# Add speech encoding/decoding to your design

*Rodger Richey, Microchip Technology Inc, Chandler, AZ*

Adding speech capabilities to a design can sometimes lead to complex algorithms and expensive DSPs or specialized audio chips. However, with the completion of a simplified adaptive differential pulse-code-modulation (ADPCM) algorithm, you can now implement these audio capabilities in low-cost 8-bit µCs, which typically have lower power consumption and cost than their DSP or audio-chip counterparts. A two-chip design is feasible by offloading the encoding and decoding tasks onto the µC as if it were a peripheral.

Since 1991, the Interactive Multimedia Association (IMA) Digital Audio Technical Working Group (DATWG) has been working to define a cross-platform digital-audio exchange format. An inherent problem exists with the exchange of audio data between PC, Mac, and workstation computers. Each computer has its own data types and sampling rates. In May 1992, IMA DATWG published the first revision of the Cross-Platform Digital Audio Interchange recommendation that specifies three uncompressed and one compressed data type at various sample rates. The compressed data type is the Intel (www.intel.com) 4-bit DVI ADPCM algorithm. This algorithm compresses a 16-bit signed audio sample into 4 bits and takes advantage of the high correlation between consecutive samples, which enables the prediction of future samples. Instead of encoding the sample itself, ADPCM encodes the difference between a predicted sample and the actual sample. This method provides more efficient compression with fewer bits per sample and yet preserves the overall quality of the audio signal. Both the encoder routine (**Listing 1**) and the decoder routine (**Listing 2**) are written in C to ease readability.

**LISTING 1—ADPCM ENCODER**

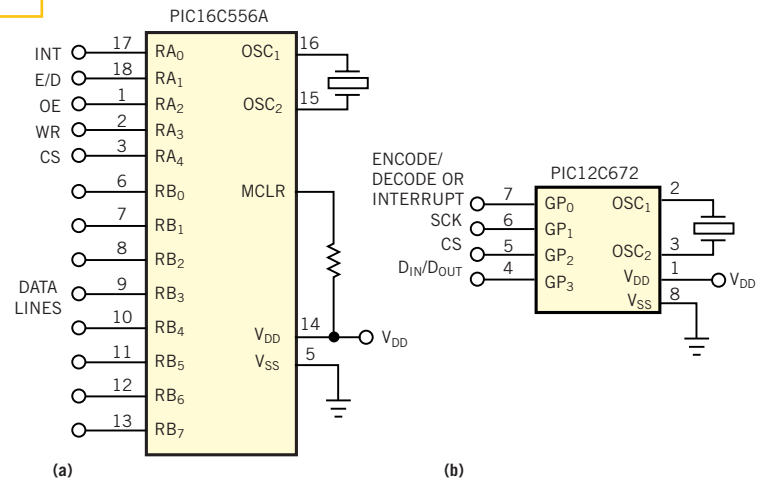
```
/* Table of index changes */
const char IndexTable[16] = {-1,-1,-1,-1,2,4,6,8,-1,-1,-1,-1,2,4,6,8};

/* 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};

signed int diff;                        // Diff. between sample and predicted sample
int step;                               // Quantizer step size
signed int predsample,diffq;            // ADPCM predictor output, Predicted diff.
char index;                             // Index into step size table

char ADPCMEncoder( signed int sample )
{
        char code;                      // ADPCM output value

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

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

        diffq = step >> 3;              // Quantize the diff. into 4-bit ADPCM code
        if( diff >= step )              // using the quantizer step size
        {       code |= 4;              // Inverse quantize the ADPCM code into a
                diff -= step;           // predicted diff. using the quantizer step
                diffq += step;          // size
        }
        step >>= 1;
        if( diff >= step )
        {       code |= 2;
                diff -= step;
                diffq += step;
        }
        step >>= 1;
        if( diff >= step )
        {       code |= 1;
                diffq += step;
        }

        if( code & 8 )                  // Compute new predicted sample by adding
                predsample -= diffq;    // the old predicted sample to new diff.
        else
                predsample += diffq;
        if( predsample > 32767 )        // Check if overflow of the new pred. sample
                predsample = 32767;
        else if( predsample < -32768 )
                predsample = -32768;

        index += IndexTable[code];      // Find new quantizer stepsize
        if( index < 0 )                 // Check if overflow of new quantizer step
                index = 0;
        if( index > 88 )
                index = 88;

        state.prevsample = predsample;  // Save values for next iteration
        state.previndex = index;
        return ( code & 0x0f );         // Return the ADPCM code
}
```

The hardware implementation depends on the type of interface. With a parallel interface, you can use the PIC16C556A (Microchip Technology, www.microchip.com) (**Figure 1a**). A standard parallel interface uses the chip-

select (CS), output-enable (OE), and write-enable (WR) pins on Port A. The 8-bit data interface connects to the 8-bit Port B on the µC. You can use two additional I/O lines for status information, such as encode/decode select, to the µC or an interrupt line to the main controller to indicate when data is ready. CS, which connects to the $RA_4$ pin, can interrupt the PIC16C556A on the start of a transmission.

The second hardware implementation uses a serial interface and an eight-pin µC (**Figure 1b**). The PIC12C672 uses four of the pins for power, ground, and oscillator input and output. Three of the remaining I/O pins are for clock (SCK), data in/out ($D_{IN}/D_{OUT}$), and CS. You can use the other I/O pin to indicate the desired encode/decode operation or as an interrupt to the main controller. CS connects to the external interrupt pin, $GP_2$, to detect the start of a transmission.

Figure 1




**Using the simplified ADPCM algorithm, you can now implement audio capabilities in µCs by offloading the encoding and decoding tasks onto the parallel (a) or serial (b) mC as if it were a peripheral.**

Both µCs have a flexible oscillator structure for use with a crystal, a resonator, or an external clock signal. Both parts of the **figure** show the µC using an external crystal as the clock source. Because these devices are fully static, the main controller can provide the clock source to the µC. By turning the clock source on and off as necessary, the main controller can further decrease overall power consumption. This method also allows control of the speed at which the algorithm runs, which is proportional to the sample rate of the system.

To fully implement the µC as an ADPCM-encoder/decoder peripheral requires firmware to implement the serial or parallel interface, such as **listings 1** and **2**, and a main routine to tie everything together. The main controller is responsible for sampling the incoming audio waveform, storing and retrieving the ADPCM codes from nonvolatile memory, and then playing the resulting samples. The main controller feeds samples or ADPCM codes to the µC and then reads the resulting ADPCM codes or samples from the µC.

The **listings** are available for downloading from at *EDN*'s Web site, www.ednmag.com. At the registered-user area, go into the Software Center to download the file from DI-SIG, #2292. (DI #2292)

## LISTING 2—ADPCM DECODER

```
// This routine also uses the IndexTable and StepSizeTable from Listing 1
signed int ADPCMDecoder( char code )
{
        predsample = state.prevsample;          // Restore previous values
        index = state.previndex;

        step = StepSizeTable[index];            // Find quantizer step size

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

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

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

        index += IndexTable[code];              // Find new quantizer step size

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

        state.prevsample = predsample;          // Save values for next iteration
        state.previndex = index;

        return( predsample );                   // Return new sample

}
```

To Vote For This Design, Circle No. 335