CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

by Chris Coulston, Brian Nypaver, & Jeffrey Rimko

DTMF Decoder

Using some serious math and timing-saving design techniques, you can build a PIC18F452 microcontroller-based device that decodes the tones generated by a telephone keypad. Chris, Brian, and Jeffrey describe a system that can decode a touch-tone in 1 s.

One of the great joys of being an educator is that I'm continually learning—sometimes in unexpected ways. Students in my Embedded Systems class at Penn State Erie are required to define, plan, and implement their own final project. Last year, two of my students, Brian Nypaver and Jeffrey Rimko, set out to build a DTMF decoder. I had taken a Signals and Systems course many years ago, but beyond knowing that a Fourier transform of some sort was required, I had no idea of how Brian and Jeffrey were going to accomplish their goal. With a lot of hard work they completed their DTMF decoder project and taught me something new. In this article, we will share the mathematics and electronics you'll need to confidently incorporate DTMF decoding in your next project.

DTMF

Each button on a telephone keypad generates a unique tone that's the sum of two sinusoids. Hence its name: dual-tone multifrequency (DTMF). The frequency of the first sinusoid is determined by the key's column. The key's row determines the second sinusoid.

	1,209 Hz	1,336 Hz	1,477 Hz
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Table 1—*Each telephone key generates a tone with two frequencies.*

The relationship between the keys and their frequencies is defined in Table 1. For example, pressing the 5 button transmits a dial tone that's the sum of a 1,336-Hz sinusoid and a 770-Hz sinusoid.

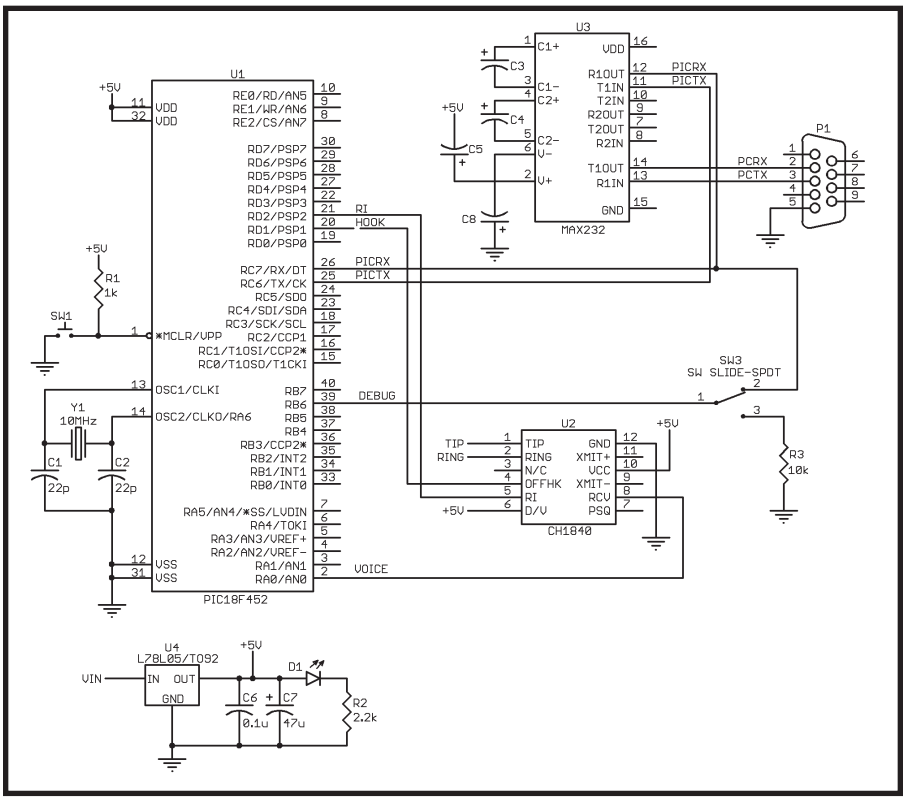

Figure 1—*The Cermetek CH1840 module simplifies the hardware required for our decoding project.*

POTS

The plain old telephone service (POTS) interface consists of four colored wires: a red/green pair (Christmas pair) and a black/orange pair (Halloween pair). Most phone jacks use only the Christmas pair.

With a Christmas pair, you can form a full-duplex connection with the central office (your local switching exchange) to communicate with the outside world. The green (positive) line is referred to as the "tip." The red (negative) line is referred to as the "ring." These terms are based on the physical construction of the plugs used by early phone operators. Check out a stereo jack for a contemporary example.

To prevent leakage currents from damaging the POTS wiring, the tip line is held to a ground potential. The ring line is held at –48 V.

The POTS signals an incoming call by asserting a 20-Hz, 80- to 100-V_{PP} sine wave across the tip and ring. This

voltage is enough to notify the fingers of even the most inattentive hobbits that there is an incoming call. (Use caution when working with phone lines!)

You can take your phone off the hook by drawing 35 mA through the tip and ring lines. This causes the voltage across the tip and ring lines to drop to 10 V. Note that this is enough power to cook the average 0.25-W resistor. After an incoming call is received, the incoming signal is picked up from the voltage drop across the tip and ring lines.

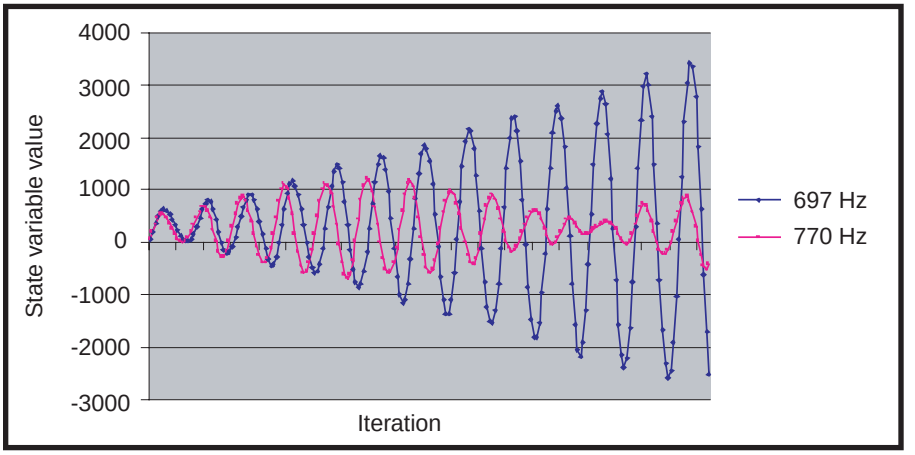

Figure 2—*We graphed the trajectory of the 697- and 770-Hz state variables through 205 iterations of samples from the 1 button.*

HARDWARE

A Cermetek CH1840 data access arrangement module simplified the hardware portion of this project (see Figure 1). Its FCC Part 68 registration can be transferred to your design. This enables you to bypass Part 68 registration and testing.

A single 5-V source powers the CH1840. Information on the tip and ring lines is accessible to your design as the ring indication (RI) and receive (RCV) signals. An incoming phone call is signaled over the POTS as an 80- to 100-V_{PP}, 20-Hz sine wave between the ring and tip lines. The CH1840 transforms this into a 5-V_{PP}, 20-Hz waveform on the RI output.

You can have the CH1840 pick up the phone (placing it off hook) by asserting OFFHK low. The incoming signal from the POTS is accessible as a 2.5-V, centered, 5-V_{PP} signal on the RCV output. Although we didn't use it in this project, outgoing data can be placed on the POTS by asserting a 2.5-V, centered, 5-V_{PP} signal on the transmit (XMIT+) pin. We sent the incoming RCV signal into channel 0 of the PIC18F452 microcontroller's ADC, which was sampling at 11.312 kHz.

The SPDT slide switch puts the PIC18F452 microcontroller in Debugging mode. The debugger is part of an on-chip monitor that enables designs to be flashed over the RS-232 connection.

MATH

Hopefully the word "math" doesn't have you off looking for another article to read. Don't worry. There aren't too many technical details.

In order to determine which key was pressed, you need to determine the frequency composition of the incoming digitized audio waveform. In other words, you need to perform a Fourier transform.

Joseph Fourier (1768–1830) showed that a periodic signal could be represented as the sum of sinusoids—the so-called Fourier series. These sinusoids are characterized by their amplitude, frequency, and phase. The amplitude of a sinusoid in a Fourier series tells you how much its frequency contributes to the formation of the signal. The discrete Fourier transformation (DFT) is a mathematical prescription to convert a sequence of N signal samples taken at a sampling frequency F_S into the magnitude of the sinusoids at a discrete set of frequencies.

Unfortunately, even the fast Fourier transform (FFT), which is the speedy variant of the DFT, is time-consuming for a modest MCU. Part of the reason that FFT algorithms are computationally expensive is that the FFT determines the magnitude of the frequency contributions at N equally spaced frequencies from 0 Hz to F_S.

For example, suppose the sampling frequency is 7.3 kHz. Because the dial tone problem requires discrimination between frequencies that can be within 73 Hz (697 Hz and 770 Hz) of one another, N must be at least 100 samples (7.3 kHz/73 Hz). Hence, an FFT would determine the magnitude of 100 different sinusoids with frequencies 0 Hz, 73 Hz, 146 Hz, and so on up to 7.3 kHz. Clearly, if you're looking for dial-tone frequencies (there are only seven of them), then the FFT produces a lot of information you don't need. Enter Goertzel's algorithm.

You can think of Goertzel's algorithm as a partial DFT. It determines the magnitude of a chosen set of frequencies and ignores all others. In this project, however, we focused on determining the magnitude of the seven frequencies described in Table 1. In order to determine the magnitude of a

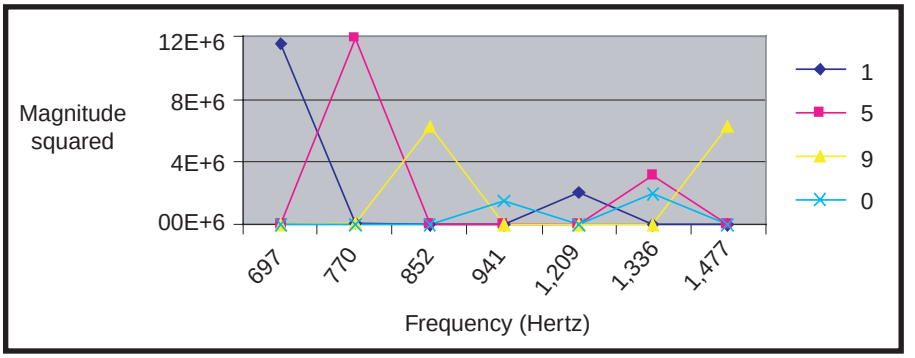

Figure 3—*These frequency components of the key tones are for buttons 1, 5, 9, and 0. Note that only four buttons are shown here.*

target frequency's (F_T) contribution to a signal, use the following constant:

$$a = 2 \times \cos\left(\frac{2\pi F_T}{F_S}\right) \quad [1]$$

If F_T = 697 Hz and F_S = 7.3 kHz, then a = 1.65. For each frequency of interest, Goertzel's algorithm maintains a state variable and its previous two values. The current value of the state variable is called $Q(t)$. The two previous values are called $Q(t-1)$ and $Q(t-2)$. Note that $Q(t)$ initially equals $Q(t-1) = Q(t-2) = 0$. Let $s(t)$ denote the ADC sample value acquired at time t. The current state variable value depends on its previous two values, the constant determined by Equation 1, and the ADC sample:

$$Q(t) = a \times Q(t-1) - 1(t-2) + s(t) \quad [2]$$

There is no formal requirement on the number of iterations to run the Goertzel algorithm. Anecdotal evidence suggests that 205 iterations of the state variable provide reliable tone decoding. This is supported by the trajectory (successive values) of the state variable shown in Figure 2 (p. 81). The graph shows that the magnitude of the target frequency state variable steadily increases while the non-target frequency state variable's magnitude trails off and reaches a minimum at around 200 iterations.

The relationship between the state variable and the squared magnitude of its corresponding frequency is given by:

$$\text{magnitude}^2 = Q(t-1)^2 + Q(t-2)^2 - aQ(t-1) \times Q(t-2) \quad [3]$$

In the algorithm used to decode the dial tones, you need to know if the strongest magnitude surpasses a fixed threshold. Therefore, there is no reason to go through the extra trouble of computing the square root to get the magnitude because we can compare the squared magnitude against a (precomputed) squared threshold.

FIRMWARE

The firmware's main responsibility is running the Goertzel algorithm on the seven frequencies of interest. There are two alternative implementations to the Goertzel algorithm. You can incrementally compute the trajectory of the seven state variables after each new ADC value is acquired. Or, you can acquire all the samples and then compute the complete trajectory of each state variable.

The first alternative spreads the computation of the state variable through the unavoidable delay of acquiring the samples. However, this requires seven multiplications and 14 additions to be performed at each interrupt. Unfortunately, the PIC18F452 couldn't perform this heavy lifting in the time between successive ADC samples. Therefore, our DTMF decoding implementation acquired all of its samples and then computed the state variable trajectory.

Even if you have the advantage of using a C compiler to write your embedded firmware, it's usually a good idea to avoid floating-point computation in favor of fixed-point math. The advantage of fixed-point math is that you can use integer multiplication hardware with the microcontroller, or you can use the multiplication algorithms implemented by the compiler writers.

We used the fixed-point format because we needed an accurate resolution for the coefficients defined by Equation 1. It also provided for the large magnitudes defined by Equation 3. Using different formats accommodates both needs.

A pragmatic approach to any fixed-point representation involves fitting it in a word size with a length that's a power of two. Because the ADC's sampling rate was set to 11.312 kHz, the coefficients in Equation 1 were between 1.852 and 1.365. We used a fixed-point representation to accomodate this range. The most significant 4 bits contained the whole number portion of the coefficient. The lower 12 bits contained the fractional portions. Table 1 lists the values of a for each target frequency and the associate 16-bit fixed-point value.

In order to determine the number of bits required for the state variables, we ran simulations to determine the range of values under a variety of circumstances. Figure 2 shows the trajectory of the 697- and 770-Hz state variables over 205 iterations of the 1 button's signal input. The maximum $Q(t)$ observed after 205 iterations was always safely below 2^{15}—the maximum allowable magnitude in a 16-bit signed number. In order to save memory space, the fractional remainder of the state variable computation was truncated to 16 bits.

Because we're dealing with the magnitude squared, a rough estimate of a maximum magnitude is $4{,}000^2$, a number requiring roughly 24 bits. Hence, the magnitudes (squared) are put into a 32-bit format. It clearly makes sense to abandon the fractional portion. After the numbering format decisions have been decided, it's time to examine the firmware's structure.

The main routine initializes the PIC18F452 and then waits for an incoming phone call. The PIC18F452 then picks up the phone and listens for tones by converting the incoming audio signal into 8-bit samples using the ADC subsystem. After collecting 205 samples at 11.318 kHz, the PIC18F452 computes the magnitude of the frequency components using the code segment shown in Listing 1.

Notice that the coefficients defined in Listing 1 differ from those specified in Table 1. This discrepancy will be

Listing 1—*Use this firmware to update the state variables according to Equation 2.*

```
int8 SAMPLE[205];
int16 COEFF[7]={ 0x1D83, 0x1D1B, 0x1CB0, 0x1BC0, 0x1900, 0x17C0, 0x15C0};

void GOERTZEL(int8 sample[205]) {
   for (step=0; step<N; step++) {
      for(i=0; i<7; i++) {
         if (Q0[i]&0x8000)
            Q0[i] =~((COEFF[i]*~Q1[i])>>12) - Q2[i] + SAMPLE[step];
         else
            Q0[i] = ((COEFF[i]* Q1[i])>>12) - Q2[i] + SAMPLE[step];
         Q2[i] = Q1[i];
         Q1[i] = Q0[i];
}  }   }  } // end GOERTZEL, for, for
```

Frequency	*a*	16-bit Representation
697 Hz	1.8522	0x1DA3
770 Hz	1.8202	0x1D20
852 Hz	1.7806	0x1C7D
941 Hz	1.7335	0x1BBC
1,209 Hz	1.5666	0x190E
1,336 Hz	1.4751	0x179A
1,477 Hz	1.3651	0x15D7

Table 2—*These constants are used in Equation 2. Also included is the 16-bit representation of the fraction.*

explained in the next section. The if/then structure in the computation of Q0 is necessary because the compiler used for this project assumes unsigned numbers when performing integer multiplication. Hence, a negative state variable is complemented, multiplied, and then complemented back (complementing was used instead of negation for computational efficiency).

The 12-bit shift right in the Q0 computation is required to normalize the product after multiplication by the fixed-point coefficient. Even though the computation shown in Listing 1 involves only integer multiplications, the entire subroutine call requires about 0.25 s.

In order to determine which key was pressed, the state variables are converted into magnitudes (squared) using Equation 3. The maximum row frequency magnitude and column frequency magnitude are then compared to a preset threshold to determine which key, if any, was pressed.

RESULTS

When we had the hardware and firmware working properly, it was time to decode dial tones. Unfortunately, the algorithm's performance was spotty at best. The problem was that the magnitude of the 852-Hz signals was always high regardless of the input. The 697- and 770-Hz signals seemed to take on random values. We narrowed down the possible causes to the coefficients defined by Equation 1 and shown in Table 2.

The coefficients were systematically varied for each frequency, using the values defined in Table 2 as a starting point. In general, the best coefficients generated a small magnitude to DC inputs, medium responses to non-related frequencies, and strong responses to their own frequencies. The results are given in Table 3, where we compare the theoretical and experimentally derived coefficients for each frequency. There is no systematic variation in the derived coefficients. This rules out any systematic error in the system.

The performance of the algorithm is summarized in Figure 3 (p. 81), which is a graph of magnitude squared versus frequency for four buttons on the keypad. Two characteristics are critical for the successful operation of the decoding algorithm: the magnitude of the target and non-target frequencies. For example, when the 5 button is pressed, the algorithm has a strong response in the 770- and 1336-Hz bands—exactly as Table 1 predicts.

GO DECODE

I hope this project has taken some of the mystery out of the frequency domain. Perhaps you'll even incorporate DTMF decoding into your next project.

The added expense of the CH1840 module is more than offset by its reliable operation and out-of-the-box FCC compliance. It's a real timesaver.

While the frequency domain may always remain mysterious to bit pushers, we can all agree that the computations required by Goertzel's algorithm an be computed by almost any microcontroller.

Chris Coulston holds a Ph.D. in computer science and engineering from the Penn State University. He is currently an associate professor and the program chair of the Electrical and Computer Engineering Department at Penn State Erie, The Behrend College. You may contact him at coulston@psu.edu.

Brian Nypaver is a senior majoring in computer engineering at Penn State Erie, The Behrend College. He is expected to graduate in May 2006. Brian worked as an intern for two summers at Consol Energy. He is currently working with Jeffrey Rimko on a senior design project that will control a computer mouse using brainwaves. You may contact Brian at bmn127@psu.edu.

Jeffrey Rimko is a senior majoring in computer engineering at Penn State Erie, The Behrend College. He is expected to graduate in May 2006. He is currently working with Brian Nypaver on a senior design project that will control a computer mouse using brainwaves. You may contact Jeffrey at jrr232@psu.edu.

Implies frequency	697 Hz	770 Hz	852 Hz	941 Hz	1,209 Hz	1,336 Hz	1,477 Hz
Theoretical	0x1DA3	0x1D20	0x1C7D	0x1BBC	0x190E	0x179A	0x15D7
Experimental	0x1D83	0x1D1B	0x1CB0	0x1BC0	0x1900	0x17C0	0x15C0

Table 3—*We compared the theoretical and experimental constants used in our Goertzel implementation.*

PROJECT FILES

To download the code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2006/187.

RESOURCES

K. Banks, "The Goertzel Algorithm," Embedded Systems Programming, August 2002, www.embedded.com/story/OEG20020819S0057.

DSPRelated.com, "DTMF Detection with Goertzel," 2005, www.dsprelated.com/showmessage/38661/1.php.

Dual-tone multi-frequency, en.wikipedia.org/wiki/DTMF.

J. Macassey, "Understanding Telephones," *Ham Radio Magazine*, September 1985, massis.lcs.mit.edu/archives/technical/how.phones.work.

SOURCES

CH1840 DAA module
Cermetek Microelectronics, Inc.
www.cermetek.com/data.htm

PIC18F452 Microcontroller
Microchip Technology, Inc.
www.microchip.com