICmicro, PICMASTER, PICSTART, PRO MATE, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

dsPIC, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPSIM, MXDEV, PICC, PICDEM, PICDEM.net, rFIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2002, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.
WORLDWIDE SALES AND SERVICE

AMERICAS
Corporate Office
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-7456

Atlanta
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92861
Tel: 949-263-1888 Fax: 949-263-1338

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northarm Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC
Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-67662000 Fax: 86-28-6766599

China - Fuzhou
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusl Road
Fuzhou 350001, China
Tel: 86-591-75350500 Fax: 86-591-7535021

China - Shanghai
Microchip Technology Consulting (Shanghai) Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xing Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai) Co., Ltd.,
Rm. 1315, 13/F, Shenzhen Kerry Centre, Remannun Lu,
Shenzhen 518001, China
Tel: 86-755-2366086 Fax: 86-755-2366086

Hong Kong
Microchip Technology Hong Kong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

India
Microchip Technology India
India Liaison Office
Divyasarre Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaunessy Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-22 Prime Centre
Singapore, 188980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan
Microchip Technology Taiwan
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Denmark
Microchip Technology Nordic ApS
Regerus Business Centre
Lautrup høj 1-3
Ballupert DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trappu
Batiment A - 1er Etage
91900 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899983

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

© 2002 Microchip Technology Inc.