Digital Tone Detection 8

8.1 OVERVIEW

ADSP-2100 Family DSPs are well suited for applications that detect sinusoidal tones. These applications include telephone signaling, remotely controlled equipment, test instruments for tone based systems, and tone-encoded data transmission.

One of the most common examples of tone detection is the touch-tone signaling standard used in telephones. This standard is called DTMF, or dual-tone, multi-frequency signaling. Since DTMF is an in-band signaling system (superimposed on the voice channel), it rejects interference from the simultaneously present voice frequencies. Telephones systems also use other standards, for example, trunk switching circuits may use out-of-band MF (multi-frequency) signaling, while other switching equipment may use single-tone signaling.

Another common application for tone detection is remotely controlled equipment, such as a remotely-piloted drone aircraft. These applications pass servo instructions to the drone aircraft by radio control. These instructions are binary numbers that are coded in frequency. Each binary digit is assigned a frequency. The receiver reconstructs binary numbers by detecting the presence (logic “1”) or the absence (logic “0”) of each possible tone.

Digital tone detection applications usually have fast execution speeds and require minimum memory storage. You can take advantage of these features by coding tone detection as a sub task of a larger, single-chip DSP application, or by using a single DSP to simultaneously handle tone processing for many independent channels.

DTMF tone detection and generation is covered in Chapter 14, of *Digital Signal Processing Applications Using the ADSP-2100 Family*, Volume 1. Refer to that chapter for more details on the Goertzel method of tone detection and validation, as well as precision sine wave generation using fast polynomial expansions.

8 Digital Tone Detection

8.2 IMPLEMENTATION

This section outlines the steps you can use to implement the tone detection subroutines included at the end of this chapter.

8.2.1 Choosing A Sampling Frequency

Sometimes your application dictates the sampling frequency. This is true for telephone band applications where the local telephone administration specifies a sampling frequency (8000 Hz, for example). For applications where you choose the frequency, remember that the Nyquist theory states that the minimum sampling frequency must be at least twice the frequency of the highest frequency you want to process.

Once you have a list of frequencies, to help you select the best sampling frequency, factor each frequency into its prime factors. Listings 8.1 and 8.2 contain two C programs (FACTOR.C and PRIMES.C) to help you. Once each frequency is broken down into its constituent prime factors, pick out the prime factors that are most common to the greatest number of frequencies, then multiply the prime factors together. Call the resulting product "A". The best sampling frequency is an integer multiple of "A" that is greater than or equal to twice the highest input frequency of interest.

Listing 8.3 (BESTFS.C) is a C program that verifies if the chosen sampling frequency is the best fit. The program sweeps through the specified range of sampling frequencies and calculates the maximum squared mismatch error for the tone set. The mismatch error is how closely the tone of interest matches the integral subdivisions of the sampling frequency. Frequencies with many common prime factors will match more precisely.

For a given tone set, the mismatch error-squared is calculated for each individual tone. The largest mismatch in the tone set is chosen as the maximum error-squared value for the tone set at that sampling frequency. This is the term that should be minimized. The following equation describes the mismatch error-squared:

$$E^2(i) = \left[\left(\frac{f_{sample}}{f_{tone}(i)}\right)\right] - n\left[\left(\frac{f_{sample}}{f_{tone}(i)}\right)\right]^2$$

where $E^2(i)$ is the mismatch error-squared and n is the nearest integer.

BESTFS.C stores the resulting error values in a file called BESTFS.ERR and displays them on the terminal screen. To find the best sampling frequency,

sort the error file alphabetically using any sort utility. You may want to plot the error values before sorting them to graphically identify a minimum.

8.2.2 Picking The Best Value Of N For The Goertzel Iterations

The Goertzel algorithm operates on a sample-by-sample basis, like an IIR filter. After N iterations, or N samples received, the output value of this algorithm is the Goertzel output of interest. This output value is equivalent to what a single-frequency DFT calculates. You can think of the Goertzel algorithm as an IIR filter with an output that is sampled after every N samples. Consider the following parameters when you select a value for N:

- Leakage Loss
- Frequency Resolution
- Detection Time

8.2.2.1 Leakage Loss

The Goertzel algorithm has the same frequency characteristics as the DFT algorithm. In other words, if an N-point DFT is performed on a data sequence sampled at frequency F_S, the output frequency samples, sometimes called frequency bins, are equally spaced at F_S / N. If a tone is present that matches an integer multiple of F_S / N, it is completely contained in one of the output frequency bins. If however, the tone falls between the center of two adjacent frequency bins, the total energy is distributed among several neighboring frequency bins. This phenomenon is called spillage, or leakage. See Chapter 6, *One-Dimensional FFTs*, of *Digital Signal Processing Applications Using the ADSP-2100 Family*, Volume 1 for more information.

You can think of the frequency samples as if they were output frequency samples of an FFT or DFT calculation. Therefore, you can imagine that all frequency samples are present, even though the actual implementation only calculates the frequency samples of interest. As a result, spillage into neighboring bins appears to decrease the level present in the bin of interest. In a tone detection application, the missing energy (that, mathematically, is spilled into nearby bins) is never seen in the neighboring bins since the neighboring bin levels are not calculated.

Because of leakage, for a given sampling frequency and a given frequency to detect, some values of N show poor performance, some values have better performance, and some values perform optimally. In the optimal case, the tone to detect is an exact integer multiple of F_S / N, for example when $F_{tone}=k*F_S / N$. When decoding several frequencies, try to pick one value of N for all frequencies. This means that some frequencies will closely match $k*F_S / N$, and some will not. The reason you should try to use a single N value for multiple frequencies is because the valid Goertzel outputs are available after N samples are processed, so the valid output of all frequencies is available at the same time.

8.2.2.2 Frequency Resolution

Frequency resolution is the second consideration. Larger values for N provide better frequency resolution. If N is large, F_S / N (the individual frequency bin widths, or the spacing of the resulting frequency samples) is small. This means that the detector will reject more off-frequency tones, or resolve between two tones that are close together.

8.2.2.3 Detection Time

You must also consider detection time. When you choose a larger value for N, it takes longer for N samples to be received, and consequently, the time between valid Goertzel outputs is longer. This directly affects the speed at which the decoder detects the presence of a tone.

Listing 8.4, called BESTN.C, tests the "goodness of fit" for values of N within a specified range, given the sampling frequency. `BESTN.C` stores the results in a file called `BESTN.ERR` and displays them on the terminal screen. To find the best values for N, sort the error file alphabetically using any sort utility. The result is a list of the best values for N at the chosen sampling frequency in descending order of "goodness of fit."

8.2.2.4 Tone Detection Categories

Tone detection code falls into two categories:

- Symbol detection (applications such as DTMF)—More than one tone is detected, then tested for relative amplitudes, number of tones present, etc. to validate the presence of a symbol (made up tones).

- Independent, single, tone, presence detection—Any number of tones can be present, and the indication of a tone's presence is the only requirement. Tests, such as relative amplitudes and number of tones, are not necessary in this case.

Symbol detection follows the DTMF decoder described in Chapter 14 of Volume 1. That example only listens for two tones from a predetermined alphabet. By changing coefficients and post-testing thresholds, the symbol detector 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, US Army TA-314/PT, etc.

Of the two categories of tone detection code, single tone presence detection is simpler to implement. Most of the post-testing can be eliminated and replaced by simpler energy presence (threshold comparison) tests. Section 8.2.3.5 contains an example.

Both categories use the basic Goertzel algorithm for each tone. Voice rejection requires the monitoring of energy content at the intended tone's second harmonic. Slight changes must be made when switching from single-channel decoding to multiple-channel decoding. When decoding several channels, the input samples of all channels are stored in a circular buffer. The buffer length is equal to the number of channels. When you decode a single channel, the replace the circular buffer with a single data memory variable.

8.2.2.5 Tone Detection Example

This example was designed for single tone detection on the frequencies shown in Table 8.1. Using FACTOR.C, the prime factors of the frequencies of interest are also shown in Table 8.1. To choose a good sampling frequency, pick the prime factors that are common to most of the frequencies of interest. In this example, most of the frequencies have the prime factors 3, 3, 5, 5, and 7. Multiply them together (the product is 1575), and select the integer multiple of that product that is the next one higher than twice the highest input frequency of interest.

Frequencies Of Interest	*Prime Factors*
11025 Hz	3, 3, 5, 5, 5, 7
12600 Hz	2, 2, 2, 3, 3, 5, 5, 7
14175 Hz	3, 3, 3, 3, 5, 5, 7
15750 Hz	2, 3, 3, 5, 5, 5, 7
17325 Hz	3, 3, 5, 5, 7, 11
18900 Hz	2, 2, 3, 3, 3, 5, 5, 7
20475 Hz	3, 3, 5, 5, 7, 13
23175 Hz	3, 3, 5, 5, 103

Table 8.1 Sample Frequencies & Prime Factors

The highest frequency of interest is 23175 Hz, therefore sampling must occur at a minimum frequency of 46350 Hz. The smallest integer multiple of 1575 that is greater than 46350 is 47250. This example uses BESTFS.C to verify the choice (47250 Hz). You can verify the sampling frequency by testing all integer frequencies between 46350 Hz and 49000 Hz with the following syntax:

```
c:> bestfs 46350 49000 1 | sort | more
```

Table 8.2 is an example of the resulting output. You can see that 47250 Hz is not the best choice; within the range 46350–49000 Hz, the best sampling frequency is 48600 Hz.

```
*** scanning f_sample from 46350.000000 Hz to 49000.000000 Hz,
    stepping 1.000000 Hz ***
0.183674 = maxerrsq     (at f_sample = 48600.000000)
0.183719 = maxerrsq     (at f_sample = 48599.000000)
0.183734 = maxerrsq     (at f_sample = 48601.000000)
0.183764 = maxerrsq     (at f_sample = 48598.000000)
0.183794 = maxerrsq     (at f_sample = 48602.000000)
0.183810 = maxerrsq     (at f_sample = 48597.000000)
0.183855 = maxerrsq     (at f_sample = 48596.000000)
0.183855 = maxerrsq     (at f_sample = 48603.000000)
0.183900 = maxerrsq     (at f_sample = 48595.000000)
0.183915 = maxerrsq     (at f_sample = 48604.000000)
0.183946 = maxerrsq     (at f_sample = 48594.000000)
0.183976 = maxerrsq     (at f_sample = 48605.000000)
0.183991 = maxerrsq     (at f_sample = 48593.000000)
0.184036 = maxerrsq     (at f_sample = 48592.000000)
0.184036 = maxerrsq     (at f_sample = 48606.000000)
0.184082 = maxerrsq     (at f_sample = 48591.000000)
0.184097 = maxerrsq     (at f_sample = 48607.000000)
0.184127 = maxerrsq     (at f_sample = 48590.000000)
0.184158 = maxerrsq     (at f_sample = 48608.000000)
   .
   .     etc,
   .
0.249947 = maxerrsq     (at f_sample = 47249.000000)
0.249947 = maxerrsq     (at f_sample = 47251.000000)
0.250000 = maxerrsq     (at f_sample = 47250.000000)
```

Table 8.2 Sorted Sampling Frequencies (BESTFS.ERR)

Next, look for the best value for N. For this example, assume that you must detect tones within 10 ms, and you want a frequency resolution of approximately 100 Hz. You can use the following syntax to select N:

```
c:> bestn 46350 0 600 | sort | more
```

This example evaluates all values for N from N=0 through N=600, at a sampling frequency of 48600 Hz. The program calculates the maximum square error of all tones of interest and their respective closest integer multiples of F_S / N. The program also displays the associated detection time in milliseconds and frequency resolution width in Hertz. Table 8.3 is a sample of the resulting output. The results show that several values for N are acceptable. If N=463, or N=493, it fits the requirements fairly well. If N=432, the tone set matches optimally with the sampling frequency of 48600 Hz, the detection time is less than 9 ms, but the frequency resolution is wider than the 100 Hz. For this example, let N=463.

```
*** scanning N from o to 600, where f_sample = 48600.000000 Hz ***
0.000000 = maxerrsq (N=      0) detect=    0.000 ms   resolu= Infinity Hz
0.000000 = maxerrsq (N=    216) detect=    4.444 ms   resolu=  225.000 Hz
0.000000 = maxerrsq (N=    432) detect=    8.889 ms   resolu=  112.500 Hz
0.047346 = maxerrsq (N=     31) detect=    0.638 ms   resolu= 1567.742 Hz
0.047346 = maxerrsq (N=    185) detect=    3.807 ms   resolu=  262.703 Hz
0.047346 = maxerrsq (N=    247) detect=    5.082 ms   resolu=  196.761 Hz
0.047346 = maxerrsq (N=    401) detect=    8.251 ms   resolu=  121.197 Hz
0.047346 = maxerrsq (N=    463) detect=    9.527 ms   resolu=  104.968 Hz
0.057955 = maxerrsq (N=    308) detect=    6.337 ms   resolu=  157.792 Hz
0.057955 = maxerrsq (N=    340) detect=    6.996 ms   resolu=  142.941 Hz
0.057955 = maxerrsq (N=    524) detect=   10.782 ms   resolu=   92.748 Hz
0.057955 = maxerrsq (N=    556) detect=   11.440 ms   resolu=   87.410 Hz
0.057957 = maxerrsq (N=     92) detect=    1.893 ms   resolu=  528.261 Hz
0.057957 = maxerrsq (N=    124) detect=    2.551 ms   resolu=  391.935 Hz
0.090552 = maxerrsq (N=    371) detect=    7.634 ms   resolu=  130.997 Hz
0.090552 = maxerrsq (N=    493) detect=   10.144 ms   resolu=   98.580 Hz
0.090552 = maxerrsq (N=    587) detect=   12.078 ms   resolu=   82.794 Hz
0.090557 = maxerrsq (N=     61) detect=    1.255 ms   resolu=  796.721 Hz
0.090557 = maxerrsq (N=    155) detect=    3.189 ms   resolu=  313.548 Hz
0.090557 = maxerrsq (N=    277) detect=    5.700 ms   resolu=  175.451 Hz
0.105024 = maxerrsq (N=     34) detect=    0.700 ms   resolu= 1429.412 Hz
0.105024 = maxerrsq (N=    182) detect=    3.745 ms   resolu=  267.033 Hz
0.105024 = maxerrsq (N=    250) detect=    5.144 ms   resolu=  194.400 Hz
0.105029 = maxerrsq (N=    398) detect=    8.189 ms   resolu=  122.111 Hz
0.105029 = maxerrsq (N=    466) detect=    9.588 ms   resolu=  104.292 Hz
0.111108 = maxerrsq (N=    354) detect=    7.284 ms   resolu=  137.288 Hz
0.111108 = maxerrsq (N=    510) detect=   10.484 ms   resolu=   95.294 Hz
   .
   .     etc,
   .
0.250000 = maxerrsq (N=    585) detect=   12.037 ms   resolu=   83.077 Hz
0.250000 = maxerrsq (N=    588) detect=   12.099 ms   resolu=   82.653 Hz
0.250000 = maxerrsq (N=    594) detect=   12.222 ms   resolu=   81.818 Hz
```

Table 8.3 Sorted Values For N (BESTN.ERR)

Next, the Goertzel coefficients must be calculated. Use the following syntax to start COEFGEN.C (Listing 8.5):

```
c:> coefgen
N > 463
f_sample > 48600
```

The resulting output is shown in Table 8.4. The columns show each tone to detect with its associated k(flt), k(int), and k(err) values. The k(flt) value is the floating point value of N*(f_{tone} / f_{sample}); k(int) is the closest integer to k(flt). This integer is the index of the frequency bin for the closest match. If k(flt) and k(int) are equal, they are perfectly matched and there is no leakage loss occurs. Discrepancies between k(flt) and k(int) lead to leakage losses, and this difference is measured in the k(err) variable. The "goodness of fit" of the N value is judged by the largest squared k(err) value of all the individual f_{tone} values.

```
N=463.000000
fs=48600.000000

f_tone[0]=  11025.00 Hz
                k(flt)=105.032410
                k(int)=105
                k(err)= +0.032410
                coef(flt)= +0.290734 coef(2.14 hex)=0x129B
f_tone[1]=  12600.00 Hz
                k(flt)=120.037041
                k(int)=120
                k(err)= +0.037041
                coef(flt)= -0.115286 coef(2.14 hex)=0xF89F
f_tone[2]=  14175.00 Hz
                k(flt)=135.041672
                k(int)=135
                k(err)= +0.041672
                coef(flt)= -0.516546 coef(2.14 hex)=0xDEF1
f_tone[3]=  15750.00 Hz
                k(flt)=150.046295
                k(int)=150
                k(err)= +0.046295
                coef(flt)= -0.896475 coef(2.14 hex)=0xC6A0
f_tone[4]=  17325.00 Hz
                k(flt)=165.050919
                k(int)=165
                k(err)= +0.050919
                coef(flt)= -1.239387 coef(2.14 hex)=0xB0AE
```

```
f_tone[5]=  18900.00 Hz
                k(flt)=180.055557
                k(int)=180
                k(err)= +0.055557
                coef(flt)= -1.531119 coef(2.14 hex)=0x9E02
f_tone[6]=  20475.00 Hz
                k(flt)=195.060181
                k(int)=195
                k(err)= +0.060181
                coef(flt)= -1.759627 coef(2.14 hex)=0x8F62
f_tone[7]=  23175.00 Hz
                k(flt)=220.782410
                k(int)=221
                k(err)= +0.217590
                coef(flt)= -1.979731 coef(2.14 hex)=0x814C
```

Table 8.4 Goertzel Coefficients

The Goertzel algorithm uses a single, real coefficient for each f_{tone} to detect. That coefficient is listed in a column with the hexadecimal equivalent for the ADSP-2100 family software. Since the equation for the coefficient is $2*\cos(2\pi*k(int)/N)$, the coefficient values are in the range $-2.0 \leq coef \leq 2.0$. For this reason, the coefficients are interpreted by the Goertzel algorithm is 2.14 format.

Listing 8.6 is an example of ADSP-2100 tone detection code. It establishes a routine that waits for interrupts. For each interrupt, the sample is counted and fed into the Goertzel feedback loop. When the sample count reaches N, the Goertzel feedforward instructions are executed, and a frequency-domain sample is calculated. Since N is the same for all tones being detected, all the results are available during the interrupt period. The following software checks the energy level found in each frequency bin of interest, and compares it to the predefined threshold. If the threshold is exceeded, a routine indicates the presence of that particular tone.

8.3 BENCHMARKS FOR THE EXAMPLE PROGRAM

The example detailed above detects the presence of energy in eight frequencies. The levels are threshold tested, and a binary number is output as a result every N input samples. Benchmarks will vary as specific applications deviate from this example, although this example is fundamental enough to demonstrate how you can evaluate your own benchmarks. Table 8.5 shows typical benchmark performance of the ADSP-2100A in this example application, as well as processor loading values for a similar example sampled at 8 kHz instead of the 48600 Hz.

8 Digital Tone Detection

Memory Usage:	*PM RAM*	*DM RAM*	
	102 Locations	302 Locations	
DSP	*Processor Speed*	*Number of Cycles*	*Execution Time*
ADSP-2101/2111	20 MHz	75	3.75 µs
ADSP-2171	33 MHz	75	2.25 µs

Table 8.5 Typical Benchmark Performance

Monitoring more frequencies requires more computation time. Having a faster sampling rate reduces the amount of time available for Goertzel feedback iterations between interrupts. The number of instructions available between interrupts is equal to the sampling period divided by the instruction cycle time.

The above example assumes that the ADSP-2100A is executing at a 12.5 MHz instruction rate. Dividing 12.5 MHz by the sampling frequency of 48600 Hz yields 257 available instructions between interrupts to maintain real-time processing.

Choosing different values of N has no impact on the computational benchmark. As long as the Goertzel feedback iterations can be performed between input samples, the Goertzel algorithm works. Large values of N mean that it takes the algorithm longer to generate an output value after N input iterations.

The example code uses very little memory. A major portion of the data memory storage (256 places out of 302 total) is taken by a lookup table for the µ-law PCM conversion. The ADSP-2101/2 has this function built into its serial ports and it does not usurp data memory storage.

Program memory is limited to 94 total instructions with 8 coefficients. All program memory and data memory requirements are easily fulfilled by the on-chip memory of the ADSP-2100 Family processors, leaving the remaining on-chip memory space for other DSP functions.

8.4 LISTINGS

This sections contains the listing for this chapter.

```
#include <stdio.h>

int prime[2000], factor[2000];
FILE *fp;

int isaprime( a, howmany )
int a, howmany;
{
   int i;
   for (i=0; i<howmany; i++)
         if (a==prime[i]) return(1);
   return(0);
}

int findaprime( a, howmany )
int a, howmany;
{
   int i;
   for (i=1; i<howmany; i++)
         if ((a%prime[i])==0) return(prime[i]);
   return(0);
}

main(argc,argv)
int argc; char **argv;
{
   int i, j, orig, freq, num, maxprimes;

   if (argc!=2)
         {
         printf("number to factor> ");
         scanf("%d",&freq);
         }
   else
         sscanf(argv[1],"%d",&freq);

   fp=fopen("primes.dat","r");
   if (fp==NULL) {printf("\nerror opening primes.dat\n");return(-1);}
   i=0;
   while (!feof(fp)) fscanf(fp,"%d",&prime[i++]);
   maxprimes=i-1;
   printf("\n%d primes read\n", maxprimes);
   fclose(fp);
```

(listing continues on next page)

```
   orig=freq;
   i=0;
   while(1)
         {
         if (isaprime(freq,maxprimes)==1) { factor[i++]=freq; break; }
         num=findaprime(freq,maxprimes);
         freq=freq/num;
         factor[i++]=num;
         }
   printf("\n %d factored out = ", orig);
   for (j=0; j<i; j++) printf("%d ", factor[j]);
   printf("\n");
}
```

Listing 8.1 Prime Factors Routine (FACTOR.C)

```
#include <stdio.h>

int fact, num, k, prime[1000];
FILE *fp;

main()
{
   fp=fopen("primes.dat","w");
   fprintf(fp,"%4d\n",1);
   fprintf(fp,"%4d\n",2);
   num=2;
   k=0;
   while(num<=2000)
         {
         fact=num-1;
         while((num/fact*fact)!=num)
            {
            –fact;
            if (fact<=1)
                  {
                  prime[k] = num;
                  fprintf(fp,"%4d\n",prime[k]);
                  k++;
                  }
            }
         printf("\r%d",num);
         ++num;
         }
   printf("\nDONE.\n");
}
```

Listing 8.2 Prime Numbers Routine (PRIMES.C)

```
#include <string.h>
#include <stdio.h>
#include “tones.def”

int round( x )
float x;
{
   if (x>0) return ( (int) (x+0.5) );
   else if (x<0) return ( (int) (x-0.5) );
   else if (x==0) return ( 0 );
   else printf(“\7bad data in round() function!”);
   return(-1);
}

char f1name[]=”bestfs.err”;
char f2name[]=”bestfs.fs”;

main(argc,argv)
int argc;
char **argv;
{
   int i, kint;
   float f_min, f_max, f_incr, f_sample;
   float kflt, maxerrsqr, errsqr;
   FILE *f1, *f2;
   if (argc!=4)
         {
         printf(“\n\7usage: %s <f_min> <f_max> <f_incr>\n”,argv[0]);
         printf(“\n(where f_min, f_max, f_incr are real numbers)\n”);
         return(-1);
         }
   sscanf(argv[1],”%f”,&f_min);
   sscanf(argv[2],”%f”,&f_max);
   sscanf(argv[3],”%f”,&f_incr);
   printf(“*** scanning f_sample from %f Hz to %f Hz,
           stepping %f Hz ***\n”,f_min,f_max,f_incr);
   if (f_min>=f_max)
         {
         printf(“f_min>=f_max!”);
         return(-1);
         }
   else if ((f_max<f_min+f_incr)||(f_incr<=0))
         {
         printf(“bad f_incr value!”);
         return(-1);
         }
```

(listing continues on next page)

```
   else for (i=0; i<HOW_MANY_TONES; i++)
         {
         if (f_min<(2*f_tone[i]))
            {
            printf("\n\7Nyquist violation:\nf_tone=%f
                    at f_sampling=%f\n",f_tone[i],f_min);
            return(-1);
            }
         }
   f1=fopen(f1name,"w"); if (f1==NULL) printf("\nerror opening %s\n",f1);
   f2=fopen(f2name,"w"); if (f2==NULL) printf("\nerror opening %s\n",f2);
   for (f_sample=f_min; f_sample<=f_max; f_sample=f_sample+f_incr)
         {
         maxerrsqr=0;
         for (i=0; i<HOW_MANY_TONES; i++)
            {
            kflt=f_sample/f_tone[i];
            kint=round(kflt);
            errsqr=(kflt-(float)(kint))*(kflt-(float)(kint));
            if (errsqr>maxerrsqr) maxerrsqr=errsqr;
            }
         printf("\n%f = maxerrsqr (at f_sample = %f)",
             maxerrsqr, f_sample);
         if ((f1)&&(f2))
            {
            fprintf(f1,"%f\n",maxerrsqr);
            fprintf(f2,"%f\n",f_sample);
            }
         }
   printf("\nyou may want to pipe output to sort utility");
   printf("\nor plot %s vs %s\n",f1name,f2name);
   fclose(f1);  fclose(f2);
}
```

Listing 8.3 Best Sampling Frequency Routine (BESTFS.C)

```
#include <string.h>
#include <stdio.h>
#include “tones.def”

int round( x )
float x;
{
   if (x>0) return ( (int) (x+0.5) );
   else if (x<0) return ( (int) (x-0.5) );
   else if (x==0) return ( 0 );
   else printf(“\7bad data in round() function!”);
   return(-1);
}

char f1name[]=”bestn.err”;
char f2name[]=”bestn.N”;

main(argc,argv)
int argc;
char **argv;
{
   int i, N, minN, maxN, kint;
   float f_sample, kflt, errsqr, maxerrsqr, detect, binwidth;
   FILE *f1, *f2;

   if (argc!=4)
         {
         printf(“\nusage: %s <f_sample Hz [%%f] > <minN [%%d] > <maxN [%%d]
               >\n”,argv[0]);
         return(-1);
         }
   sscanf(argv[1],”%f”,&f_sample);
   sscanf(argv[2],”%d”,&minN);
   sscanf(argv[3],”%d”,&maxN);
   printf(“*** scanning N from %d to %d, where f_sample
          = %f Hz ***\n”,minN,maxN,f_sample);
   if (minN>=maxN)
         {
         printf(“minN>=maxN!”);
         return(-1);
         }
```

(listing continues on next page)

```
f1=fopen(f1name,"w"); if (f1==NULL) printf("\nerror opening %s\n",f1);
f2=fopen(f2name,"w"); if (f2==NULL) printf("\nerror opening %s\n",f2);
for (N=minN; N<=maxN; N++)
   {
   maxerrsqr=0;
   binwidth=f_sample/(float)N;
   for (i=0; i<HOW_MANY_TONES; i++)
         {
         kflt=((float)(N))*(f_tone[i]/f_sample);
         kint=round(kflt);
         errsqr=(kflt-(float)(kint))*(kflt-(float)(kint));
         if (errsqr>maxerrsqr) maxerrsqr=errsqr;
         }
   detect=((float)(N)*1000.0)/f_sample;
   printf("\n%f =maxerrsqr ",maxerrsqr);
   printf("(N=%6d) ",N);
   printf("detect= %10.3f ms ",detect);
   printf("resolu= %10.3f Hz",binwidth);
   if ((f1)&&(f2))
         {
         fprintf(f1,"%f\n",maxerrsqr);
         fprintf(f2,"%d\n",N);
         }
   }
printf("\nyou may want to pipe output to sort utility");
printf("\nsort according to incr maxerrsqr");
printf("\nor plot %s vs %s\n",f1name,f2name);
fclose(f1);
fclose(f2);
}
```

Listing 8.4 Best Number Of Samples Routine (BESTN.C)

```
#include <math.h>
#include <stdio.h>
#include “tones.def”

int round( x )
float x;
{
   if (x>0) return ( (int) (x+0.5) );
   else if (x<0) return ( (int) (x-0.5) );
   else if (x==0) return ( 0 );
   else printf(“\7bad data in round() function!”);
   return(-1);
}

int flt_to_Q15( x, txt )
float x;
char txt[];
{
   int i, err=0;

   i = round(x*32768.0);
   if (x>=1.0)   { i=0x7FFF; err=1; }
   if (x<(-1.0)) { i=0x8000; err=(-1); }
   sprintf( txt, “%08X\n”, i );
   txt[0]=txt[4]; txt[1]=txt[5]; txt[2]=txt[6];
   txt[3]=txt[7]; txt[4]=’\000';
   return(err);
}

main(argc,argv)
int argc;
char **argv;
{
   int i, kint;
   float N, f_sample, kflt, kerr, coef;
   char Q15coef[255];

   switch(argc) /* get missing arguments */
         {
         case 1:  printf(“N > “);        scanf(“%f”,&N);
         case 2:  printf(“f_sample > “); scanf(“%f”,&f_sample);
         case 3:  break;
         default: printf(“\n\7usage: %s <N [%%f]> <f_sample [%%f]>\n”);
         return(-1);
         }
```

(listing continues on next page)

```
   switch(argc) /* read the arguments */
         {
         case 3:  sscanf(argv[2],"%f",&f_sample);
         case 2:  sscanf(argv[1],"%f",&N);
         case 1:  break;
         }
   printf("\nN=%f\nfs=%f\n",N,f_sample);
   for (i=0; i<HOW_MANY_TONES; i++)
         {
         kflt=N*(f_tone[i]/f_sample);
         kint=round(kflt);
         kerr=kflt-(float)(kint);
         coef=2*cos((2*PI*(float)kint)/N);
         flt_to_Q15( coef/2, Q15coef );
         printf("\nf_tone[%2d]=%10.2f Hz",i,f_tone[i]);
         printf("\n\t\tk(flt)=%10.6f",kflt);
         printf("\n\t\tk(int)=%4d",kint);
         printf("\n\t\tk(err)=%+10.6f",kerr);
         printf("\n\t\tcoef(flt)=%+10.6f coef(2.14 hex)=0x%s",coef,Q15coef); }
         printf("\n");
}
```

Listing 8.5 Coefficient Generating Routine (COEFGEN.C)

Digital Tone Detection 8

```
.module/ram/abs=0    Tone_Detection;
{
   This example shows digital tone detection using the Goertzel algorithm.
   This example was designed to run on the ADSP-2100 Evaluation Board.
   Actual implementation in other systems would require modifications
   such as redefining the i/o ports and the data i/o handling.

   Analog Devices, Inc. — DSP Division — Norwood, MA 02062 — 4-April-1989
}

.const   f_sample  =48600;
.const   N         =463;
.const   tones     =8;
.const   tones_x_2 =16;

.var/circ   Q1Q2_buff[tones_x_2];   { Goertzel feedback loop storage elements  }
.var        outcode;
.var        in_sample;              { input samples (scaled down 8 bits)       }
.var        countN;                 { counts samples 1, 2, 3, ..., N           }
.var        mu_lookup_table[256];   { mu-law to linear tbl(scaled down 8 bits) }
.var        min_tone_level[tones];  { min "tone-present" mnsqr level           }
.var        mnsqr[tones];           { 1.15 mnsqr Goertzel result values        }
.var        bits[tones];            { 1.15 mnsqr Goertzel result values        }
.var/pm/ram/circ
         coefs[tones];              { 2.14 Goertzel coefs: 2*cos(2*PI*k/N)     }
.var/pm  trashbin;                  { see release note about writes to pm(I4)  }

.port       codec;                  { telephone band speech i/o on Eval. Bd.   }
.port       cntl_port;              { part of above hardware                   }
.port       dac;                    { D/A converter used to monitor decode out }

.init       coefs[00]: h#129B00, h#F89F00, h#DEF100, h#C6A000;
.init       coefs[04]: h#B0AE00, h#9E0200, h#8F6200, h#814C00;

.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 bits:           h#0001,h#0002,h#0004,h#0008,h#0010,h#0020,h#0040,h#0080;
{————————————————————————————————————————————————————————————————————————————}
{———————— M A I N   C O D E ————————————————————————————————————————————————}
{————————————————————————————————————————————————————————————————————————————}
IRQ0:    rti;
IRQ1:    rti;
IRQ2:    rti;
IRQ3:    jump sample;

   i4=^trashbin;        { see release note about writes to pm(I4)              }
   call setup;
   call restart;
   imask=b#1000;        { enable IRQ3 for samples                              }
here:       jump here;
```

(listing continues on next page)

8 Digital Tone Detection

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

{———— G E T   A   S A M P L E   T O   P R O C E S S ————————————————————}
{                                                                          }
sample:  ax0=h#00FF;
         ax1=^mu_lookup_table;
         ay0=dm(codec);              { read codec, mu-law data             }
         af=ax0 and ay0;
         ar=ax1+af;
         i6=ar;
         si=dm(i6,m4);               { look-up scaled, linear value        }
         dm(in_sample)=si;           { store input sample                  }
         i0=^Q1Q2_buff;
         i5=^coefs;

{————— D E C R E M E N T   S A M P L E   C O U N T E R ————————————————————}
{                                                                          }
decN:    ay0=dm(countN);
         ar=ay0-1;
         dm(countN)=ar;
         if lt jump skip_backs;

{————— G O E R T Z E L   F E E D B A C K   P H A S E ————————————————————}
{                                                                          }
feedback:
         ay1=dm(in_sample);                 {get input sample   AY1=1.15       }
         cntr=tones;
         do backs until ce;
            mx0=dm(i0,m0), my0=pm(i5,m4);   {get Q1 and 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                   }
backs:      dm(i0,m0)=mx0;                  {old Q1 = new Q2                   }
         rti;

{————— W H E N   F E E D B A C K   P H A S E   I S   D O N E ————————————}
{                                                                          }
skip_backs:
         call feedforward;
         call test_and_output;
         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 TP3051 codec control ports (ADSP-2100 Evaluation Board),     }
{    M and L registers in address generators, and sets ICNTL to edge-sens.   }
{                                                                            }
setup:  si=0;
         dm(cntl_port)=si;

         l0 = tones_x_2;
         l1 =     0;
         l2 =     0;
         l3 =     0;
         l4 =     0;
         l5 = tones_x_2;
         l6 =     0;

         m0 =     1;
         m1 = -1;
         m4 =     1;

         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=^Q1Q2_buff;
         i5=^coefs;
         cntr=tones_x_2;
         do zloop until ce;
zloop:   dm(i0,m0)=0;
         ax0=N;
         dm(countN)=ax0;
         rts;
```

(listing continues on next page)

```
{%%%%%%%%%% G O E R T Z E L   F E E D F O R W A R D   P H A S E %%%%%%%%%%%%}
{                                                                             }
feedforward: cntr=tones;
   i2=^mnsqr;
   do forwards until ce;
         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(i5,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       }
forwards:   dm(i2,m0)=mr1;         { store in mnsqr buffer       1.15       }
   rts;

{%%%%% T E S T   T O N E   L E V E L S   A N D   O U T P U T   C O D E %%%%%%}
{                                                                             }
test_and_output:
         i3=^bits;
         i1=^min_tone_level;
         i2=^mnsqr;
         cntr=tones;
         af=pass 0;
         do thresholds until ce;
            ax1=dm(i3,m0);         { get bit position to set/clear          }
            ax0=dm(i2,m0);         { get tone mnsqr calculated value        }
            ay0=dm(i1,m0);         { get min tone level threshold value     }
            ar=ax0-ay0;            { mnsqr - min_tone_level                 }
thresholds:
         if gt af=ax1 or af;
         ar=pass af;
         dm(outcode)=ar;           { write bit-coded result to output       }
         rts;
.endmod;
```

Listing 8.6 Tone Detection Routine (EXAMPLE.DSP)