Dual-Tone Multi-Frequency Coding 14

14.1 INTRODUCTION

DTMF is the generic name for pushbutton telephone signaling equivalent to the Bell System's TouchTone®. Dual-Tone Multi-Frequency (DTMF) signaling is quickly replacing dial-pulse signaling in telephone networks worldwide. In addition to telephone call signaling, DTMF is becoming popular in interactive control applications, such as telephone banking or electronic mail systems, in which the user can select options from a menu by sending DTMF signals from a telephone.

To generate (encode) a DTMF signal, the ADSP-2100 adds together two sinusoids, each created by software. For DTMF decoding, the ADSP-2100 looks for the presence of two sinusoids in the frequency domain using modified Goertzel algorithms. This chapter shows how to generate and decode DTMF signals in both single channel and multi-channel environments. Realizable hardware is briefly mentioned.

DTMF signals are interfaced to the analog world via codec (coder/decoder) chips or linear analog-to-digital (A/D) converters and digital-to-analog (D/A) converters. Codec chips contain all the necessary A/D, D/A, sampling and filtering circuitry for a bidirectional analog/digital interface. These codecs with on-chip filtering are sometimes called codec/filter combo chips, or combo chips for short. They are referred to as codecs in this chapter.

The codec channel used in this example is bandlimited to pass only frequencies between 200Hz and 3400Hz. The codec also incorporates companding (audio compressing/expanding) circuitry for either of the two companding standards (A-law and μ-255 law). These two standards are explained in Chapter 11, *Pulse Code Modulation*. Companding is the process of logarithmically compressing a signal at the source and expanding it at the destination to maintain a high end-to-end dynamic range while reducing the dynamic range requirement within the communication channel.

In the example of DTMF signal generation shown in this chapter, the ADSP-2100 reads DTMF digits stored in data memory in a relocatable look-up list. Alternatively, a DTMF keypad could be used for digit entry. In either case, the resultant DTMF tones are generated mathematically and added together. The values are logarithmically compressed and passed to the codec chip for conversion to analog signals. Multi-channel DTMF signal generation is performed by simply time-multiplexing the processor among the channels.

On the receiving end, the ADSP-2100 reads the logarithmically compressed, digital data from the codec's 8-bit parallel data bus, logarithmically expands it to its 16-bit linear format, performs a Goertzel algorithm — a fast DFT (discrete Fourier transform) calculation — for each tone to detect, then passes the results through several tests to verify whether a valid DTMF digit was received. The result is coded and written to a memory-mapped I/O port. Multi-channel DTMF decoding is also performed by time-multiplexing the channels.

14.2 ADVANTAGES OF DIGITAL IMPLEMENTATION

Several chips are available which employ analog circuitry to generate and decode DTMF signals for a single channel. This function can be digitally implemented using the ADSP-2100. The advantages of a digital system include better accuracy, precision, stability, versatility, and reprogrammability as well as lower chip count, and thereby reduced board-space requirements.

Table 14.1 compares the tone accuracy expected of a DTMF tone dialer chip with the accuracy of tones generated by the ADSP-2100. Note that a tone dialer chip is an application-specific device with preprogrammed frequencies for use on a single channel, whereas the ADSP-2100 is a general purpose microprocessor which can be programmed to generate any frequency for many separate channels. When using the ADSP-2100, the frequency values can be specified to within 16 bits of resolution, implying an accuracy of 0.003%.

By simply changing the frequency values, the ADSP-2100 tone generator can be fine-tuned or reprogrammed for other tone standards, such as CCITT 2-of-6 Multi-Frequency (MF), call progress tones, US Air Force 412L, and US Army TA-341/PT. Since the numbers stored in data memory do not change in value over time or temperature, the precision and stability of a digital solution surpasses any analog equivalent. DTMF encoding and decoding can be written as a subfunction of a larger program, eliminating the need for separate components and specialized interface circuitry.

DTMF Standard Frequency	*Typical frequency from hybrid tone-dialer chip using 3.579545MHz crystal*	% deviation	*Frequency from ADSP-2100 program*	% deviation
697.0Hz	699.1Hz	+0.31%	697.0Hz	±0.003%
770.0Hz	766.2Hz	–0.49%	770.0Hz	±0.003%
852.0Hz	847.4Hz	–0.54%	852.0Hz	±0.003%
941.0Hz	948.0Hz	+0.74%	941.0Hz	±0.003%
1209.0Hz	1215.9Hz	+0.57%	1209.0Hz	±0.003%
1336.0Hz	1331.7Hz	–0.32%	1336.0Hz	±0.003%
1477.0Hz	1471.9Hz	–0.35%	1477.0Hz	±0.003%
1633.0Hz	1645.0Hz	+0.73%	1633.0Hz	±0.003%

Table 14.1 Precision of Analog and Digital Tone Generation

14.3 DTMF STANDARDS

The DTMF tone signaling standard is also known as TouchTone or MFPB (Multi-Frequency, Push Button). TouchTone was developed by Bell Labs for use by AT&T in the American telephone network as an in-band signaling system to supersede the dial-pulse signaling standard. Each administration has defined its own DTMF specifications. They are all very similar to the CCITT standard, varying by small amounts in the guardbands (tolerances) allowed in frequency, power, twist (power difference between the two tones) and talk-off (speech immunity). The CCITT standard appears as Recommendations Q.23 and Q.24 in Section 4.3 of the CCITT Red Book, Volume VI, Fascicle VI.1. Other standards (AT&T, CEPT, etc.) are listed in *References*, at the end of this chapter.

Two tones are used to generate a DTMF digit. One tone is chosen out of four row tones, and the other is chosen out of four column tones. Two of eight tones can be combined so as to generate sixteen different DTMF digits. Of the sixteen keys shown in Figure 14.1, on the next page, twelve are the familiar keys of a TouchTone keypad, and four (column 4) are reserved for future uses.

A 90-minute audio-cassette tape to test DTMF decoders is available from Mitel Semiconductor (part number CM7291). There also exists a standard describing requirements for systems which test DTMF systems. This standard is available from the IEEE as ANSI/IEEE Std. 752-1986.

14 Dual-Tone Multi-Frequency

		Column 1 1209Hz	Column 2 1336Hz	Column 3 1477Hz	Column 4 1633Hz
Row 1	697Hz	1	2	3	A
Row 2	770Hz	4	5	6	B
Row 3	852Hz	7	8	9	C
Row 4	941Hz	*	0	#	D

DTMF digit = Row Tone + Column Tone

Figure 14.1 DTMF Digits

14.4 DTMF DIGIT GENERATION PROGRAM

Generation of DTMF digits is relatively straightforward. Digital samples of two sine waves are generated (mathematically or by look-up tables), scaled and then added together. The sum is logarithmically compressed and sent to the codec for conversion into the analog domain.

A sine look-up table is not used in this example because the sine can be computed quickly without using the large amount of data memory a look-up table would require. The sine computation routine is efficient, using only five data memory locations and executing in 25 instruction cycles. The routine used to compute the sine is from Chapter 4, *Function Approximation*. This routine evaluates the sine function to 16 significant bits using a fifth-order Taylor polynomial expansion. The sine computation routine is called from the tone generation program as an external routine; refer to that chapter for details on the sine computation routine.

To build a sine wave, the tone generation program utilizes two values. One, called *sum*, keeps track of where the current sample is along the time axis, and the other, called *advance*, increments that value for the next sample. Since the DTMF tone generation program generates two tones, there are two different *sum* values and two different frequency values (stored in the variables *hertz*). The value *sum* is stored in data memory in a variable called *sum*. *Sum* is modified every time a new sample is calculated. The value *advance* is calculated from a data memory variable called *hertz*. *Hertz* is a constant for a given tone frequency. The ADSP-2100 calculates the *advance* value from a stored *hertz* variable instead of storing *advance* as the data memory variable because this allows you to read the frequency being generated in Hz directly from the data memory display (in decimal mode) of the ADSP-2100 Simulator or Emulator.

The *sum* values are dm(sin1) and dm(sin2) in Listing 14.1 (see *Program Listings* at the end of this chapter). The *advance* values are derived from the variables dm(hertz1) and dm(hertz2) in Listing 14.1. For readable source code and easy debugging, Listing 14.1 uses two data memory variables dm(sin1) and dm(sin2) to store the value returned from each call to the sine subroutine rather than storing the value in a register. These variables are then added together, resulting in a DTMF output signal value.

The sampling frequency of telephone systems is 8kHz. Therefore, the ADSP-2100 must output samples every 125µs. An 8kHz TTL square wave is applied to an interrupt (IRQ3 in this case) pin of the ADSP-2100. The ADSP-2100 is initialized for edge-sensitive interrupts with interrupt-nesting mode disabled (see Listing 14.1, two lines immediately preceding the label *wait_int* near the top of executable code). The sampling frequency, in conjunction with the *advance* value, determines the frequency of the sine wave generated.

Circular movement around a unit circle is analogous to linear motion along the time axis of a sine wave, one revolution of the circle corresponding to one period of the wave. The range of inputs to the sine function approximation subroutine is $-\pi$ to 0 to $+\pi$ radians. This range maps to the 16-bit hexadecimal numbers H#8000 to H#FFFF and H#0000 to H#7FFF (see Figure 14.2 on the next page and Table 14.2 following it). All of the 16-bit numbers are equally spaced around the unit circle, dividing it into 65536 parts. The *advance* value is added to the *sum* value during each interrupt. To generate a 4kHz sine wave, the *advance* value would have to be 32768, equivalent to π radians, or a jump halfway around the unit circle. Because Nyquist theory dictates that 4kHz is the highest frequency that can be represented in an 8kHz sampling-frequency system, this is the maximum *advance* value that can be used. For a sine wave frequency of less than 4kHz, the *advance* value would be proportionally less (see Figure 14.3).

$$\textit{advance} \quad \text{value} = 65536\left(\frac{\text{tone frequency desired}}{\text{sampling frequency}}\right)$$

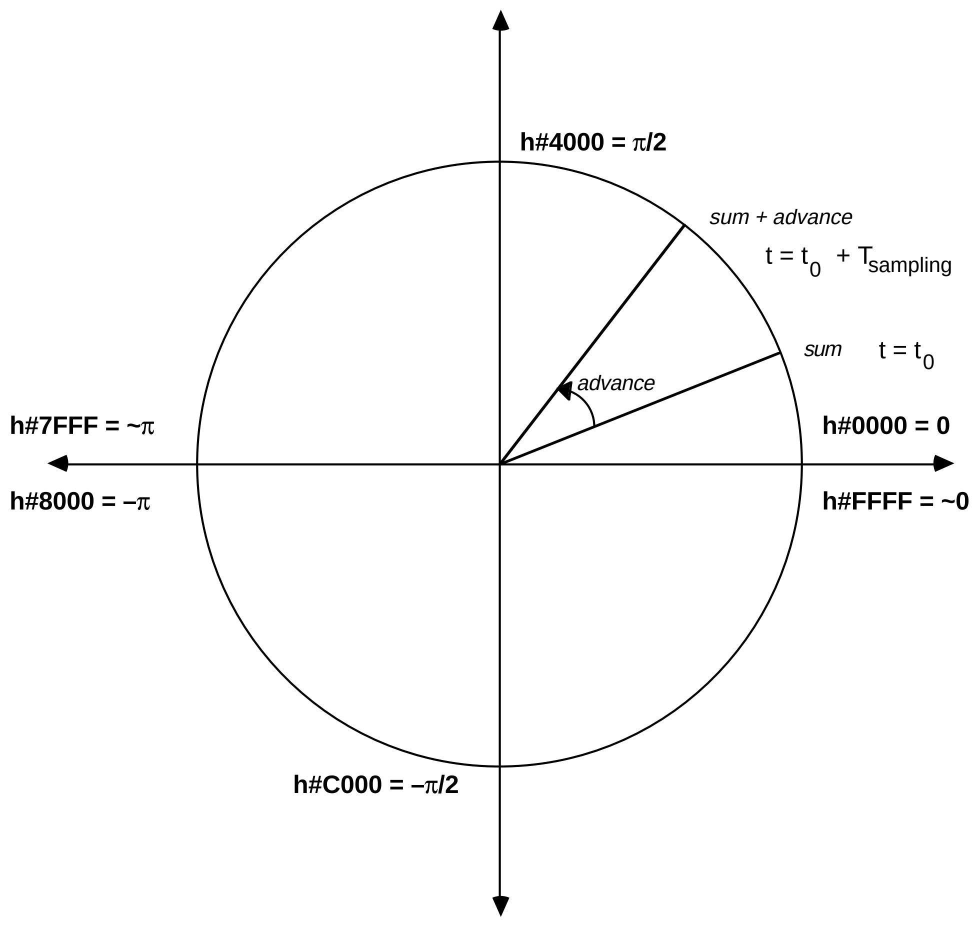


Figure 14.2 Sine Routine Input Angle Mapping

Input to Sine Approximation Routine	*Equivalent Input Angle (radians/degrees)*	
H#0000	0	0
H#2000	π/4	45
H#4000	π/2	90
H#6000	3π/4	135
H#7FFF	~π	~180
H#8000	–π	–180
H#A000	–3π/4	–135
H#C000	–π/2	–90
H#E000	–π/4	–45
H#FFFF	~0	~0

Table 14.2 Sine Routine Input Angle Mapping

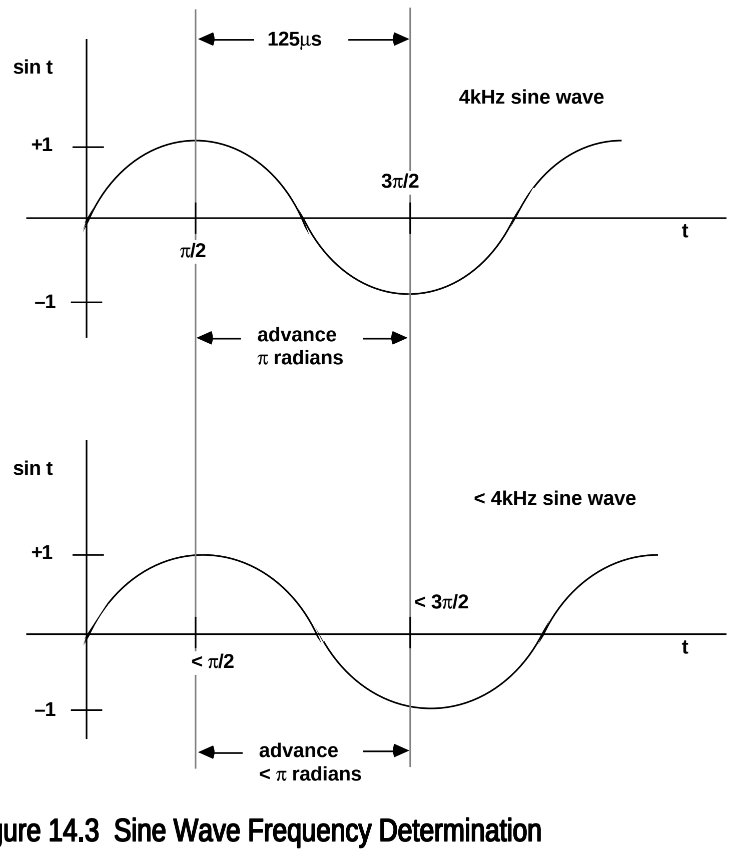


Figure 14.3 Sine Wave Frequency Determination

For examples of some tones and their *advance* values, see Table 14.3. Since telephone applications require an 8kHz sampling frequency, the formula above can be reduced to:

advance required = 8.192 (tone frequency desired in Hz)

Advance Value	*Tone Frequency (Hz)*
H#0000	0
H#2000	$(1/8)f_{sampling}$
H#4000	$(1/4)f_{sampling}$
H#6000	$(3/8)f_{sampling}$
H#7FFF	$\sim(1/2)f_{sampling}$

Table 14.3 Some Advance Values and Frequencies

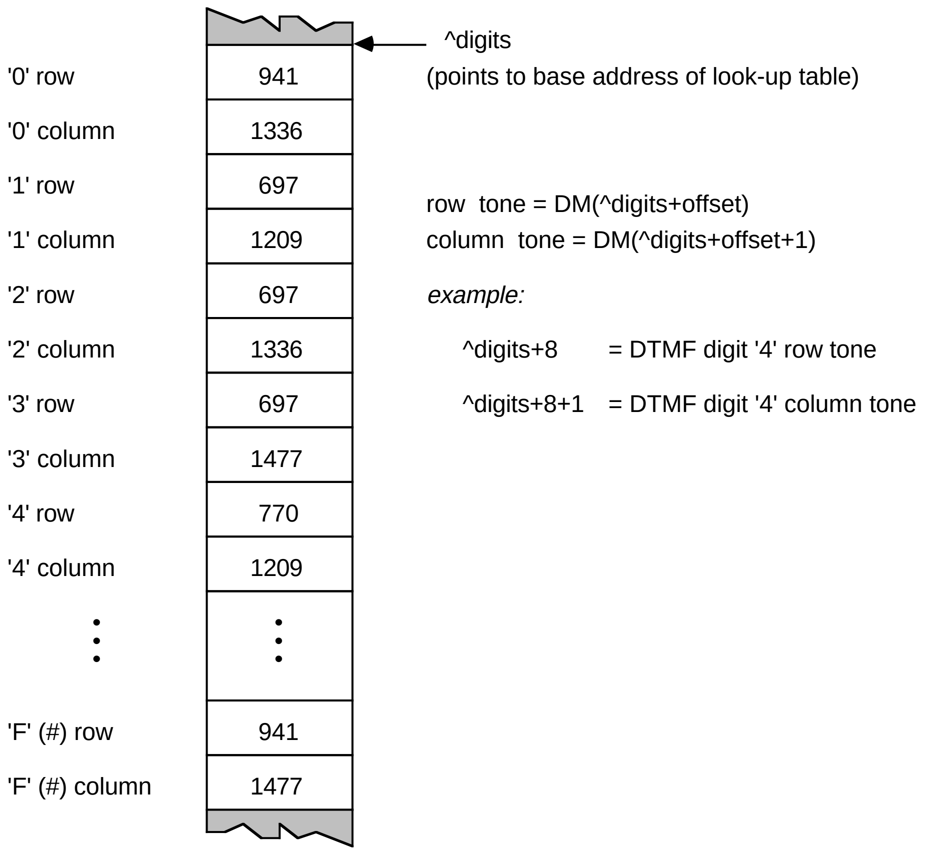
The program shown in Listing 14.1 reads the frequency out of data memory at dm(hertz1) and dm(hertz2). The multiplication by 8.192 is implemented as a multiplication by 8, by 0.512, and then by 2. This approach ensures optimal precision. The multiplications by 8 and by 2 are done in the shifter, and the multiplication by 0.512 is done in the multiplier. The multiplications by 8 and by 2 do not cause any loss of precision. For the multiplication by 0.512, the value 0.512 is represented in 1.15 format, i.e., the full 15 bits of fractional precision. A multiplication by 8.192, which must be represented in at least 5.11 format, would leave only 11 bits of fractional precision. Although not explicitly shown here, this proves to be too little precision for high frequency tones.

After the *advance* value has been added to the *sum* value, the result is written back to the *sum* location in data memory, overwriting the past contents. The result is also passed in register AX0 to the sine function approximation subroutine. That subroutine calculates the sine in 25 cycles and returns the result in the AR register. The sine result is then scaled by downshifting (right arithmetic shift) by the amount specified in dm(scale). This scaling is to avoid overflow when adding the two sine values together later on. The *scale* amount is stored as a variable in data memory at dm(scale) so you can adjust the DTMF amplitude. The scaled result is stored in data memory in either the dm(sin1) or dm(sin2) locations, depending which sine is being evaluated.

When both sines have been calculated, the scaled sine values are recalled out of data memory and added together. That 16-bit, linear result is then passed via the AR register to the µ-255 law, logarithmic compression subroutine (called *u_compress* in Listing 14.1; this routine is listed in Chapter 11, *Pulse Code Modulation*), and the compressed result is finally written to the memory-mapped codec chip.

14.4.1 Digit Entry

There are two methods for entering DTMF digits into the ADSP-2100 program for conversion into DTMF signals. One method is to memory-map a keypad so that when you press a key, the resultant DTMF digit is generated. The other method is to have the ADSP-2100 read a data memory location which contains the DTMF digit in the four LSBs of the 16-bit number (see Figure 14.4). The program shown in Listing 14.1 uses the latter of the two methods. In either case, the row and column frequencies are determined by a look-up table. The keypad method is described in theory, then the data memory method is described, referring to Listing 14.1.



^digits
(points to base address of look-up table)
'0' row 941
'0' column 1336
'1' row 697
'1' column 1209
'2' row 697
'2' column 1336
'3' row 697
'3' column 1477
'4' row 770
'4' column 1209
'F' (#) row 941
'F' (#) column 1477
row tone = DM(^digits+offset)
column tone = DM(^digits+offset+1)
example:
^digits+8 = DTMF digit '4' row tone
^digits+8+1 = DTMF digit '4' column tone

Figure 14.4 Tone Look-Up Table

14.4.1.1 Key Pad Entry

For DTMF digit entry using the keypad, a 74C922 16-key encoder chip is used in conjunction with a 16-key SPST switch matrix and address decoding circuitry. An example of this circuit is shown in Figure 14.5.

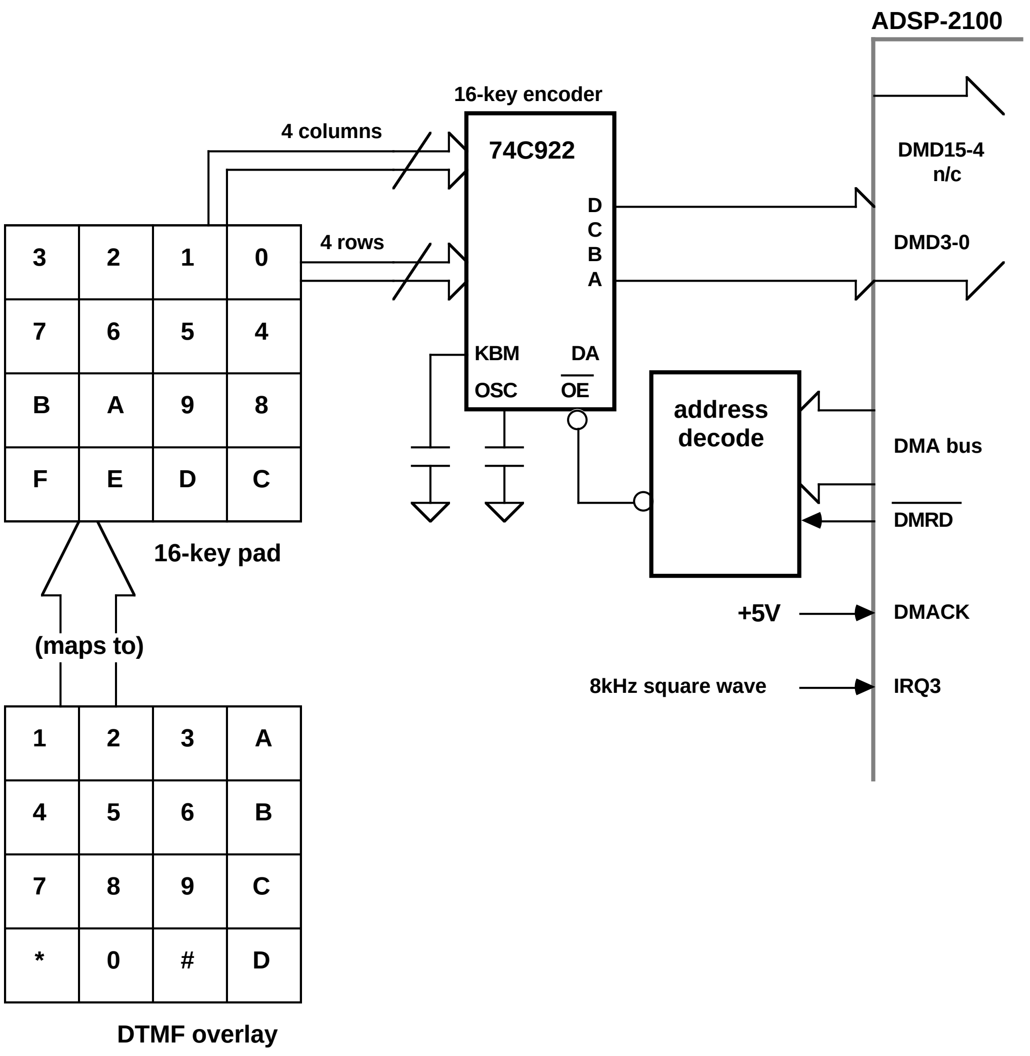


Figure 14.5 Keypad Entry Circuit

The CMOS key encoder provides all the necessary logic to fully encode an array of SPST switches. The keyboard scan is implemented by an external capacitor. An internal debounce circuit also needs a external capacitor as shown in Figure 14.5. The Data Available (DA) output goes high when a valid keyboard entry has been made. The Data Available output returns low when the entered key is released, even if another key is pressed. The Data Available will return high to indicate acceptance of the new key after a normal debounce period. To interrupt the ADSP-2100 when a key has been pressed, invert the Data Available signal and tie it directly to one of the four independent hardware interrupt pins. An internal register in the keypad encoder chip remembers the last key pressed even after the key is released.

Because a DTMF keypad maps to a hexadecimal keypad as shown in Figure 14.5 and Table 14.4, a program using the keypad for digit entry would use a row and column tone look-up list which is similar to that used in Listing 14.1, but its contents would be slightly different. The operation of the tone look-up list is the same as in Listing 14.1, although the values stored in data memory within the look-up list reflect the remapping of the keypad.

DTMF Key	*74C922 Output*	*DTMF Key*	*74C922 Output*
0	xxxE	8	xxxA
1	xxx3	9	xxx9
2	xxx2	A	xxx0
3	xxx1	B	xxx4
4	xxx7	C	xxx8
5	xxx6	D	xxxC
6	xxx5	*	xxxF
7	xxxB	#	xxxD

Table 14.4 DTMF to Keypad Encoder Conversion

14.4.1.2 Data Memory List

For DTMF digit entry by reading the digit in data memory (as in Listing 14.1), a row and column tone look-up list is implemented. The DTMF digits are stored in the four LSBs of the 16-bit data word. All DTMF digits are mapped to their hexadecimal numerical equivalent. The * digit is assigned to the hexadecimal number H#E, and the # digit is assigned to the hexadecimal number H#F. Table 14.5 on the next page shows this mapping. The DTMF digits are stored in such a way that you can see the DTMF sequence being dialed directly out of data memory using the ADSP-2100 Simulator or Emulator in the hexadecimal data memory display mode.

When a DTMF digit is read from data memory, the twelve upper MSBs are first masked out. Then the numerical value of the DTMF digit (0, 1,..., 15 decimal) is multiplied by 2 (yielding 0, 2,..., 30 decimal) because each entry within the look-up table is two 16-bit words long. One word holds the row frequency, and the other word holds the column frequency. This offset value is then added to the base address of the beginning of the look-up table. The resultant address is used to read the row frequency and postincremented by one. The incremented address is used to read the column frequency.

DTMF Digit	*Base Address Offset Value*
0	0
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	16
9	18
A	20
B	22
C	24
D	26
* (E)	28
# (F)	30

Table 14.5 Look-Up Table Offset Values

For example, referring to Listing 14.1 at the label *nextdigit* and Figure 14.4, the DTMF digit is read out of data memory and stored in register AX0 by the instruction AX0=DM(I0,M0). For this example, assume the value in AX0 is H#0004. Control flow is passed to the instruction labeled *newdigit*. The twelve MSBs are set to zero, and the result is multiplied by 2 in the barrel shifter yielding H#0008. AY0 is set to the base address (^*digits*) of the tone look-up table. The base address is added to the offset and placed in the I1 register. I1 now holds the value ^*digits*+8, M0 was previously set to 1, and L1 to 0. The instruction AX0=DM(I1,M0) reads the row frequency (770Hz) from the look-up table and stores it in the variable *hertz1* used by the sinewave generation code. I1 is automatically postmodified by M0 (1), and the next instruction AX0=DM(I1,M0) reads the column frequency (1209Hz) and stores that in the variable *hertz2* used by the sinewave generation code. When the sinewave generation code is executed (at the label *maketones*), a signal of 770Hz + 1209Hz (DTMF digit 4) is generated.

14.4.2 Dialing Demonstration

The program listed in Listing 14.1 is a DTMF dialing demonstration program. DTMF digits are read sequentially out of a linear buffer. The variable *sign_dura_ms* (signal duration in milliseconds) sets the length of time that a DTMF digit is generated. The variable *interdigit_ms* (interdigit time in milliseconds) sets the length of the silent period following a DTMF digit. When a new digit is started, the variable *time_on* is set to 8 x *sign_dura_ms* and the variable *time_off* is set to 8 x *interdigit_ms*. *Time_on* and *time_off* are counters which the program decrements during interrupts to count the passing of time. Because the ADSP-2100 is interrupted at an 8kHz rate, the amount of time between interrupts is 1/8 millisecond (125µs). By taking the time in milliseconds and multiplying that numerical value by 8, the number of interrupts to count results.

The DTMF digit dialing list stored in data memory starting at location ^*dial_list* can contain values other than DTMF digits. It can also contain control words. Refer to the comment immediately following the variable declarations at the top of the source code in Listing 14.1. If a value is encountered in the dialing list which has any bits set in the four MSBs, the dialing stops. This value is used as a delimiter to terminate the dialing list. If a value is encountered in which the four bits 15-12 are zeros, but any of the bits 11-8 are set, the dialing restarts at the top of the dialing list. If a value is encountered in which all eight bits 15-8 are zeros, but any of the bits 7-4 are set, then a quiet space of length *sign_dura_ms* plus *interdigit_ms* is generated. Finally, if all twelve bits 15-4 are zeros, then the four LSBs represent a valid DTMF digit to generate.

A software state machine has been implemented in the program of Listing 14.1. It is partly controlled by IRQ2, which in this example is wired to a debounced switch. The state machine has three states. Pushing the IRQ2 pushbutton moves the state machine into the next state. The current state of the machine is stored in data memory variable *state*. Figure 14.6, on the next page, shows the demonstration program's state machine. The program starts in state 0. In this state, no digits are generated. Pushing the IRQ2 pushbutton moves the machine into state 1, in which a continuous dial tone (350Hz + 400Hz) is generated. The state machine moves to state 2 when the IRQ2 pushbutton is pushed again. In state 2, the DTMF dialing list is sequentially read and DTMF digits generated. The state machine stays in state 2 until IRQ2 is pushed again or a "stop" control word is read out of the dialing list, in which case the machine jumps back to state 0.

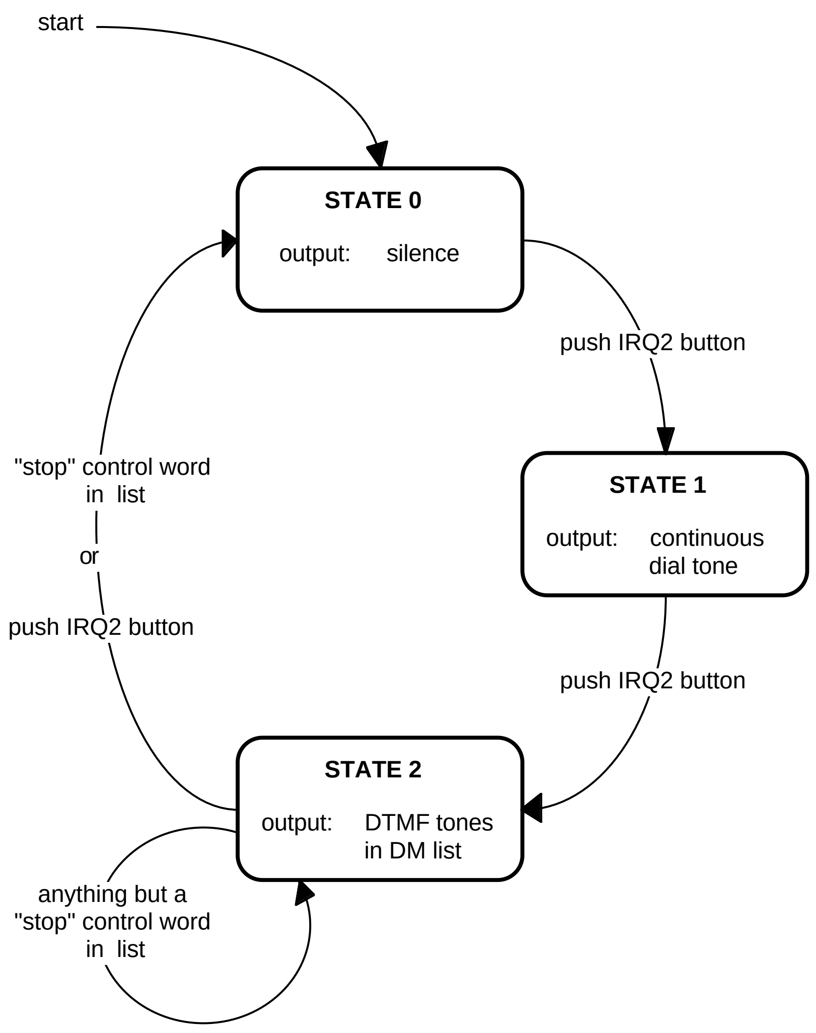

start
STATE 0
output: silence
push IRQ2 button
STATE 1
output: continuous dial tone
push IRQ2 button
STATE 2
output: DTMF tones in DM list
"stop" control word in list
or
push IRQ2 button
anything but a "stop" control word in list

Figure 14.6 Dialing Demonstration Program State Machine

A flowchart of the operation of this demonstration program is shown in Figure 14.7. The data memory variables *row0*, ..., *row3*, *col0*, ..., *col3* in Listing 14.1 are initialized with their appropriate frequencies. These variables are not used by the program at all, but are included as a handy reference so you can look up the frequencies using the data memory display of the ADSP-2100 Simulator or Emulator without having to refer back to any literature.

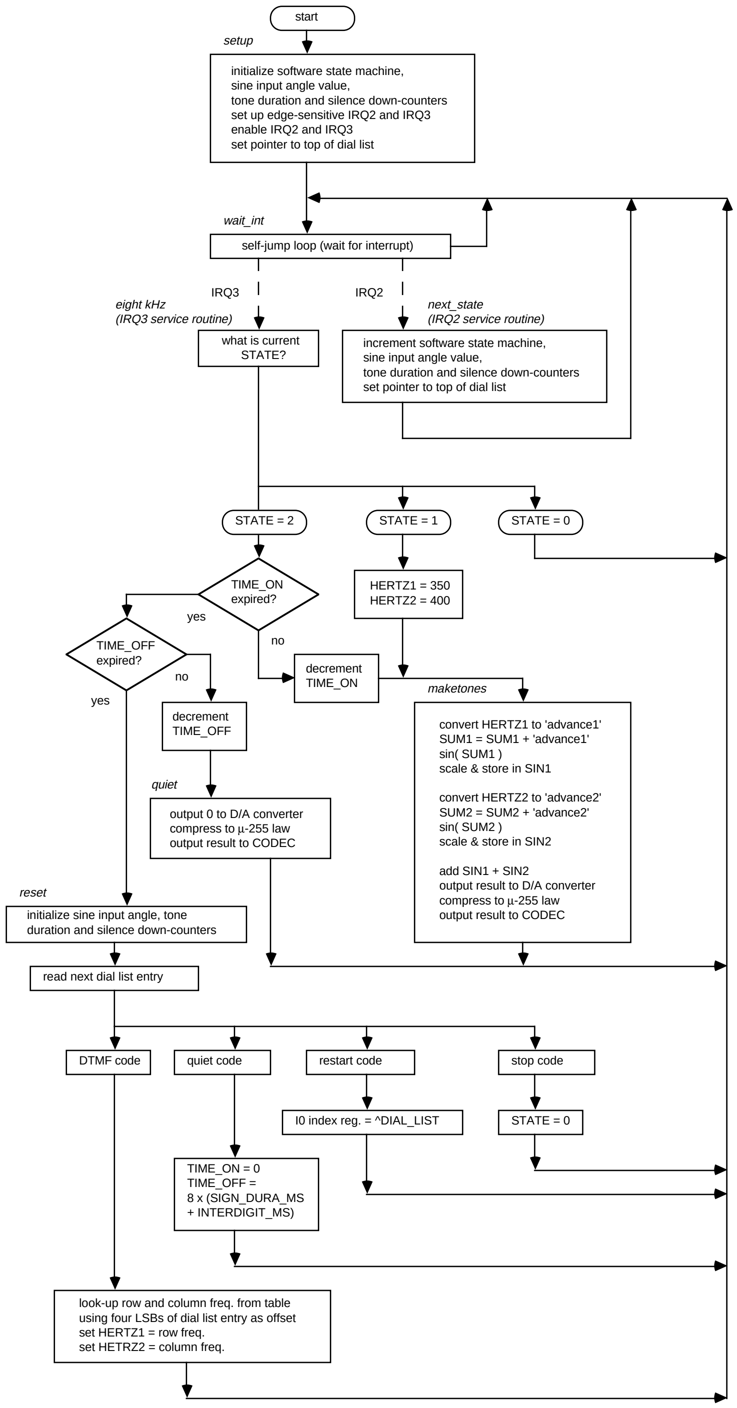


Figure 14.7 Tone Generation Block Diagram

14.4.3 Multi-Channel Generation

A single ADSP-2100 can generate simultaneous DTMF signals for many channels in real time. The multi-channel program uses the same method as the single channel version, but each channel has its own scratchpad variables and memory-mapped I/O ports. The channels are computed sequentially in time every time an interrupt occurs. See Figure 14.8 and

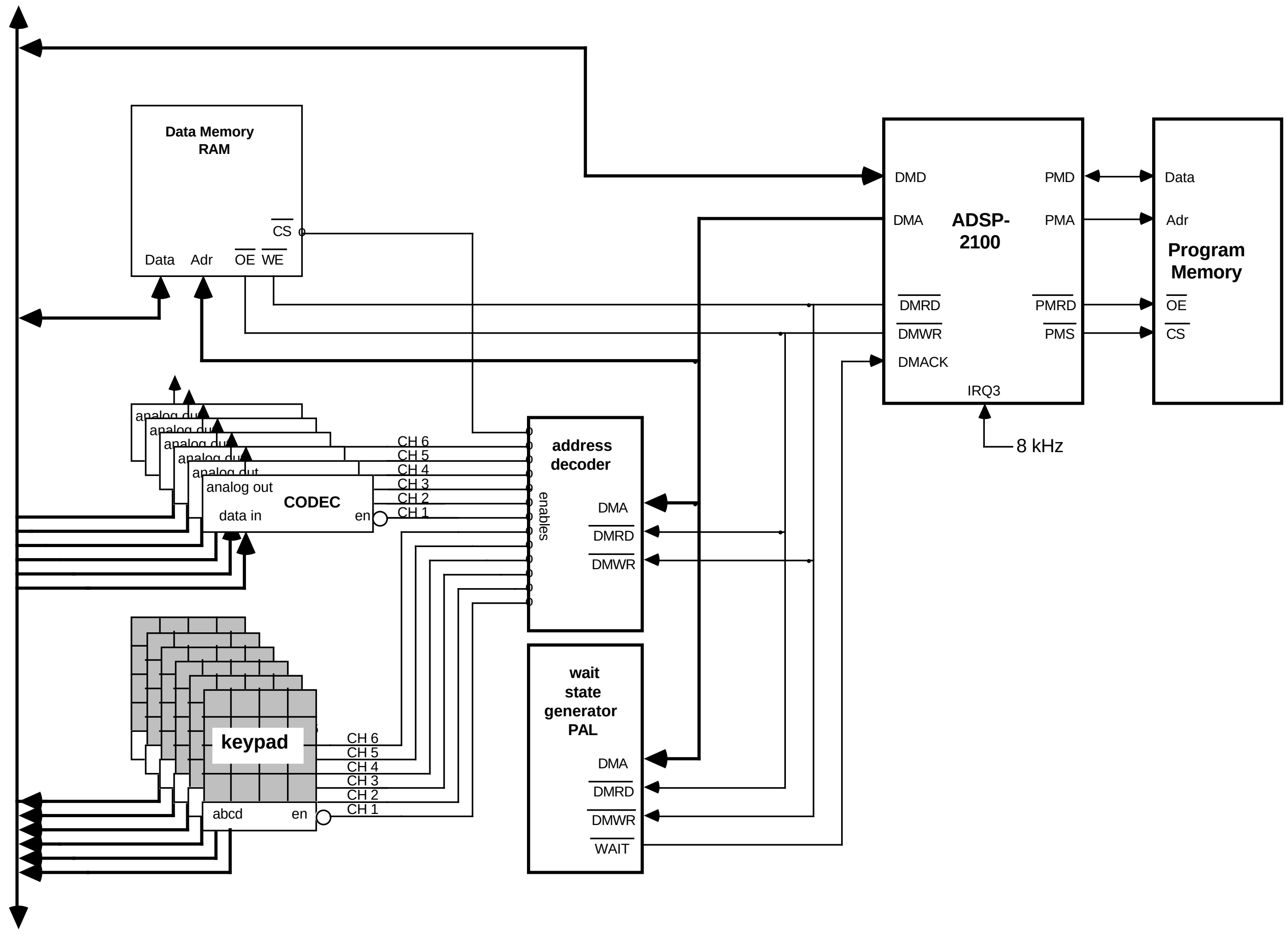
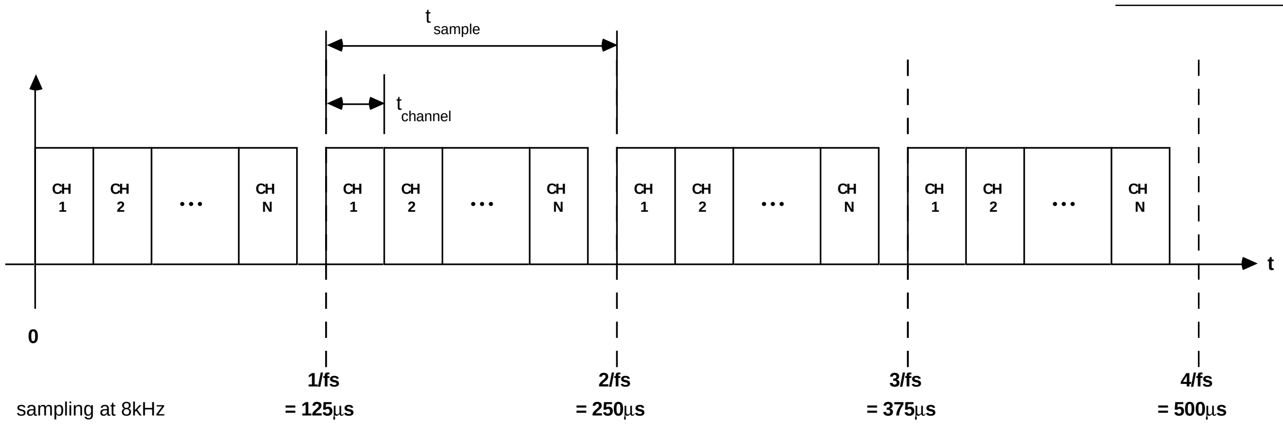


Figure 14.8 Six-Channel Schematic



$t_{channel}$ = time to calculate and output a sample within a channel

t_{sample} = time between interrupts (sampling period)

total number of channels possible, $N = \text{TRUNCATE} \left(\frac{t_{sample}}{t_{channel}} \right)$

Figure 14.9.

Figure 14.9 Maximum Number of Channels (Encoding)

14.5 DECODING DTMF SIGNALS

Decoding a DTMF signal involves extracting the two tones in the signal and determining from their values the intended DTMF digit. Tone detection is often done in analog circuits by detecting and counting zero-crossings of the input signal. In digital circuits, tone detection is easier to accomplish by mathematically transforming the input time-domain signal into its frequency-domain equivalent by means of the Fourier transform.

14.5.1 DFTs and FFTs

The discrete Fourier transform (DFT) or fast Fourier transform (FFT) can be used to transform discrete time-domain signals into their discrete frequency-domain components. The FFT (described in Chapter 7, *Fast Fourier Transforms*) efficiently calculates all possible frequency points in the DFT (e.g., a 256-point FFT computes all 256 frequency points). On the other hand, the DFT can be computed directly to yield only some of the points, for example, only the 20th, 25th, and 30th frequency points out of the possible 256 frequency points. Typically, if more than $\log_2 N$ of N points are desired, it is quicker to compute all the N points using an FFT and discard the unwanted points. If only a few points are needed, the DFT is faster to compute than the FFT. The DFT is faster for finding only eight tones in the full telephone-channel bandwidth.

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \qquad \text{where } k = 0, 1, \ldots, N-1 \text{ and } W_N^{nk} = e^{-j(2\pi/N)nk}$$

The definition of an N-point DFT is as follows:
A single frequency point of the N points is found by computing the DFT

$$X(15) = \sum_{n=0}^{N-1} x(n)W_N^{15n} \qquad \text{where } k = 15 \text{ and } W_N^{15n} = e^{-j(2\pi/N)15n}$$

for only one k index within the range $0 \le k \le N-1$. For example, if k=15:

14.5.2 Goertzel Algorithm

The Goertzel algorithm evaluates the DFT with a savings of computation and time. To compute a DFT directly, many complex coefficients are required. For an N-point DFT, N^2 complex coefficients are needed. Even for just a single frequency in a N-point DFT, the DFT must calculate (or look up) N complex coefficients. The Goertzel algorithm needs only two coefficients for every frequency: one real coefficient and one complex coefficient.

The Goertzel algorithm computes a complex, frequency-domain result just as a DFT does, but the Goertzel algorithm can be modified algebraically so that the result is the square of the magnitude of the frequency component (a real value). This modification removes the phase information, which is irrelevant in the tone detection application. The advantage of this modification is that it allows the algorithm to detect a tone using only one, real coefficient.

Not only is the number of coefficients reduced, but the Goertzel algorithm can process each sample as it arrives. There is no need to buffer N samples before computing the N-point DFT, nor do any bit-reversing or windowing. As shown in Figure 14.10, the Goertzel algorithm can be though of as a second-order IIR filter.

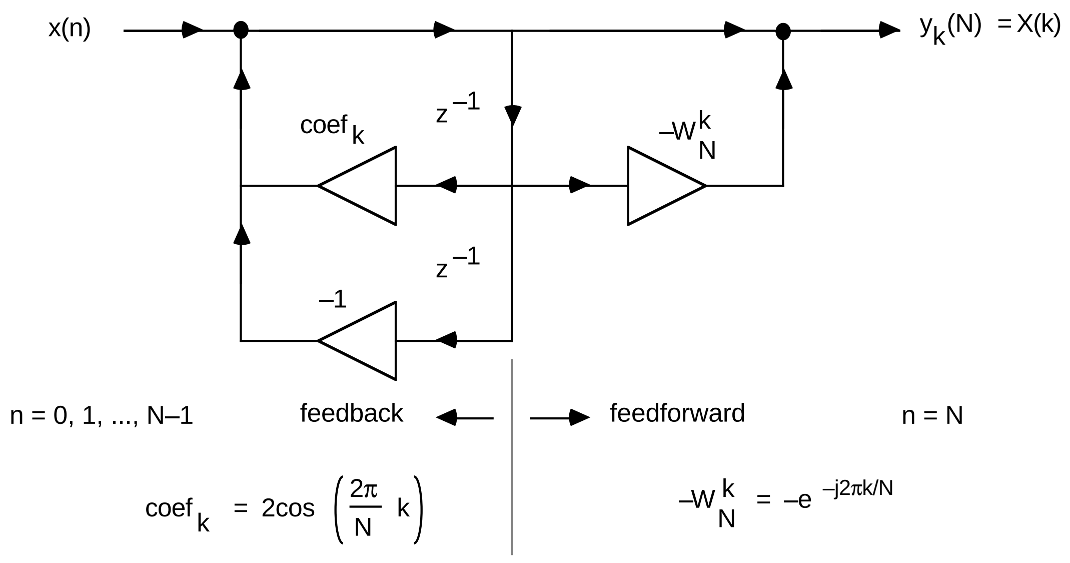


Figure 14.10 Goertzel Algorithm

The Goertzel algorithm can be used to compute a DFT; however, its implementation has much in common with filters. A DFT or FFT computes a buffer of N output data items from a buffer of N input data items. The transform is accomplished by first filling an input buffer with data, then computing the transform of those N samples, yielding N results. By contrast, an IIR or FIR filter computes a new output result with each occurrence of a new input sample. The second-order recursive computation of the DFT by means of the Goertzel algorithm as shown in Figure 14.10 computes a new $y_k(n)$ output for every new input sample $x(n)$. The DFT result, $X(k)$, is equivalent to $y_k(n)$ when $n=N$, i.e., $X(k)=y_k(N)$. Since any other value of $y_k(n)$, in which $n \neq N$, does not contribute to the end result $X(k)$, there is no need to compute $y_k(n)$ until $n=N$. This implies that the Goertzel algorithm is functionally equivalent to a second-order IIR filter, except that the one output result of the filter is generated only after N input samples have occurred.

Computation of the Goertzel algorithm can be divided into two phases. The first phase involves computing the feedback legs in Figure 14.10 as depicted in Figure 14.11 on the next page. The second phase evaluates X(k) by computing the feedforward leg in Figure 14.10 as shown in Figure 14.12, also on the next page.

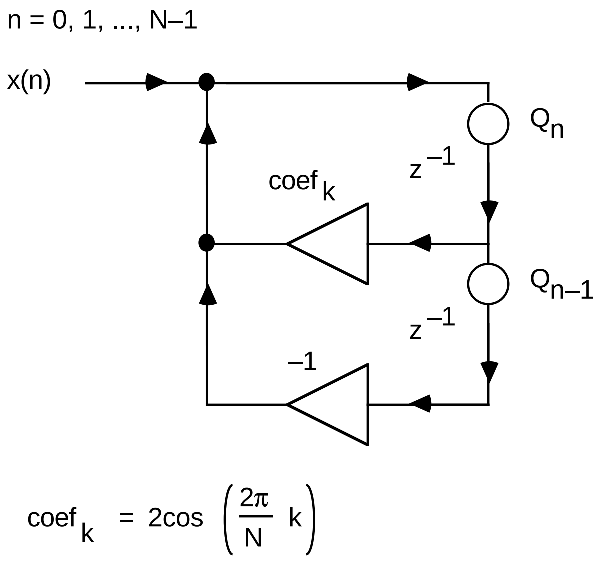


Figure 14.11 Feedback Phase

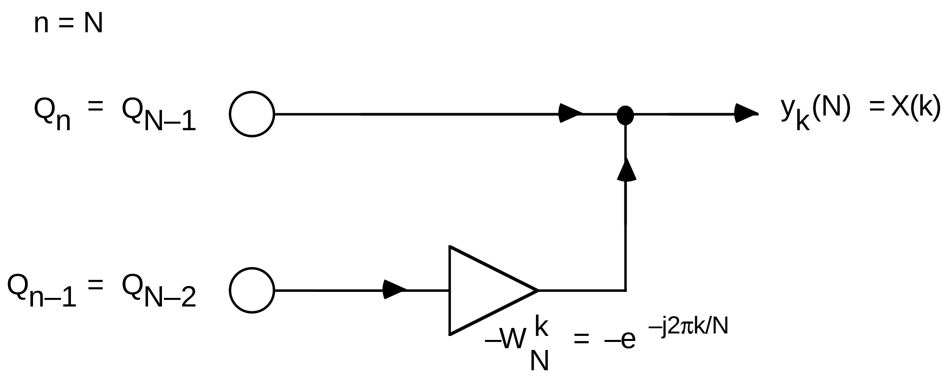


Figure 14.12 Feedforward Phase

14.5.2.1 Feedback Phase

The feedback phase occurs for N input samples (counted as n=0, 1,..., N–1). During this phase, two intermediate values Q(n) and Q(n–1) are stored in data memory. Their values are evaluated as follows:

$$Q_k(n) = coef_k \times Q_k(n-1) - Q_k(n-2) + x(n)$$

where:

$coef_k = 2\cos(2\pi k/N)$
$Q_k(-1) = 0$
$Q_k(-2) = 0$
$n = 0, 1, 2, ..., N-1$
$Q_k(n-1)$ and $Q_k(n-2)$ are the two feedback storage elements for frequency point k and x(n) is the current input sample value

Upon each new sample x(n), Q(n–1) and Q(n–2) are read out of data memory and used to evaluate a new Q(n). This new Q(n) is stored where the old Q(n) was. That old Q(n) value is put where the old Q(n–1) was. This action updates Q(n–1) and Q(n–2) for every sample.

14.5.2.2 Feedforward Phase

The feedforward phase occurs once after the feedback phase has been performed for N input samples. The feedforward phase generates an output sample. During computation of the feedforward phase, no new input is used, i.e., new inputs are ignored. As shown in Figure 14.12, the complex value X(k), equivalent to the same X(k) calculated by a DFT, is computed using the two intermediate values Q(n) and Q(n–1) from the feedback phase calculations. At this time, those two intermediate values

$$y_k(N) = Q_k(N-1) - W_N^k\, Q_k(N-2) = X(k)$$

are Q(N–1) and Q(N–2) since n=N–1. X(k) is calculated as follows:
As stated previously, tone detection does not require phase information, and through some algebraic manipulation, the Goertzel algorithm can be modified to output only the squares of the magnitudes of X(k) (magnitudes squared). This implementation not only saves time needed to compute the magnitude squared from a complex result, but also eliminates the need to do any complex arithmetic. The modified Goertzel algorithm is exactly the same as the Goertzel algorithm during the

feedback phase, but the feedforward phase is simplified. The magnitude squared of the complex result is expanded in terms of the values available at the end of the feedback iterations. The complex coefficient thereby becomes unnecessary, and the only coefficient needed is conveniently the same as the real coefficient previously used in the feedback phase. The

$$
\begin{aligned}
y_k(N) &= Q_k(N-1) \; - \; W_N^k Q_k(N-2) \\
&= A - B\, W_N^k \\
&= A - B\, e^{-j\left(\frac{2\pi}{N}k\right)} \\
&= A - B^{-j\,ø} \\
&= A - B\; [\,\cos ø - j \sin ø\,] \\
&= A - B \cos ø + j\, B \sin ø
\end{aligned}
$$

$$
\begin{aligned}
\left|y_k(N)\right|^2 &= (\text{Real Part})^2 + (\text{Imag. Part})^2 \\
&= (A - B \cos ø)^2 + (B \sin ø)^2 \\
&= A^2 - 2AB \cos ø + B^2 \cos^2 ø + B^2 \sin^2 ø \\
&= A^2 - 2AB \cos ø + B^2 (\cos^2 ø + \sin^2 ø) \\
&= A^2 - AB\, (2\cos ø) + B^2
\end{aligned}
$$

$$\left|y_k(N)\right|^2 = A^2 + B^2 - AB\, \text{coef}_k \qquad \text{where coef}_k = 2 \cos\left(\frac{2\pi}{N}k\right)$$

$$\left|y_k(N)\right|^2 = \left|X(k)\right|^2$$

formula for the magnitude squared is derived as follows:

The DTMF decoder calculates magnitudes squared of all eight fundamental tones as well as the magnitude squared of each fundamental's second harmonic. This information is used in one of the validation tests to determine if tones received make up a valid DTMF digit. Specifically, it will be used to give the DTMF decoder the ability to discriminate between pure DTMF sinusoids and speech. Speech waveforms may also contain sinusoids similar to DTMF digits, but speech also has energy in higher-order harmonics, typically the second harmonic. This test is described later.

The modified Goertzel algorithms (one for each k value) have the ability to detect tones using less computation time and fewer stored coefficients than an equivalent DFT would require. For each frequency to detect, the modified Goertzel algorithms need only one real coefficient. This coefficient is used both in the feedback and feedforward phases.

14.5.2.3 Choosing N and k

Determining the coefficient's value for a given tone frequency involves a trade off between accuracy and detection time. These parameters are dependent on the value chosen for N. If N is very large, resolution in the frequency domain is very good, but the length of time between output samples increases, because the feedback phase of Goertzel algorithm is executed N times (once on each input sample) before the feedforward phase is executed once (yielding a single output sample).

If tone detection had been implemented using FFTs, the values of N would have been limited to those that were a power of the radix of the FFT: 16 point, 32 point, 64 point, 128 point, 256 point, etc. for radix-2 (power of 2) FFTs and 16 point, 64 point, 256 point, 1024 point, etc. for radix-4 (power of 4) FFTs. DFTs and Goertzel algorithms, however, are not limited to any radix. These can be computed using any integer value for N.

When an N-point DFT is being evaluated, N input samples (equally spaced in time) are processed to yield N output samples (equally spaced in frequency). The N output samples are:

$$X(k) \quad \text{where } k = 0, 1, 2, \ldots, N-1$$

The spacing of the output samples is determined by half the sampling frequency divided by N. If some tone is present in the input signal which does not fall exactly on one of these points in the frequency domain, its frequency component appears partly in the closest frequency point, and partly in the other frequency points. This phenomenon is called leakage. To avoid leakage, it is desirable for all the tones to be detected to be exactly centered on a frequency point. The discrete frequency points are referenced by their k value. The value of k can be any integer within the range 0, 1, 2, ..., N–1. The actual frequency to which k corresponds is dependent on the sampling frequency and N as determined by the

$$\left(\frac{f_{tone}}{f_{sampling}}\right) = \frac{k}{N}$$

following formula:
or

$$k = \left(\frac{N}{f_{sampling}}\right) \bullet f_{tone}$$

where f_{tone} is the tone frequency being detected and k is an integer.

Since the sampling frequency is set at 8kHz by the telephone system, and the tones to detect are the DTMF tones, which are also set, the only variable we can modify is N. The numbers k must be integers, so the corresponding frequency points may not be exactly the DTMF frequencies desired. The corresponding absolute error is defined as the difference between what k would be if it could be any real number and the closest

$$\text{absolute k error} = \left| \left(\frac{Nf_{tone}}{f_{sampling}}\right) - \text{CLOSEST INTEGER}\left(\frac{Nf_{tone}}{f_{sampling}}\right) \right|$$

integer to that optimal value. For example:
Bell Labs specifically chose the DTMF tones such that they would not be harmonically related. This makes it difficult to choose a value N for which all tones exactly match the DTMF frequency points. A solution could be to perform separate Goertzel algorithms (each with a different value of N) for each tone, but that would involve a lot of non-computational processor overhead. Instead, in this example, values of N were chosen for which the maximum absolute k error of any one of the tones was considered acceptably small. Then, the length of time to detect a tone (which is

proportional to the sampling rate multiplied by N) was taken into consideration. The value of N best suited for detecting the eight DTMF fundamental tone frequencies was chosen to be 205. The value of N best suited for detecting the eight second harmonic frequencies was chosen to be 201. See Table 14.6 for the corresponding k values and their respective absolute errors. These values of k allow Goertzel outputs to occur approximately once every 26 milliseconds.

Fundamental Frequency (N=205)	*k value floating-point*	*k value nearest integer*	*Absolute k error*	*$coef_k$*
697.0Hz	17.861	18	0.139	1.703275
770.0Hz	19.731	20	0.269	1.635859
852.0Hz	21.833	22	0.167	1.562297
941.0Hz	24.113	24	0.113	1.482867
1209.0Hz	30.981	31	0.019	1.163138
1336.0Hz	34.235	34	0.235	1.008835
1477.0Hz	37.848	38	0.152	0.790074
1633.0Hz	41.846	42	0.154	0.559454
Second Harmonic (N=201)				
1394.0Hz	35.024	35	0.024	0.917716
1540.0Hz	38.692	39	0.308	0.688934
1704.0Hz	42.813	43	0.187	0.449394
1882.0Hz	47.285	47	0.285	0.202838
2418.0Hz	60.752	61	0.248	–0.659504
2672.0Hz	67.134	67	0.134	–1.000000
2954.0Hz	74.219	74	0.219	–1.352140
3266.0Hz	82.058	82	0.058	–1.674783

sampling frequency = 8kHz

Table 14.6 Values of k and Absolute k Error

14.5.3 DTMF Decoding Program

For DTMF decoding, the ADSP-2100 solves sixteen separate modified Goertzel algorithms, eight of length 205 to detect the DTMF fundamentals, and eight of length 201 to detect the DTMF second harmonics. To implement concurrent Goertzel algorithms of lengths 205 and 201, the feedback phase iterations of all the Goertzel algorithms (fundamentals and second harmonics) are performed for 201 samples (n=0, 1,..., 200). For the next four samples, only the Goertzel algorithms of length 205 (fundamentals) are iterated (n=201, 202, 203, 204). The other Goertzel algorithms of length 201 (second harmonics) ignore the new samples. On the last iteration, when n=N=205, all the feedforward phases are evaluated (both fundamentals and second harmonics), and any new input samples at that time are ignored. In the DTMF decoder application presented here which uses the modified Goertzel algorithm, the magnitude squared calculations are performed for the feedforward phases.

DTMF decoding is done in two major tasks, as shown in the block diagram in Figure 14.13. The first task solves sixteen Goertzel algorithms to calculate the magnitudes squared of tone frequencies present in the input signal, then the second task tests the frequency results to determine if the tones detected constitute a valid DTMF digit. The first task spans N+1 processor interrupts, N for the feedback phases of the Goertzel algorithms, then one more for the feedforward phases. The second task immediately follows completion of all sixteen feedforward phases. The length of time required by the processor for this testing may span the next few interrupts (during which time new input samples are ignored), but since the number of input samples lost is small compared to the number of interrupts serviced during the Goertzel evaluations, the loss is insignificant (see Figure 14.15). The Goertzel algorithms are not sensitive to incoming signal phase, and therefore no phase synchronization is attempted.

14.5.3.1 Input Scaling

It is important to notice that the input sample values are scaled down by eight bits to eliminate the possibility of overflows within 205 iterations of the feedback phase. Scaling by eight bits increases the quantization error of the input samples, but this does not affect the effectiveness of the decoder. Input samples are read from a μ-law-compressed codec. The 8-bit data values are used as offset values to a μ-law-to-linear conversion look-up table. The corresponding linear values are scaled such that the input samples range from H#007F to H#FF80 instead of the normalized equivalent range of H#7FFF to H#8000.

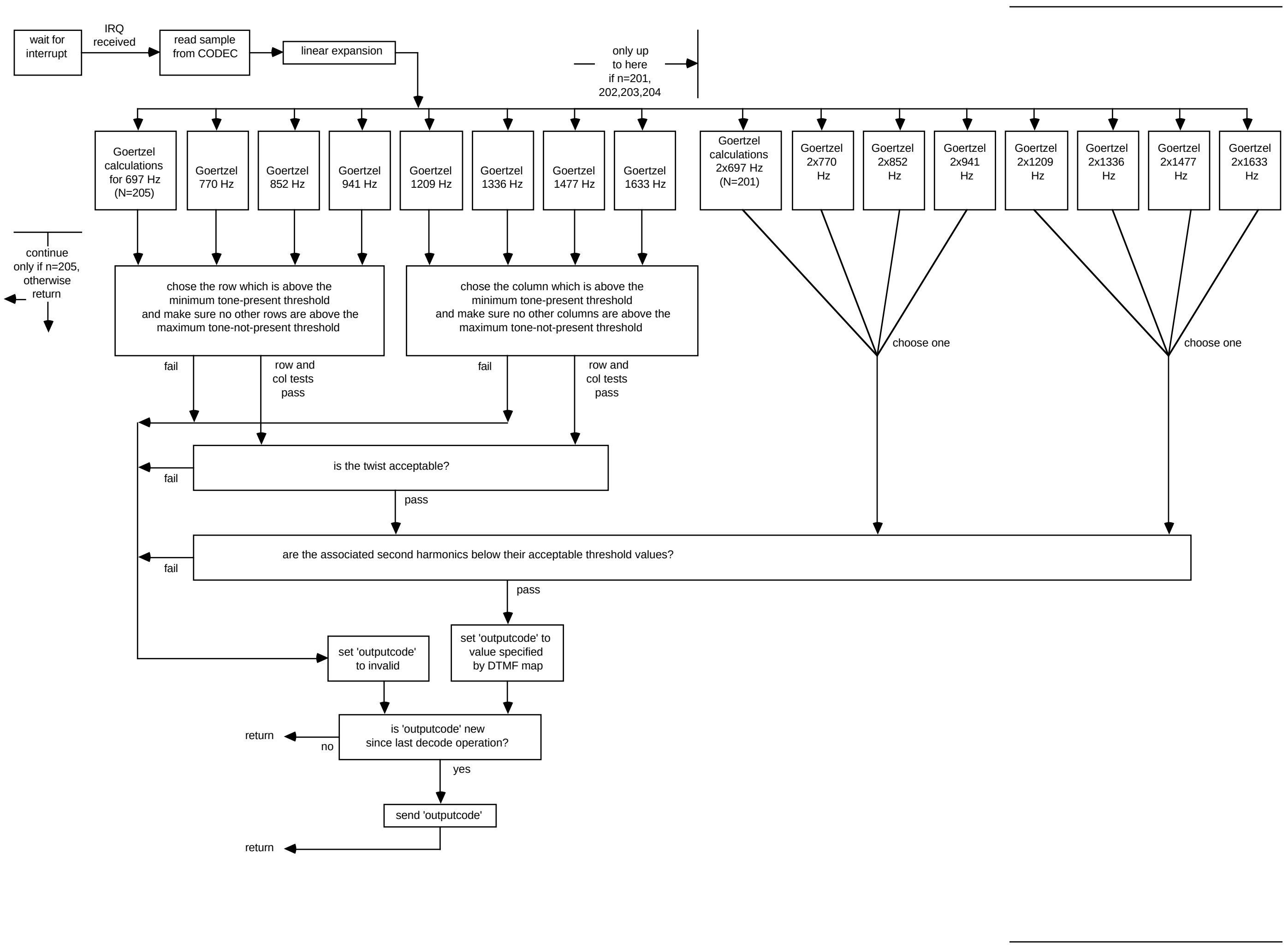

wait for interrupt
IRQ received
read sample from CODEC
linear expansion
only up to here if n=201, 202,203,204
Goertzel calculations for 697 Hz (N=205)
Goertzel 770 Hz
Goertzel 852 Hz
Goertzel 941 Hz
Goertzel 1209 Hz
Goertzel 1336 Hz
Goertzel 1477 Hz
Goertzel 1633 Hz
Goertzel calculations 2x697 Hz (N=201)
Goertzel 2x770 Hz
Goertzel 2x852 Hz
Goertzel 2x941 Hz
Goertzel 2x1209 Hz
Goertzel 2x1336 Hz
Goertzel 2x1477 Hz
Goertzel 2x1633 Hz
continue only if n=205, otherwise return
chose the row which is above the minimum tone-present threshold and make sure no other rows are above the maximum tone-not-present threshold
chose the column which is above the minimum tone-present threshold and make sure no other columns are above the maximum tone-not-present threshold
choose one
choose one
fail
row and col tests pass
fail
row and col tests pass
is the twist acceptable?
fail
pass
are the associated second harmonics below their acceptable threshold values?
fail
pass
set 'outputcode' to invalid
set 'outputcode' to value specified by DTMF map
is 'outputcode' new since last decode operation?
return
no
yes
send 'outputcode'
return

Figure 14.13 Tone Decoder Block Diagram

14.5.3.2 Multi-Channel DTMF Decoder Software

An example of a multi-channel DTMF decoder is given in Listing 14.2. This example is a 6-channel version, although at least twelve channels can be decoded by an ADSP-2100A running at a 12.5MHz instruction rate. The six channels are labeled channel A, channel B, ..., channel F. The following software description gives an overview of the variables and constants used, then outlines the core executive routine and describes the interrupt service routine. The macros and subroutines are explained as needed.

14.5.3.3 Constants, Variables and I/O Ports

The number of channels to decode is specified in the .CONST declaration section. The number of channels must be greater than one and less than or equal to the maximum number of channels allowed, which is dependent on the processor cycle time. The faster the processor cycle time, the more channels can be decoded in real time. The limiting factor is how many of the Goertzel feedback operations can be performed between successive processor interrupts (see Figure 14.14). To decode a single channel only, the source code must be slightly altered. The only necessary alteration involves the circular buffer of length *channels* which stores the input samples; this buffer must be changed to a single data memory variable. Although not necessary, other alterations can be done to optimize the software for a single channel if desired.

The hexadecimal value which the decoder outputs when no DTMF digit is received is defined by the constant called *baddigitcode*. For debugging purposes, a variable called *failurecode* was incorporated into each channel. This variable is assigned a value whether or not a valid DTMF digit was received. If a valid digit was received, *failurecode* is set to zero. A nonzero *failurecode* means that the signals received failed one of the qualifying tests. The *failurecode* value, defined in the constant declaration section, identifies which test failed.

The data variables are separated into two functional groups: housekeeping variables and individual-channel variables. The housekeeping variables are used by the software shared by all sections of the decoder. The individual channel variables are used to keep track of specifics for each channel separately. The variables are explained in detail in Listing 14.2.

The input port for each channel is a codec. The output port for each channel is a D/A converter in this example. In a more realistic application, such as a PBX, switching machine, or electronic voice-mail system, the output would probably be sent to dual-port memory or a mailbox register for use by another processor.

The initialization directives set up some of the static, housekeeping variables. The Goertzel coefficients are initialized as well as the µ-law-to-linear look-up table, thresholds for the received signal levels and twist test limits. (The file containing the µ-law look-up values is not included in the program listings; however, it is included in the diskette that contains the programs in this manual, which is available from your local Analog Devices Sales Office.) These initializations could be adjusted for application-specific requirements, such as meeting the specifications of other administrations or operating as a DTMF signal tester.

To decode more or fewer than six channels, simply add or remove sections identified by the comment “edit this for more channels” in the source code listing.

14.5.3.4 Main Code

The main code section is very simple. The processor is initialized for the decoding operation, then put into an endless loop, waiting for interrupts that occur at the sampling rate (8kHz). After the ADSP-2100 is reset, the first task is to set up the static environment, such as the M and L registers in both data address generators and the ICNTL register. This task is done only once since these initializations are never changed. This set-up is performed by the subroutine called *setup*. Another subroutine called *restart* then initializes other variables which are needed for the decoding operation, but which change and must therefore be reinitialized after each decode operation. The specific tasks here include resetting the I registers of the data address generators to the top of their associated buffers, zeroing out the delay elements for the Goertzel algorithm implementation (Q values), and resetting the counters which keep track of the input sample number (n). The *restart* routine is called after every decode operation, immediately after completion of the digit validation tests, as well as before the very first decode operation.

14.5.3.5 Interrupt Service Routine

The ADSP-2100 is interrupted at an 8kHz sampling rate. The first task done by the processor is to read a new input sample from each codec and store the input samples in a buffer. Next, counters are decremented and tested to determine which sample (n = 0, 1, 2, ..., 205) is currently being processed. If n is between 0 and 200 (inclusive), Goertzel feedback operations are performed on sixteen frequencies per channel, eight fundamental tones and eight second harmonic tones. If n is 201, 202, 203, or 204, then Goertzel feedback operations are performed on the eight fundamental tones for each channel only. The second harmonics are skipped. If n is 205, then magnitude-squared calculations are performed on the eight fundamental tones of each channel, the post testing and digit validation takes place, a new digit is output if necessary, and the *restart* routine is called to reinitialize the processor for the next decoding operation.

Since interrupt nesting mode is not enabled, and since no other interrupts are being used, the processor should never see another interrupt until it has finished executing the interrupt service routine. You need to ensure that the processor does not attempt to decode too many channels, in which case the length of time to perform the sixteen Goertzel operations per channel during samples 0 through 200 would be greater than the sampling period (125µs) (see Figure 14.14). There is one special case in which an interrupt may be overlooked. This is when n=205. In this case, the length of time it takes to perform the magnitude-squared calculations and all the digit validation tests for all channels may exceed the sampling period. Since losing a few input samples out of 205 samples is relatively insignificant, and since incoming signal phase is unimportant, this overlap can be disregarded.

14.5.3.6 Post-Testing and Digit Validation

When calling the macros in the source code listing, the various channels are identified by prefixing each channel's variables with an alphabetic character and underscore. For example, the macro *maxrowcol* is called thus:

```
maxrowcol
(^A_mnsqr,A_maxrowval,A_whichrow,A_maxcolval,A_whichcol)
maxrowcol
(^B_mnsqr,B_maxrowval,B_whichrow,B_maxcolval,B_whichcol)
```
etc.

Each channel is tested sequentially, identifying each channel's variables by alphabetic prefixes.

Maxrowcol

After completion of the magnitude-squared computations for the eight fundamental tones of each channel, those results reside in each channel's *mnsqr* buffer. The *maxrowcol* macro scans through the results and picks out the largest row result, storing its value in the variable *maxrowval* and its index in the variable *whichrow*. *Whichrow* can be 1, 2, 3, or 4. These values correspond to the frequencies 697Hz, 770Hz, 852Hz, and 941Hz. The column results are likewise scanned, and the largest value and its index assigned to *maxcolval* and *whichcol*. Subsequent testing could set *whichrow* or *whichcol* to zero, indicating that some validation test has failed.

Minsiglevel

The *minsiglevel* macro checks the largest row result and the largest column result chosen to determine whether or not each value exceeds the minimum level necessary for a valid tone. Each tone has its own minimum level threshold in the buffer called *min_tone_level*. Each tone was given its own threshold because of the absolute k error (see previous section on chosing k and N). The DTMF tone frequencies do not correspond exactly to integer multiples of 205/8000; in fact, each tone has a different absolute k error making it necessary to test each magnitude-squared result independently.

The macro *minsiglevel* takes the address of the *min_tone_level* buffer and adds to it the value of *whichrow* minus one. The resulting address is used to look up the minimum signal threshold for that particular row tone. The magnitude squared, stored in *maxrowval*, is compared to the threshold. Failure here sets *whichrow* and *whichcol* each to zero, sets *failurecode* to H#0001 and exits. Otherwise, the row tone passes the test, and the column tone is checked in the same manner as row tone.

No_Other_Peaks

DTMF specifications require that the decoder detect a digit if and only if one row tone is present as well as one and only one column tone is present. The *no_other_peaks* macro makes sure that all the tones other than the maximum row and column tones are below the non-digit threshold.

As in the *minsiglevel* test, this test uses independent thresholds for each tone. The thresholds are stored in the buffer called *max_notone_level*. However, instead of computing each address independently as in the minimum signal level test, this test scans through the whole result (*mnsqr*) buffer and increments a counter (AF register) once for each tone which exceeds the maximum no-tone level. At the end of the scan, the AF register should contain the value H#0002, for one valid row tone and one valid column tone. If any other number is in the AF register, this test fails. If the test fails, *whichrow* and *whichcol* are each set to zero and *failurecode* is set to H#0002.

Twisttests

Twist is the difference, in decibels, between the row tone level and the column tone level. Forward twist (also called standard twist) exists when the column tone level is greater than the row tone level. Reverse twist exists when the column tone is less than the row tone level. DTMF digits are often generated with some forward twist to compensate for greater losses at higher frequencies within a long telephone cable. Different administrations recommend different amounts of allowable twist for DTMF receivers. For example, CEPT recommends not more than 6dB of either twist, Brazil allows 9dB, Australia allows 10dB, Japan allows only 5dB, and AT&T recommends not more than 4dB of forward twist or 8dB of reverse twist.

The twist test macro uses the variables *maxrowcol*, *maxcolval*, *whichrow*, and *whichcol* to compute the twist value, setting *twistval* and comparing that value against the predefined twist limits stored in the variables *maxfortwist* and *maxrevtwist*. The macro sets *failurecode* to H#0003 upon failure and sets either the flag *fortwistflag*, or *revtwistflag* as appropriate.

First, the row tone level is compared to the column tone level. If the row tone is greater, program flow jumps to the label *reverse*; otherwise, it continues at *standard*. The standard twist test divides *maxrowval* by *maxcolval* and compares the resultant ratio to *maxfortwist*. (A ratio of powers is equivalent to a difference in decibels.) *Maxfortwist* is the ratio that would result if the greatest allowable twist was encountered. Any ratio which is between that value and unity passes the twist test.

Likewise, if the column tone level was greater, *maxcolval* is divided by *maxrowval*, and the resulting ratio is compared to *maxrevtwist*. If the ratio is greater than *maxrevtwist*, the twist test passes. Of course, a twist value of 0dB would result in a ratio of unity. A very large twist value would result in a very small ratio. The ratio (row or column) is calculated in such a way to ensure that the numerator is always smaller than the denominator. This is done because of how the ALU of the ADSP-2100 performs division.

Check2ndharm

The last item to verify is the level of second harmonic energy present in the detected row and column tones. This test is performed to help the decoder reject speech which might be detected as DTMF tones. This property is referred to as talk-off. DTMF tones should be pure sinusoids, and therefore contain very little second harmonic energy, if any. Speech, on the other hand, contains significant amount of second harmonic energy.

To test the level of second harmonic energy present, the decoder must concurrently evaluate Goertzel algorithms for the second harmonic frequency of all eight DTMF fundamental tones. The second harmonic frequencies (1394Hz, 1540Hz, 1704Hz, 1882Hz, 2418Hz, 2672Hz, 2954Hz, and 3266Hz) can be detected at an 8kHz sampling rate (concurrently with the fundamentals) using Goertzel algorithms of length N=201. This is conveniently close to the length N=205 chosen for the fundamental tones.

During the execution of the Goertzel feedback section of code, both the fundamentals and the second harmonic tones are processed until the counter variable called *count201* expires. For the next four interrupts, another counter called *count4* controls the Goertzel feedback operations. In the latter case, only the eight fundamentals are processed for each channel. The second harmonics are skipped. After the 205th sample has been processed, the *Q1Q2_buff* buffer contains the Goertzel feedback results of the fundamentals at N=205 and of the second harmonics at N=201. When the next interrupt is received, the magnitude-squared computations are carried out for the eight fundamentals of each channel, but no processing is done on the second harmonics yet.

After all the other DTMF digit validation tests are performed, the *check2ndharm* macro carries out the magnitude-squared computations for the second harmonics. But at this time, only two magnitude-squared calculations are performed, one for each detected DTMF tone. This saves the time which would have been wasted if all eight second harmonics had been computed concurrently with the eight fundamentals.

To check the second harmonic level, the address of the channel's Goertzel feedback buffer is passed the *check2ndharm* macro along with the detected tone index variables *whichrow* and *whichcol.* The addresses of the Goertzel feedback values are calculated from the base address plus the index values. The magnitude-squared subroutine is called, and the results are stored in the variables called *rowharm* and *colharm*. Those results are compared to each tone's maximum second harmonic level threshold. The thresholds are stored in the buffer *max_2nd_harm* and, like the fundamental thresholds, are each independently adjustable for each tone.

Outputcode

The last task performed during a decoding sequence is to output a code representing the DTMF digit to the output port. In many applications, this would probably involve writing a hexadecimal code to dual-port RAM or a mailbox for use by the host processor in a PBX, electronic mail system, or digital telephone switch. In this software example, the code is written to a D/A converter. The output of the D/A converter can be used to deflect the vertical trace of an oscilloscope to monitor decoder activity. The extent of the deflection varies according to which DTMF digit is received, if any. If nothing is received, or an invalid digit is received, the code sent to the D/A converter is the constant *baddigitcode*. If a valid digit is received, the index variables *whichrow* and *whichcol* are used to compute the code as follows:

$$\text{16-bit code} = 4096 \times [4(\mathit{whichrow}-1) + (\mathit{whichcol}-1)]$$

In effect, a hexadecimal digit is generated and placed in the most significant hexadecimal digit (4 bits) position of a 4-digit hexadecimal (16 bits) word. The three less significant hexadecimal digits are set to zeros. See Table 14.7 for the one-to-one relationship between received DTMF digits and hexadecimal code output.

DTMF Digit	*Output Code*	*DTMF Digit*	*Output Code*
1	H#0000	7	H#8000
2	H#1000	8	H#9000
3	H#2000	9	H#A000
A	H#3000	C	H#B000
4	H#4000	*	H#C000
5	H#5000	0	H#D000
6	H#6000	#	H#E000
B	H#7000	D	H#F000
Invalid	H#FFFF (*baddigitcode*)		

Table 14.7 DTMF Tones and Output Codes

The *outputcode* macro not only outputs the appropriate code to the output port, but it also decides whether or not to output anything at all. There must be some distinction made between a long, sustained DTMF signal and several short DTMF signals of the same digit. In other words, it would be undesirable to have the DTMF decoder interrupt a host processor informing it of a stream of new DTMF digits when actually only one DTMF digit was received, but sustained for a long period of time.

For each channel, three decoder-output codes are compared: the current code, the last code, and the next-to-last code. The last and next-to-last codes are stored in the *digit_history* buffers.

The current code is written to the channel's D/A converter if and only if the current code is equal to the last code, but different than the next-to-last code. Whether or not a new digit was detected, the *digit_history* list is updated every time by overwriting the last code with the current code, and overwriting the next-to-last code with the last code. This updates the history list for the next decode operation.

Restart

Before starting the next decode operation, the *restart* subroutine is called. This routine sets all the Goertzel feedback elements to zero, restoring the Goertzel algorithm's initial conditions. The *restart* routine also resets data memory pointers and the two counters (*count201* and *count4*) which keep track of which input sample is being processed.

14.5.3.7 Performance Considerations

The fastest rate of detecting DTMF digits is dictated by how long it takes to perform two successful (all tests passed), sequential decode operations. The amount of time it takes to execute a single decode operation can be calculated as follows:

$$T_{sample} = \frac{1}{f_{sample}} = 125\ \mu s$$

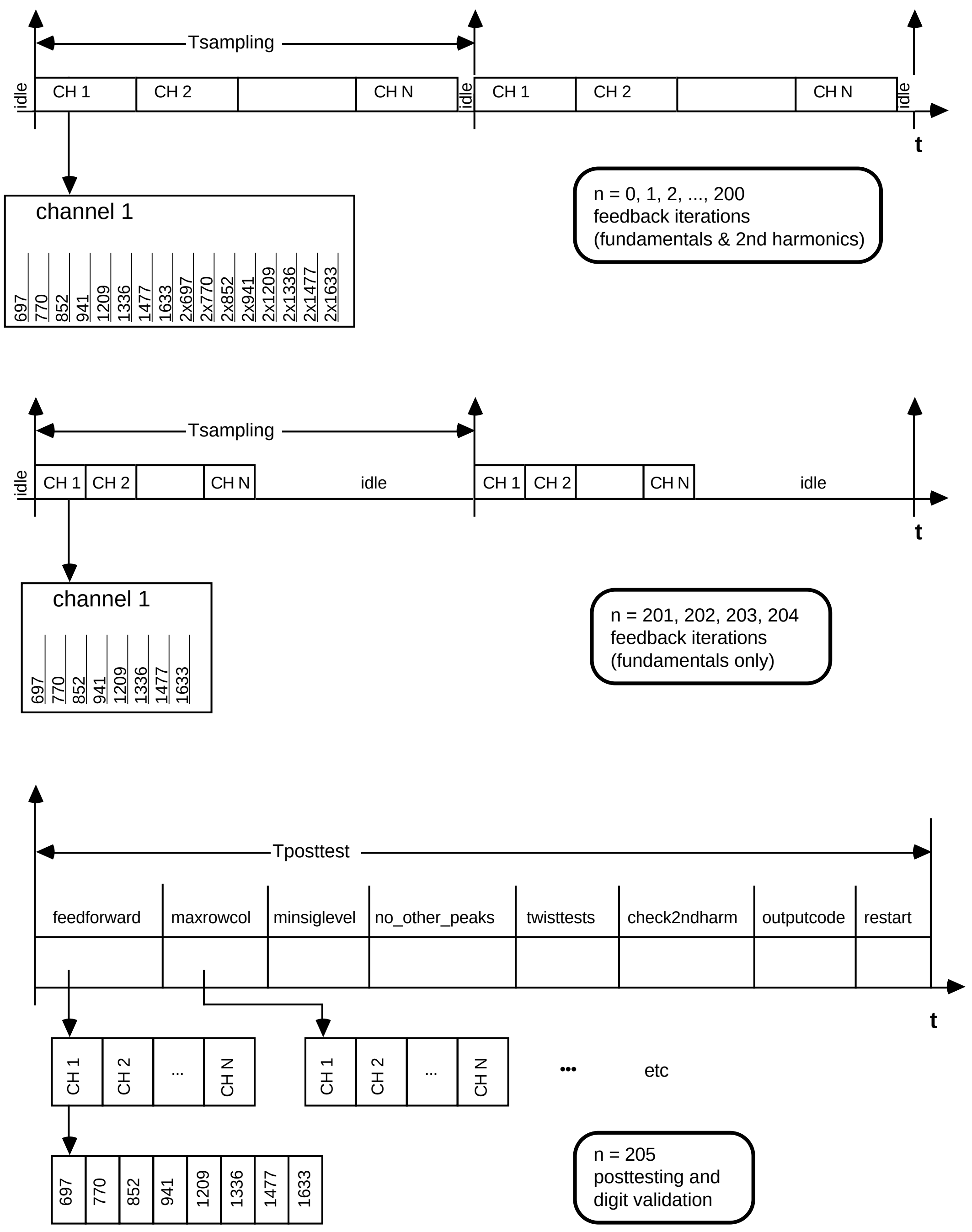
$$201(T_{sample}) + 4(T_{sample}) + T_{posttest}$$

where
and

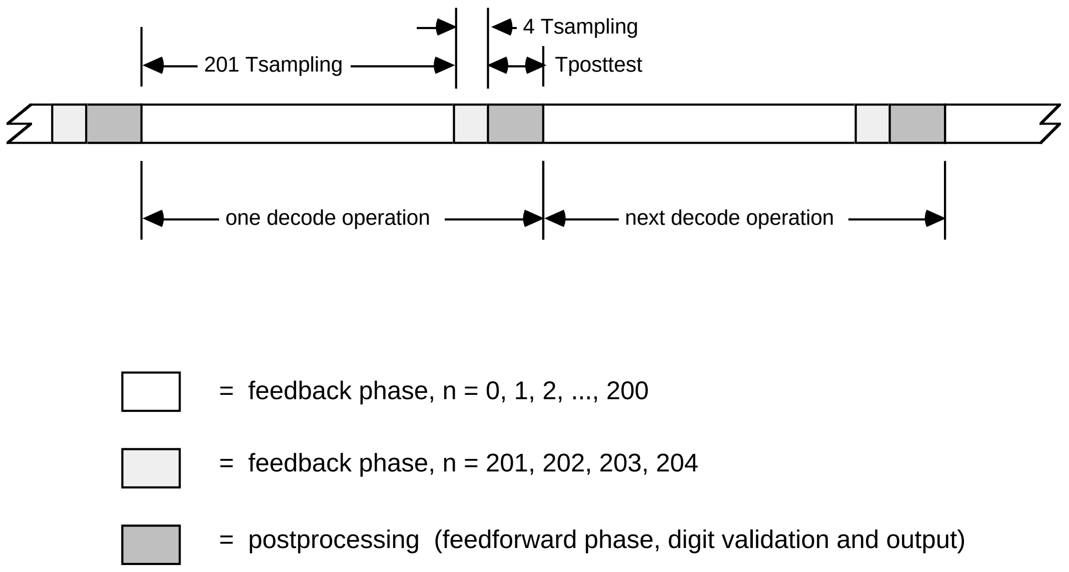
$$T_{posttest} = \text{number of channels}\ (T_{8\_fund\_mnsqr} + T_{each}) + T_{restart}$$

$$T_{each} = T_{maxrowcol} + T_{minsiglevel} + T_{no\_other\_peaks} + T_{twisttests} + T_{check2ndharm} + T_{outputcode}$$

The exact execution time for $T_{posttest}$ varies with the number of channels being decoded, and also with the actual data processed on those channels. Some of the tests could fail early in their executions, thereby skipping



subsequent instructions and testing. Figures 14.14 and Figure 14.15, on the following pages, show processor loading for the 6-channel decoder


4 Tsampling
201 Tsampling
Tposttest
one decode operation
next decode operation
= feedback phase, n = 0, 1, 2, ..., 200
= feedback phase, n = 201, 202, 203, 204
= postprocessing (feedforward phase, digit validation and output)

example in Listing 14.2.

Figure 14.14 Multi-Channel Decoding Timing

Figure 14.15 Multi-Channel Decoder Output Rate

14.6 REFERENCES

Bellamy, John. 1982. *Digital Telephony*. New York: John Wiley & Sons.

Blahut, Richard E. 1985. *Fast Algorithms for Digital Signal Processing*. Reading, MA: Addison-Wesley.

–. 1982. *Reference Data for Radio Engineers*. New York: Howard Sams.

–. 1986. *IEEE Standard for Functional Requirements for Methods and Equipment for Measuring the Performance of Tone Address Signaling Systems*. (ANSI/IEEE Standard 752-1986). New York: IEEE Press.

Burrus, C.S. and T.W. Parks. 1985. *DFT/FFT and Convolution Algorithms*. New York: John Wiley & Sons.

Oppenheim, Alan V. and Ronald W. Schafer. 1975. *Digital Signal Processing*. New York: Prentice-Hall.

Pearce, J. Gordon. 1981. *Telecommunications Switching*. New York: Plenum Press.

Some Administration Specifications

CCITT:
Recommendations Q.23 and Q.24 in Section 4.3 of the CCITT Red Book, Volume VI, Fascicle VI.1.

AT&T:
Compatibility Bulletin No. 105. *TOUCHTONE Calling - Requirements for the Central Office*. 1975.

CEPT:
Recommendations T/CS 46-02, T/STI 46-04, T/CS 28-01, T/CS 34-08, T/CS 34-09, T/CS 42-02, T/CS 49-04, T/CS 49-07, T/CS 01-02, T/CS 14-01. 1986.

British Post Office:
POR 1151. Section 2. Issue 4. 1979.

14.7 PROGRAM LISTINGS

The complete listings for both the DTMF encoder and 6-channel DTMF decoder are presented in this section.

14.7.1 DTMF Encoder Listing

The code below encodes DTMF digits from a list in data memory. It implements the 3-state software state machine described earlier in this chapter. In state 0, no digits are generated; the ADSP-2100 is idle. In state 1, a continuous dial tone (350Hz + 400Hz) is generated. In state 2, the DTMF dialing list is sequentially read and DTMF digits generated. The state machine stays in each state until the IRQ2 interrupt (connected in the

```
{  DTMF Signal Generator using ADSP-2100       }

.MODULE/RAM/ABS=0             DTMF_Dialer;

.VAR     row0,row1,row2,row3,col0,col1,col2,col3;
                            {for reference only}
.VAR     hertz1, hertz2;    {scratchpad storage: frequency}
.VAR     sum1, sum2;        {scratchpad storage: phase accum}
.VAR     sin1, sin2;        {scratchpad storage: sine result}
.VAR     scale;             {divisor for scaling sine waves}
.VAR     state;             {current state of state machine}
.VAR     sign_dura_ms;      {signal duration time in milliseconds}
.VAR     interdigit_ms;     {interdigit time in milliseconds}
.VAR     time_on;           {down counter: tones on }
.VAR     time_off;          {down counter: tones off (silence)}
.VAR     digits[32];        {lookup table for row,col freqs}
.VAR     dial_list[100];    {stores sequence to dial}

{  store values in dial_list as follows:
                                                       (E=*, F=#)
   DTMF tone:     h#000y, y=0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
   quiet space:   h#00yX, y is non-zero, X is don’t care
   redial:        h#0yXX, y is non-zero, X is don’t care
   stop:          h#yXXX, y is non-zero, X is don’t care
}

.PORT    dac;     {DTMF output to D/A for inspection}
.PORT    codec;   {DTMF output also to μ-law codec}

.INIT    row0:    h#02B9;     { 697 Hz}
.INIT    row1:    h#0302;     { 770 Hz}
.INIT    row2:    h#0354;     { 852 Hz}
.INIT    row3:    h#03AD;     { 941 Hz}
.INIT    col0:    h#04B9;     {1209 Hz}
.INIT    col1:    h#0538;     {1336 Hz}
.INIT    col2:    h#05C5;     {1477 Hz}
.INIT    col3:    h#0661;     {1633 Hz}

.INIT    digits[00]:      h#03AD,h#0538, h#02B9,h#04B9,
                          h#02B9,h#0538, h#02B9,h#05C5;
.INIT    digits[08]:      h#0302,h#04B9, h#0302,h#0538,
                          h#0302,h#05C5, h#0354,h#04B9;
.INIT    digits[16]:      h#0354,h#0538, h#0354,h#05C5,
                          h#02B9,h#0661, h#0302,h#0661;
.INIT    digits[24]:      h#0354,h#0661, h#03AD,h#0661,
                          h#03AD,h#04B9, h#03AD,h#05C5;
```

(listing continues on next page)

```
.INIT   scale:            h#FFFC;
.INIT   sign_dura_ms:     h#0032;   {50 ms}
.INIT   interdigit_ms:    h#0032;   {50 ms}
.INIT   dial_list:        h#00F0, h#0003, h#0000, h#0005,
                          h#0008, h#FFFF;

.EXTERNAL      u_compress;
.EXTERNAL      sin;

IRQ0:          RTI;
IRQ1:          RTI;
IRQ2:          JUMP next_state;
IRQ3:          JUMP eight_khz;

setup:         SI=0;
               DM(state)=SI;
               CALL reset;
               L0=0; L1=0; L2=0; L3=0;
               L4=0; L5=0; L6=0; L7=0;
               I0=^dial_list;
               M0=1;
               M3=1; L3=0;    {used by sine routine}
               ICNTL=b#01111;
               IMASK= b#1100;
wait_int:      JUMP wait_int;

eight_khz:     AY0=2;
               AX0=DM(state);
               AR=AX0-AY0;
               IF EQ JUMP state2;
               AF=AY0-1;
               AR=AX0-AF;
               IF EQ JUMP state1;
state0:        RTI;
state1:        AX0=350;
               DM(hertz1)=AX0;
               AX0=440;
               DM(hertz2)=AX0;
               JUMP maketones;
state2:        AY0=DM(time_on);
               AR=PASS ay0;
               IF EQ JUMP quiet;
               AR=AY0-1;
               DM(time_on)=AR;
```

```
maketones:     SE=DM(scale);

tone1:         AY0=DM(sum1);
               SI=DM(hertz1);
               SR=ASHIFT SI BY 3 (HI);   {mult Hz by 8}
               MY0=h#4189;               {mult by 0.512}
               MR=SR1*MY0(RND);          {mult by 2}
               SR=ASHIFT MR1 BY 1 (HI);  {i.e. Hz * 8.192}
               AR=SR1+AY0;
               DM(sum1)=AR;
               AX0=AR;
               CALL sin;
               SR=ASHIFT AR (HI);        {scale value in SE}
               DM(sin1)=SR1;

tone2:         AY0=DM(sum2);
               SI=DM(hertz2);
               SR=ASHIFT SI BY 3 (HI);   {mult Hz by 8}
               MY0=h#4189;               {mult by 0.512}
               MR=SR1*MY0(RND);          {mult by 2}
               SR=ASHIFT MR1 BY 1 (HI);  {i.e. Hz * 8.192}
               AR=SR1+AY0;
               DM(sum2)=AR;
               AX0=AR;
               CALL sin;
               SR=ASHIFT AR (HI);        {scale value in SE}
               DM(sin2)=SR1;

add_em:        AX0=DM(sin1);
               AY0=DM(sin2);
               AR=AX0+AY0;

sound:         AY0=h#8000;
               AR=AR XOR AY0;
               DM(dac)=AR;
               AR=AR XOR AY0;
               CALL u_compress;
               DM(codec)=AR;
               RTI;

quiet:         AY0=DM(time_off);
               AR=PASS AY0;
               IF EQ JUMP nextdigit;
               AR=AY0-1;
               DM(time_off)=AR;
```

(listing continues on next page)

```
                AY0=h#8000;
                AR=h#7FFF;
                DM(dac)=AR;
                AR=AR XOR AY0;
                CALL u_compress;
                DM(codec)=AR;
                RTI;

nextdigit:      CALL reset;
                AX0=DM(I0,M0);      {read next digit out of list}
                AY0=h#F000;
                AR=AX0 AND AY0;
                IF EQ JUMP notstop;
stop:           AR=0;
                DM(state)=AR;
                RTI;
notstop:        AY0=h#0F00;
                AR=AX0 AND AY0;
                IF EQ JUMP notredial;
redial:         I0=^dial_list;
                RTI;
notredial:      AY0=h#00F0;
                AR=AX0 AND AY0;
                IF EQ JUMP newdigit;
space:          AX0=DM(time_on);
                AY0=DM(time_off);
                AR=AX0+AY0;
                DM(time_off)=AR;
                AR=0;
                DM(time_on)=AR;
                RTI;
newdigit:       AY0=h#000F;
                AR=AX0 AND AY0;
                SR=LSHIFT AR BY 1 (HI);
                AY0=^digits;
                AR=SR1+AY0;
                I1=AR;
                AX0=DM(I1,M0);      {look up row freq}
                DM(hertz1)=AX0;
                AX0=DM(I1,M0);      {look up col freq}
                DM(hertz2)=AX0;
                RTI;
```

```
reset:          SI=0;
                DM(sum1)=SI;
                DM(sum2)=SI;

                SI=DM(sign_dura_ms);
                SR=ASHIFT SI BY 3 (HI);
                AY0=SR1;
                AR=AY0-1;
                DM(time_on)=AR;

                SI=DM(interdigit_ms);
                SR=ASHIFT SI BY 3 (HI);
                AY0=SR1;
                AR=AY0-1;
                DM(time_off)=AR;
                RTS;

next_state:     I0=^dial_list;
                CALL reset;
                AY0=DM(state);
                AR=AY0+1;
                DM(state)=AR;
                AY0=3;
                AR=AR-AY0;           {mod 3, no state 3 exists}
                IF NE RTI;
                AR=0;
                DM(state)=AR;
                RTI;

.ENDMOD;
```

example to a pushbutton) is received. Also, in state 2 when a "stop" control word is read out of the dialing list, the machine jumps back to state 0. Output from the encoder is sent via an I/O port to a D/A converter and is also logarithmically compressed and send to a codec.

Listing 14.1 DTMF Encoder Program

14.7.2 DTMF Decoder Listing

The code below is the multi-channel DTMF decoder described earlier in this chapter. The six channels are labeled channel A, channel B, ..., channel F. Channels can be added or removed by modifying sections identified by the comment "edit this for more channels."

Input samples are assumed to come from a μ-law codec, and output codes are sent to a D/A converter via an I/O port. DTMF decoding is done in two major tasks. The first task solves sixteen Goertzel algorithms to calculate the magnitudes squared of tone frequencies present in the input signal, then the second task tests the frequency results to determine if the tones detected constitute a valid DTMF digit. The first task spans N+1

```
{                                                                        }
{  Multi Channel DTMF Decoder (six channels shown here)                  }
{                                                                        }
{  INPUT:   one voiceband telephone codec per channel                    }
{  OUTPUT:  DTMF digits are detected within each channel, with a         }
{           corresponding hexadecimal code written to that channel's     }
{           output port (D/A converter) as an activity monitor           }
{                                                                        }

.MODULE/RAM/ABS=0                 Multi_Channel_DTMF_Decoder;

.CONST     channels = 6;                    {must be 2 or more}
           {edit this for more channels}
.CONST     channels_x_32 = 192;
           {edit this for more channels}
.CONST     baddigitcode       = h#FFFF;   {output code for non-digit}
.CONST     pass_posttests     = 0;
.CONST     fail_minsig        = 1;
.CONST     fail_relpeak       = 2;
.CONST     fail_twist         = 3;
.CONST     fail_2ndharm       = 4;
```

```
{ === housekeeping variables === }
{16.0 fixed-point integers}
.VAR      count201;            {counts samples 0 to 200}
.VAR      count4;              {counts samples 201 to 204}

{1.15 fixed-point fractions}
.VAR      min_tone_level[8];     {min “tone-present” mnsqr level}
.VAR      max_notone_level[8];   {max “tone-not-present” mnsqr level}
.VAR      max_2ndharm_level;     {2nd harmonic must be LT this value}
.VAR      maxfortwist;           {quotient row/col must be GT this value}
.VAR      maxrevtwist;           {quotient col/row must be GT this value}
.VAR      mu_lookup_table[256];  {mu-expansion lookup table (scaled 8 bits)}
.VAR/CIRC in_samples[channels];  {linear input samples (scaled down 8 bits)}

.VAR/CIRC A_Q1Q2_buff[32],       {Goertzel feedback storage elements}
          B_Q1Q2_buff[32],
          C_Q1Q2_buff[32],
          D_Q1Q2_buff[32],
          E_Q1Q2_buff[32],
          F_Q1Q2_buff[32];
          {edit this for more channels}

.VAR      A_mnsqr[8],            {1.15 Goertzel result values}
          B_mnsqr[8],
          C_mnsqr[8],
          D_mnsqr[8],
          E_mnsqr[8],
          F_mnsqr[8];
          {edit this for more channels}

.VAR/PM
/RAM/CIRC coefs[16];            {2.14 Goertzel coefs}

{ === individual channel variables === }
.VAR      A_maxrowval;          {1.15 value of max row frequency}
.VAR      A_maxcolval;          {1.15 value of max col frequency}
.VAR      A_whichrow;           {0,1,2,3,4 = invalid, row1, row2, row3, row4}
.VAR      A_whichcol;           {0,1,2,3,4 = invalid, col1, col2, col3, col4}
.VAR      A_fortwistflag;       {1 = forward twist}
.VAR      A_revtwistflag;       {1 = reverse twist}
.VAR      A_twistval;           {1.15 quotient row/col or col/row}
.VAR      A_rowharm;            {1.15 value of row 2nd harmonic}
.VAR      A_colharm;            {1.15 value of col 2nd harmonic}
.VAR      A_digit_history[2];   {stores last 2 output codes}
.VAR      A_failurecode;        {see .CONST definitions above}
```

(listing continues on next page)

```
.VAR        B_maxrowval;
.VAR        B_maxcolval;
.VAR        B_whichrow;
.VAR        B_whichcol;
.VAR        B_fortwistflag;
.VAR        B_revtwistflag;
.VAR        B_twistval;
.VAR        B_rowharm;
.VAR        B_colharm;
.VAR        B_digit_history[2];
.VAR        B_failurecode;

.VAR        C_maxrowval;
.VAR        C_maxcolval;
.VAR        C_whichrow;
.VAR        C_whichcol;
.VAR        C_fortwistflag;
.VAR        C_revtwistflag;
.VAR        C_twistval;
.VAR        C_rowharm;
.VAR        C_colharm;
.VAR        C_digit_history[2];
.VAR        C_failurecode;

.VAR        D_maxrowval;
.VAR        D_maxcolval;
.VAR        D_whichrow;
.VAR        D_whichcol;
.VAR        D_fortwistflag;
.VAR        D_revtwistflag;
.VAR        D_twistval;
.VAR        D_rowharm;
.VAR        D_colharm;
.VAR        D_digit_history[2];
.VAR        D_failurecode;

.VAR        E_maxrowval;
.VAR        E_maxcolval;
.VAR        E_whichrow;
.VAR        E_whichcol;
.VAR        E_fortwistflag;
.VAR        E_revtwistflag;
.VAR        E_twistval;
.VAR        E_rowharm;
```

```
.VAR      E_colharm;
.VAR      E_digit_history[2];
.VAR      E_failurecode;

.VAR      F_maxrowval;
.VAR      F_maxcolval;
.VAR      F_whichrow;
.VAR      F_whichcol;
.VAR      F_fortwistflag;
.VAR      F_revtwistflag;
.VAR      F_twistval;
.VAR      F_rowharm;
.VAR      F_colharm;
.VAR      F_digit_history[2];
.VAR      F_failurecode;
{edit this for more channels}

{ individual channel I/O ports }
.PORT     A_codec;
.PORT     A_dac;

.PORT     B_codec;     {telephone audio 8-bit parallel codec input}
.PORT     B_dac;       {monitor decoder output with voltage level}

.PORT     C_codec;
.PORT     C_dac;

.PORT     D_codec;
.PORT     D_dac;

.PORT     E_codec;
.PORT     E_dac;

.PORT     F_codec;
.PORT     F_dac;
{edit this for more channels}
```

(listing continues on next page)

```
{ variable initializations }
.INIT      coefs[00]: h#6D0200, h#68B200, h#63FD00, h#5EE700;
.INIT      coefs[04]: h#4A7100, h#409100, h#329100, h#23CE00;
.INIT      coefs[08]: h#3ABC00, h#2C1700, h#1CC300, h#0CFB00;
.INIT      coefs[12]: h#D5CB00, h#C00000, h#A97700, h#94D000;

.INIT      mu_lookup_table:        < MU255.Q8 >;
.INIT      min_tone_level:
   h#0003,h#0003,h#0003,h#0003,h#0003,h#0003,h#0003,h#0003;

.INIT      max_notone_level:
   h#0002,h#0002,h#0002,h#0002,h#0002,h#0002,h#0002,h#0002;

.INIT      max_2ndharm_level:      h#0100;
.INIT      maxfortwist:            h#32F5;
.INIT      maxrevtwist:            h#1449;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  8-bit mu-law sample read from specified codec, then converted       }
{  to linear (1.15 scaled down 8 bits) via look-up table               }
{                                                                      }
{  INPUT:   channel codec to read                                      }
{  OUTPUT:  scaled linear sample in "in_samples" circular buffer       }
{                                                                      }
.MACRO      get_sample( %0 );   {make sure I3,M0,L3,M4,L6 initialized}

            AY0=DM(%0);         {read codec, mu-law data}
            AF=AX0 AND AY0;     {make sure AX0 initialized to h#00FF}
            AR=AX1+AF;          {make sure AX1 initialized to LUT base}
            I6=AR;
            SI=DM(I6,M4);       {look-up scaled, linear value}
            DM(I3,M0)=SI;       {store input sample}
.ENDMACRO;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  pick largest row and col freq values                                }
{                                                                      }
{  INPUT:   pointer to top of channel's mnsqr buffer                   }
{  OUTPUT:  largest row and col values and their indexes               }
{                                                                      }
.MACRO      maxrowcol( %0, %1, %2, %3, %4 );
            {^mnsqr, maxrowval, whichrow, maxcolval, whichcol}
.LOCAL      findmaxrow;
.LOCAL      findmaxcol;
```

```
            I2=%0;

            AX1=0;      {set up variables in case nothing found}
            AY1=0;      {set up variables in case nothing found}
            AY0=0;      {initialize BIGGEST-VALUE-SO-FAR to zero}
            CNTR=4;
            DO findmaxrow UNTIL CE;
                        AX0=DM(I2,M0); {read CURRENT-MNSQR-VALUE}
                        AR=AX0-AY0;    {compare to BIGGEST-VALUE-SO-FAR}
                        IF LE JUMP findmaxrow;
                        AX1=5;
                        AY0=AX0;       {if CURRENT is bigger, store value}
                        AY1=CNTR;      {if CURRENT is bigger, store index}
findmaxrow:             NOP;

            DM(%1)=AY0;     {store the largest mnsqr value}
            AR=AX1-AY1;
            DM(%2)=AR;      {store index of biggest row (1,2,3,4)}

            AX1=0;
            AY1=0;
            AY0=0;
            CNTR=4;
            DO findmaxcol UNTIL CE;
                        AX0=DM(I2,M0);
                        AR=AX0-AY0;
                        IF LE JUMP findmaxcol;
                        AX1=5;
                        AY0=AX0;
                        AY1=CNTR;
findmaxcol:             NOP;
            DM(%3)=AY0;
            AR=AX1-AY1;
            DM(%4)=AR;
.ENDMACRO;
```

(listing continues on next page)

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  checks whether selected freqs are GT minimum signal power level     }
{                                                                       }
{  INPUT:      index of detected row and col tones                     }
{              value of detected row and col tones                     }
{  OUTPUT:     failurecode set if test fails                           }
{                                                                       }
.MACRO         minsiglevel( %0, %1, %2, %3, %4 );
               {whichrow, maxrowval, whichcol, maxcolval, failurecode}
.LOCAL         failsiglevel;
.LOCAL         done;

               AX0=^min_tone_level;
               AY0=DM(%0);
               AF=AY0-1;
               AR=AX0+AF;
               I5=AR;      {I5 points to ^min_tone_level+whichrow-1}
               AX1=DM(%1);
               AY1=DM(I5,M4);
               AR=AX1-AY1;
               IF LT JUMP failsiglevel;

               AY0=3;
               AR=AX0+AY0;
               AY0=DM(%2);
               AR=AR+AY0;
               I5=AR;      {I5 points to ^min_tone_level+4+whichcol-1}
               AX1=DM(%3);
               AY1=DM(I5,M4);
               AR=AX1-AY1;
               IF GE JUMP done;

failsiglevel:  AX0=0;
               DM(%0)=AX0;
               DM(%2)=AX0;
               AY0=fail_minsig;
               DM(%4)=AY0;

done:          NOP;
.ENDMACRO;
```

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  verify that only one valid row freq and col freq are present        }
{                                                                       }
{  INPUT:      pointer to top of channel’s mnsqr buffer                 }
{              index of detected row and col tones                      }
{  OUTPUT:     failurecode set if test fails                            }
{                                                                       }
.MACRO         no_other_peaks( %0, %1, %2, %3 );
               {^mnsqr, whichrow, whichcol, failurecode}
.LOCAL         looper;
.LOCAL         failrelpeak;
.LOCAL         done;

               I2=%0;
               I6=^max_notone_level;
               AF=PASS 0;
               CNTR=8;
               DO looper UNTIL CE;
                        AX0=DM(I6,M4);
                        AY0=DM(I2,M0);
                        AR=AX0-AY0;
looper:                 IF LT AF=AF+1;
               AX1=2;               {see if only 2 tones are over their}
               AR=AX1-AF;           {max notone level thresholds}
               IF EQ JUMP done;
failrelpeak:   AX0=0;               {clear whichrow,col}
               DM(%1)=AX0;
               DM(%2)=AX0;
               AY0=fail_relpeak;
               DM(%3)=AY0;          {set failurecode}
done:          NOP;
.ENDMACRO;
```

(listing continues on next page)

```
{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  checks difference between row tone level and col tone level (twist) }
{                                                                       }
{  INPUT:   index of detected row and col tones                         }
{           value of detected row and col tones                         }
{  OUTPUT:  forward twist flag or reverse twist flag set                }
{           twist value(row/col [fwd] or col/row [rev])                 }
{           failurecode set if test fails                               }
{                                                                       }
.MACRO      twisttests( %0, %1, %2, %3, %4, %5, %6, %7 );
            {maxrowval, maxcolval, fortwistflag, revtwistflag,
            whichrow, whichcol, twistval, failurecode}
.LOCAL      standard;
.LOCAL      reverse;
.LOCAL      failtwist;
.LOCAL      done;

            MX0=0;
            MX1=1;
            AX0=DM(%0);
            AY0=DM(%1);
            AR=AX0-AY0;
            IF GT JUMP reverse;
standard:   DM(%2)=MX1;                   {column tone is stronger}
            DM(%3)=MX0;
            AX0=DM(%1);
            AY0=DM(%0);
            AF=PASS AY0;
            AY0=0;
            DIVS AF,AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DM(%6)=AY0;                   {AY0 = maxrowval / maxcolval}
            AX0=DM(maxfortwist);
            AR=AX0-AY0;
            IF GT JUMP failtwist;
            JUMP done;
reverse:    DM(%2)=MX0;
            DM(%3)=MX1;                   {row tone is stronger}
            AF=PASS AY0;
            AY0=0;
            DIVS AF,AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0; DIVQ AX0;
            DM(%6)=AY0;                   {AY0 = maxcolval / maxrowval}
            AX0=DM(maxrevtwist);
            AR=AX0-AY0;
            IF GT JUMP failtwist;
            JUMP done;
failtwist:  DM(%4)=MX0;
            DM(%5)=MX0;
            AY0=fail_twist;
            DM(%7)=AY0;
done:       NOP;
.ENDMACRO;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  checks energy levels in second harmonics of detected tones          }
{                                                                      }
{  INPUT:  pointer to top of channel’s Goertzel feedback buffer        }
{          index of detected row and col tones                         }
{          pointers to variables holding channel’s 2nd harmonic levels }
{  OUTPUT: value of channel’s row and col 2nd harmonic levels          }
{          failurecode set if test fails                               }
{                                                                      }
.MACRO         check2ndharm( %0, %1, %2, %3, %4, %5, %6, %7 );
               {^Q1Q2_buff, whichrow, ^rowharm, rowharm,
               whichcol, ^colharm, colharm, failurecode}
.LOCAL         fail2ndharm;
.LOCAL         done;

               AX0=%0;
               AY0=DM(%1);
               AR=AY0-1;                          {range: 1,2,3,4 => 0,1,2,3}
               SR=ASHIFT AR BY 1 (HI);            {range: 0,1,2,3 => 0,2,4,6}
               AY1=16;
               AF=SR1+AY1;
               AR=AX0+AF;
               I0=AR;         {I0 points to ^Q1Q2_buff+16+2*(whichrow-1)}
               AX0=^coefs;
               AF=AX0+AY0;
               AX0=7;
               AR=AX0+AF;
               I4=AR;         {I4 points to ^coefs+8+whichrow-1}
               I2=%2;         {I2 points to ^rowharm}
               CALL mnsqr;
               AX0=DM(%3);
               AY0=DM(max_2ndharm_level);
               AR=AX0-AY0;
               IF GT JUMP fail2ndharm;
```

(listing continues on next page)

```
               AX0=%0;
               AY0=DM(%4);
               AR=AY0-1;                        {range 1,2,3,4 => 0,1,2,3}
               SR=ASHIFT AR BY 1 (HI);          {range 0,1,2,3 => 0,2,4,6}
               AY1=24;
               AF=SR1+AY1;
               AR=AX0+AF;
               I0=AR;         {I0 points to ^Q1Q2_buff+16+8+2*(whichcol-
1)}
               AX0=^coefs;
               AF=AX0+AY0;
               AX0=11;
               AR=AX0+AF;
               I4=AR;         {I4 points to ^coefs+8+4+whichcol-1}
               I2=%5;         {I2 points to ^colharm}
               CALL mnsqr;
               AX0=DM(%6);
               AY0=DM(max_2ndharm_level);
               AR=AX0-AY0;
               IF LT JUMP done;
fail2ndharm:   AX0=0;
               DM(%1)=AX0;
               DM(%4)=AX0;
               AY0=fail_2ndharm;
               DM(%7)=AY0;
done:          NOP;
.ENDMACRO;

{%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%}
{  hexadecimal code for a given DTMF digit is generated  and output if }
{  necessary, digit_history updated, failurecode cleared               }
{                                                                       }
{  INPUT:   index of detected row and col tones                        }
{           failurecode                                                 }
{  OUTPUT:  digit_history updated with latest hex output code          }
{           hex output code written to output port if both:            }
{           (1) the current code is the same as the previous code      }
{           (2) but different from the one before that                 }
{           failurecode cleared for next DTMF decode operation         }
{                                                                       }
.MACRO      outputcode( %0, %1, %2, %3, %4 );
            {whichrow, whichcol, digit_history, failurecode, dac}
.LOCAL      checkfailures;
.LOCAL      digitdetected;
.LOCAL      nodigit;
.LOCAL      readlist;
.LOCAL      pushlist;
```

```
checkfailures: AY0=DM(%3);
               AR=PASS AY0;
               IF NE JUMP nodigit;
digitdetected: AY0=DM(%1);
               AR=AY0-1;
               SR=LSHIFT AR BY 12 (HI);
               AY0=DM(%0);
               AR=AY0-1;
               SR=SR OR LSHIFT AR BY 14 (HI);
               JUMP readlist;
nodigit:       SR1=baddigitcode;
readlist:      AY0=DM(%2);
               AY1=DM(%2+1);
               AR=SR1-AY1;
               IF EQ JUMP pushlist;
               AR=SR1-AY0;
               IF NE JUMP pushlist;
               DM(%4)=SR1;
pushlist:      DM(%2+1)=AY0;
               DM(%2)=SR1;
               AY0=pass_posttests;
               DM(%3)=AY0;
.ENDMACRO;

{————————————————————————————————————————————————————————}
{—————————— M A I N   C O D E ———————————————————————————}
{————————————————————————————————————————————————————————}

            RTI; RTI; RTI; JUMP sample;

            CALL setup;
            CALL restart;
            IMASK=b#1000;
here:       JUMP here;

{——————————————————————————————————————————————————————————}
{——— I N T E R R U P T   S E R V I C E   R O U T I N E ———}
{——————————————————————————————————————————————————————————}

sample:     get_sample( A_codec );
            get_sample( B_codec );
            get_sample( C_codec );
            get_sample( D_codec );
            get_sample( E_codec );
            get_sample( F_codec );
            {edit this for more channels}
```

(listing continues on next page)

```
dec201: AY0=DM(count201);
        AR=AY0-1;
        DM(count201)=AR;
        IF LT JUMP dec4;

in201:  CNTR=channels;           {number of channels}
        DO chan201 UNTIL;
           AY1=DM(I3,M0);        {get sample for channel, AY1=1.15}
           CNTR=16;              {8 fundamentals, 8 2nd_harmonics per channel}
           DO freq201 UNTIL CE;
              MX0=DM(I0,M0), MY0=PM(I4,M4);   {get Q1,COEF Q1=1.15,COEF=2.14}
              MR=MX0*MY0(RND), AY0=DM(I0,M1); {mult,get Q2, MR=2.30, Q2=1.15}
              SR=ASHIFT MR1 BY 1 (HI);        {change 2.30 to 1.15}
              AR=SR1-AY0;                     {Q1*COEF - Q2, AR=1.15}
              AR=AR+AY1;                      {Q1*COEF - Q2 + input, AR=1.15}
              DM(I0,M0)=AR;                   {result = new Q1}
freq201:   DM(I0,M0)=MX0;                     {old Q1 = new Q2 }
chan201:NOP;  {do next channel}
        RTI;

dec4:   AY0=DM(count4);
        AR=AY0-1;
        DM(count4)=AR;
        IF LT JUMP last;

in4:    CNTR=channels;           {number of channels}
        DO chan4 UNTIL CE;
           AY1=DM(I3,M0);
           CNTR=8;               {8 fundamentals only}
           DO freq4 UNTIL CE;
              MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MX0*MY0(RND), AY0=DM(I0,M1);
              SR=ASHIFT MR1 BY 1 (HI);
              AR=SR1-AY0;
              AR=AR+AY1;
              DM(I0,M0)=AR;
freq4:        DM(I0,M0)=MX0;
           MODIFY(I0,M2);        {skip 2nd harmonic Q1Q2s}
chan4:     MODIFY(I4,M5);        {skip 2nd harmonic COEFs}
        RTI;
```

```
last:    CNTR=channels;
         DO chanlast UNTIL CE;
            CNTR=8;
            DO freqlast UNTIL CE;
               CALL mnsqr;
freqlast:      NOP;
            MODIFY(I0,M2);        {skip 2nd harmonic Q1Q2s}
chanlast:   MODIFY(I4,M5);        {skip 2nd harmonic COEFs}

maxrowcol(^A_mnsqr,A_maxrowval,A_whichrow,A_maxcolval,A_whichcol);
maxrowcol(^B_mnsqr,B_maxrowval,B_whichrow,B_maxcolval,B_whichcol);
maxrowcol(^C_mnsqr,C_maxrowval,C_whichrow,C_maxcolval,C_whichcol);
maxrowcol(^D_mnsqr,D_maxrowval,D_whichrow,D_maxcolval,D_whichcol);
maxrowcol(^E_mnsqr,E_maxrowval,E_whichrow,E_maxcolval,E_whichcol);
maxrowcol(^F_mnsqr,F_maxrowval,F_whichrow,F_maxcolval,F_whichcol);
{edit this for more channels}

{————————— START OF DIGIT VALIDATION TESTS ——————————}

minsiglevel(A_whichrow,A_maxrowval,A_whichcol,A_maxcolval,A_failurecode);
minsiglevel(B_whichrow,B_maxrowval,B_whichcol,B_maxcolval,B_failurecode);
minsiglevel(C_whichrow,C_maxrowval,C_whichcol,C_maxcolval,C_failurecode);
minsiglevel(D_whichrow,D_maxrowval,D_whichcol,D_maxcolval,D_failurecode);
minsiglevel(E_whichrow,E_maxrowval,E_whichcol,E_maxcolval,E_failurecode);
minsiglevel(F_whichrow,F_maxrowval,F_whichcol,F_maxcolval,F_failurecode);
{edit this for more channels}

no_other_peaks(^A_mnsqr,A_whichrow,A_whichcol,A_failurecode);
no_other_peaks(^B_mnsqr,B_whichrow,B_whichcol,B_failurecode);
no_other_peaks(^C_mnsqr,C_whichrow,C_whichcol,C_failurecode);
no_other_peaks(^D_mnsqr,D_whichrow,D_whichcol,D_failurecode);
no_other_peaks(^E_mnsqr,E_whichrow,E_whichcol,E_failurecode);
no_other_peaks(^F_mnsqr,F_whichrow,F_whichcol,F_failurecode);
{edit this for more channels}

twisttests(A_maxrowval,A_maxcolval,A_fortwistflag,A_revtwistflag,
            A_whichrow,A_whichcol,A_twistval,A_failurecode);
twisttests(B_maxrowval,B_maxcolval,B_fortwistflag,B_revtwistflag,
            B_whichrow,B_whichcol,B_twistval,B_failurecode);
twisttests(C_maxrowval,C_maxcolval,C_fortwistflag,C_revtwistflag,
            C_whichrow,C_whichcol,C_twistval,C_failurecode);
twisttests(D_maxrowval,D_maxcolval,D_fortwistflag,D_revtwistflag,
            D_whichrow,D_whichcol,D_twistval,D_failurecode);
twisttests(E_maxrowval,E_maxcolval,E_fortwistflag,E_revtwistflag,
            E_whichrow,E_whichcol,E_twistval,E_failurecode);
twisttests(F_maxrowval,F_maxcolval,F_fortwistflag,F_revtwistflag,
```

(listing continues on next page)

```
            F_whichrow,F_whichcol,F_twistval,F_failurecode);
{edit this for more channels}

check2ndharm(^A_Q1Q2_buff,A_whichrow,^A_rowharm,A_rowharm,
                        A_whichcol,^A_colharm,A_colharm,A_failurecode);
check2ndharm(^B_Q1Q2_buff,B_whichrow,^B_rowharm,B_rowharm,
                        B_whichcol,^B_colharm,B_colharm,B_failurecode);
check2ndharm(^C_Q1Q2_buff,C_whichrow,^C_rowharm,C_rowharm,
                        C_whichcol,^C_colharm,C_colharm,C_failurecode);
check2ndharm(^D_Q1Q2_buff,D_whichrow,^D_rowharm,D_rowharm,
                        D_whichcol,^D_colharm,D_colharm,D_failurecode);
check2ndharm(^E_Q1Q2_buff,E_whichrow,^E_rowharm,E_rowharm,
                        E_whichcol,^E_colharm,E_colharm,E_failurecode);
check2ndharm(^F_Q1Q2_buff,F_whichrow,^F_rowharm,F_rowharm,
                        F_whichcol,^F_colharm,F_colharm,F_failurecode);
{edit this for more channels}

{——————————— END OF DIGIT VALIDATION TESTS —————————}

outputcode(A_whichrow,A_whichcol,A_digit_history,A_failurecode,A_dac);
outputcode(B_whichrow,B_whichcol,B_digit_history,B_failurecode,B_dac);
outputcode(C_whichrow,C_whichcol,C_digit_history,C_failurecode,C_dac);
outputcode(D_whichrow,D_whichcol,D_digit_history,D_failurecode,D_dac);
outputcode(E_whichrow,E_whichcol,E_digit_history,E_failurecode,E_dac);
outputcode(F_whichrow,F_whichcol,F_digit_history,F_failurecode,F_dac);
{edit this for more channels}

   CALL restart;
   RTI;

{—————————————————————————————————————————————————————}
{——————————— S U B R O U T I N E S ———————————————————}
{—————————————————————————————————————————————————————}

{%%%%%%%%%%% O N E   T I M E   O N L Y   S E T U P %%%%%%%%%%%%%}
{  initializes digit_history lists, M and L registers in        }
{  address generators, and sets ICNTL to edge-sensitive         }

setup:      SI=baddigitcode;
            DM(A_digit_history)=SI;          DM(A_digit_history+1)=SI;
            DM(B_digit_history)=SI;          DM(B_digit_history+1)=SI;
            DM(C_digit_history)=SI;          DM(C_digit_history+1)=SI;
            DM(D_digit_history)=SI;          DM(D_digit_history+1)=SI;
            DM(E_digit_history)=SI;          DM(E_digit_history+1)=SI;
            DM(F_digit_history)=SI;          DM(F_digit_history+1)=SI;
            {edit this for more channels}
            L0 = channels_x_32;
            L1 =  0;
            L2 =  0;
            L3 = channels;
            L4 = 16;
            L5 =  0;
            L6 =  0;
            L7 =  0;

            M0 =  1;
            M1 = -1;
            M2 = 16;
            M4 =  1;
            M5 =  8;

            ICNTL=b#01111;
            RTS;

{%%%%%%%%%%%%% E V E R Y   T I M E   S E T U P %%%%%%%%%%%%%%%%%}
{  resets pointers to top of buffers, resets counter values,    }
{  clears Goertzel feedback buffers to zero, etc                }

restart:    I0=^A_Q1Q2_buff;
            CNTR=channels_x_32;
            DO zloop UNTIL CE;
zloop:                  DM(I0,M0)=0;
            I2=^A_mnsqr;
            I3=^in_samples;
            I4=^coefs;
            AX0=201;    DM(count201)=AX0;
            AX0=4;      DM(count4)=AX0;
            AX0=h#00FF;
            AX1=^mu_lookup_table;
            RTS;
```

```
{%%%%%%%%%% S Q U A R E D   M A G N I T U D E   C A L C %%%%%%%%%%%%%%%%}
{  calculates squared magnitude (mnsqr) from Goertzel feedback results  }

mnsqr:  MX0=DM(I0,M0);                      {get two copies of Q1, 1.15}
        MY0=MX0;
        MX1=DM(I0,M0);                      {get two copies of Q2, 1.15}
        MY1=MX1;
        AR=PM(I4,M4);                                    {get COEF, 2.14}
        MR=0;
        MF=MX0*MY1(RND); {Q1*Q2, 1.15}
        MR=MR-AR*MF(RND);                             {-Q1*Q2*COEF, 2.14}
        SR=ASHIFT MR1 BY 1 (HI);       {2.14 -> 1.15 format conv., 1.15}
        MR=0;
        MR1=SR1;
        MR=MR+MX0*MY0(SS);                     {Q1*Q1 + -Q1*Q2*COEF, 1.15}
        MR=MR+MX1*MY1(RND);          {Q1*Q1 + Q2*Q2 + -Q1*Q2*COEF, 1.15}
        DM(I2,M0)=MR1;                     {store in mnsqr buffer, 1.15}
        RTS;

.ENDMOD;
```

processor interrupts. The length of time required by the processor for the second task may span the next few interrupts (during which time input samples are ignored), depending on the tones detected.

Listing 14.2 DTMF Decoder Program