DTMF Decoder Reference Design

Relevant Devices

This application note applies to the following devices:
C8051F300

1. Introduction

Using the 25 MIPS CPU and on-chip ADC, the C8051F300 can perform DTMF tone generation and decoding. The combination of high performance and small size (3x3 mm) makes the C8051F300 an ideal choice for applications requiring DTMF decoding or other CPU intensive tasks.

A reference design kit is available for this reference design. The kit includes the following:

- DTMF Decoder Evaluation Board containing a DTMF generator and decoder, keypad, and 7-segment LED.
- DTMF Decoder Kit User's Guide with step-by-step instructions for running the DTMF Demo on the DTMF Decoder Evaluation Board.
- Reference Design Kit CD containing the Silicon Laboratories Integrated Development Environment (IDE) and an evaluation version of Keil C8051 Development Tools (assembler, 2 kB object code limited compiler, and linker).

This reference design includes the following:

- Background and theory of DTMF decoding using the Goertzel Algorithm.
- Description of a software implementation on a C8051F300 MCU.
- Full C source code for the DTMF decoder and generator. The software is included at the end of the application note and in a separate Zip file (AN218SW.zip) containing the firmware in C and HEX file format. The Zip file is available on the Reference Design Kit CD and on the Silicon Laboratories website.

2. Background

Dual tone multi frequency (DTMF) is a method of representing digits with tones for transmission over an analog communication channel. DTMF tones are used by all touch tone phones to represent the digits on a touch tone keypad. DTMF technology provides a robust alternative to rotary telephone systems and allows user-input during a phone call. This feature has enabled interactive, automated response systems such as the ones used for telephone banking, routing customer support calls, voicemail, and similar applications.

A DTMF tone consists of two superimposed sinusoidal signals selected from two frequency groups. The frequency groups represent rows and columns on a touch tone keypad as shown in Figure 1. Each DTMF tone must contain one sinusoid from the high-frequency group (1209, 1336, 1477 and 1633 Hz) and one sinusoid from the low-frequency group (697, 770, 852 and 941 Hz). This allows a touch tone keypad to have up to 16 unique keys.

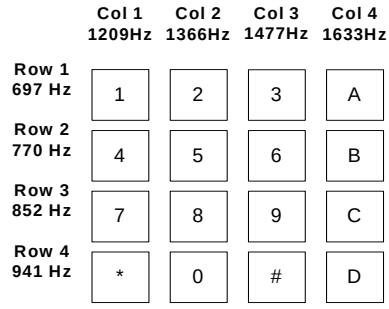

Figure 1. Touch Tone Keypad

The frequencies selected for DTMF tones have some distinguishing characteristics and unique properties.

- All tones are in the audible frequency range allowing humans to detect when a key has been pressed.
- No frequency is a multiple of another.
- The sum or difference of any two frequencies does not equal another selected frequency.

The second and third properties simplify DTMF decoding and reduce the number of falsly detected DTMF tones. The unique properties allow DTMF receivers to detect when a user has pressed multiple keys simultaneously and reject any tones that have harmonic energy. Harmonic energy can only be generated by speech/noise and not by the sum of two valid DTMF frequencies.

To maintain compatibility between DTMF generators and decoders used worldwide, DTMF tones should always be generated and decoded according to the International Telecommunication Union (ITU) Recommendations Q.23 and Q.24. The ITU website is http://www.itu.int.

3. DTMF Detection and Decoding

Detecting DTMF tones requires the capability to detect and differentiate between the 8 DTMF frequencies. It is also important to have a technique of detecting and rejecting false tones caused by noise (e.g., speech). One of the performance metrics used to evaluate DTMF decoders is talk-off error. Talk-off error is defined as the improper detection of a DTMF tone due to speech or other forms of noise in the communication channel. One method for distinguishing between DTMF tones and speech is checking for the presence of a second harmonic. If the second harmonic is detected, then the signal can be rejected because true DTMF tones only contain fundamental tones.

Frequency detection is typically accomplished by applying the discrete fourier transform (DFT) to a time-domain signal to extract frequency information. The DFT, and other transforms such as the fast fourier transform (FFT), generate frequency information for the entire range of frequencies from dc to the sampling rate divided by two. Performing a full DFT or FFT in real-time is complex, requires large amounts of memory, and is not suitable for an 8-bit microcontroller. Instead, the Goertzel Algorithm can be used to extract specific “bins” of a DFT from the input signal without performing an entire DFT or FFT. The Goertzel Algorithm is very suitable for DTMF decoding since the decoder is only concerned with detecting energy at the 8 DTMF frequencies and their second harmonics.

The Goertzel algorithm works on blocks of ADC samples. Samples are processed as they arrive and a result is produced every **N** samples, where **N** is the number of samples in each block. If the sampling rate is fixed, the block size determines the frequency resolution or “bin width” of the resulting power measurement. The example below shows how to calculate the frequency resolution and time required to capture **N** samples:

Sampling rate: 8 kHz
Block size (**N**): 200 samples
Frequency Resolution: sampling rate/block size = 40 Hz
Time required to capture **N** samples: block size/sampling rate = 25 ms

The tradeoff in choosing an appropriate block size is between frequency resolution and the time required to capture a block of samples. Increasing the output word rate (decreasing block size) increases the “bin width” of the resulting power measurement. The bin width should be chosen to provide enough frequency resolution to differentiate between the DTMF frequencies and be able to detect DTMF tones with reasonable delay. See [2] for additional information about choosing a block size.

The processing requirements for the Goertzel algorithm are equivalent to a two-pole IIR filter. Three variables Q_0, Q_1, and Q_2 are required to hold intermediate results. For each ADC sample, the value of Q_0 is computed and the previous values of Q_0 and Q_1 become the new Q_1 and Q_2, respectively. The initial values of Q_1 and Q_2 are zero.

The computation required for each ADC sample is shown below:

$$Q_0 = (coef_k \times Q_1[n]) - Q_2[n] + x[n]$$

$$Q_1 = Q_0[n-1]$$

$$Q_2 = Q_1[n-1]$$

where:

$$k = (int)\left(0.5 + \frac{N \times \text{DTMF\_Target\_Frequency}}{\text{Sampling\_Rate}}\right)$$

$$coef_k = 2\cos\left(\frac{2\pi \times k}{N}\right)$$

$x[n]$ = ADC Sample

After **N** samples have been received and Q_0, Q_1, and Q_2 have been updated **N** times, the signal power at the target frequency can be computed using the following equation:

$$power = magnitude^2 = Q_1^2[N] + Q_2^2[N] - (coef_k \times Q_1[N] \times Q_2[N])$$

Notice that the value of $coef_k$ is constant throughout the calculations and can be pre-computed at compile time for the 8 possible DTMF frequencies and their second harmonics.

Once signal power is determined for each of the DTMF frequencies, the power levels can be compared to a threshold value or with each other to determine the row and column of the key pressed. If no keys or multiple keys were pressed, or if energy is detected at the second harmonic, the controller can take no action. If a valid row and column are determined, the controller can report the DTMF tone to the user.

4. Software Implementation

The general implementation of this DTMF detector is modeled by the signal flow diagram shown in Figure 2. Each of the following sections corresponds to a block in the diagram. The blocks also describe the labeled sections of code. The flowcharts in Figures 3 and 4 show the software flow of the ADC0 Interrupt Service Routine and the DTMF_Detect() function. Following along in the code while reading this section will be helpful in understanding the operation of this software.

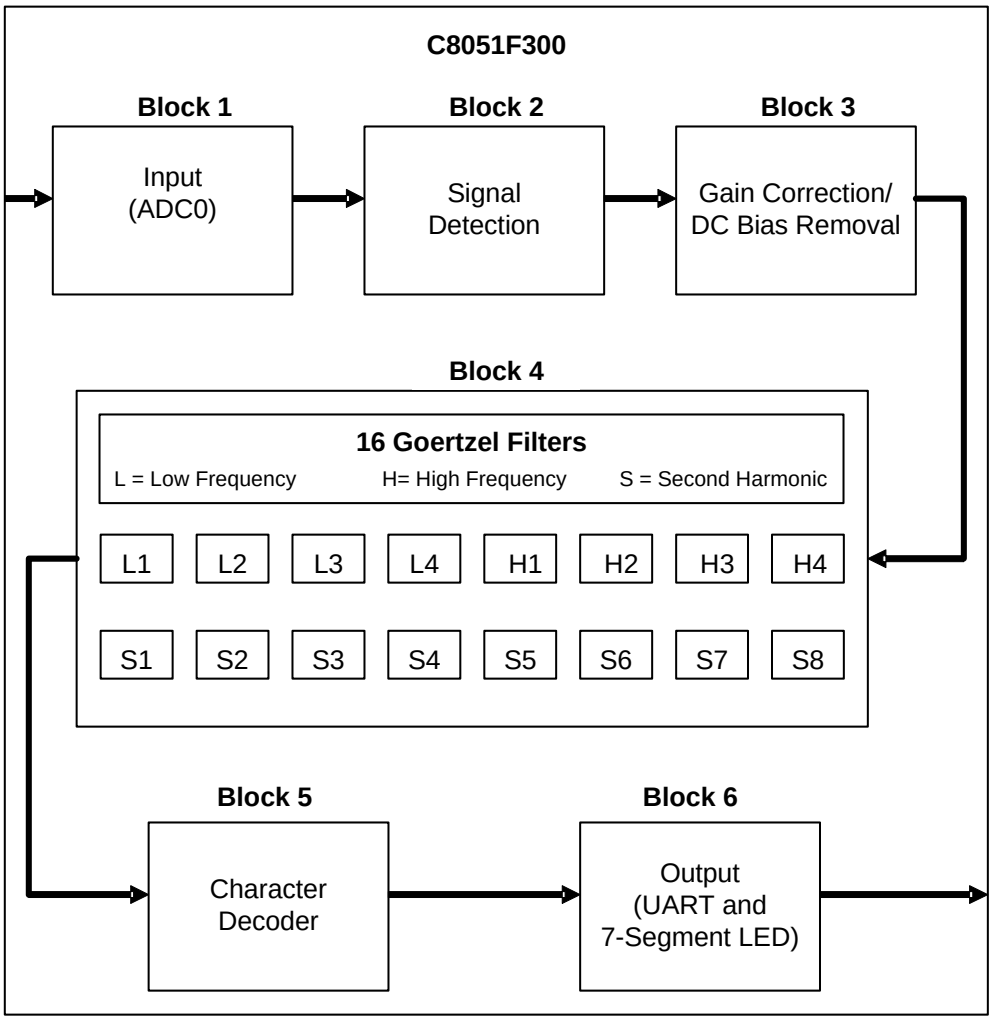

Figure 2. Software Signal Flow Diagram

4.1. Block 1—Input

ADC0 handles the input signal sampling. The sampling rate is set at 8 kHz, with conversions starting every Timer 2 overflow. Even though the highest DTMF frequency is 1633 Hz, an 8 kHz sampling rate is necessary to correctly sample the second harmonics and reduce talk-off error. This allows 125 µs of CPU time per sample.

4.2. Block 2—Signal Detection

The signal detection block determines whether or not a signal is present at the ADC input. After a sample has been acquired, its magnitude is compared to the magnitude of the previous sample. If the difference between the two is greater than a predefined threshold value, the DTMF decoding process is started. If the magnitude does not meet the valid signal condition and there is no DTMF detection currently in progress, then the sample is ignored.

Separate DTMF tones are required to have a 10 ms to 40 ms gap between them. In this implementation, DTMF detection is disabled for the 20 ms period following the end of a DTMF tone. During this 20 ms period, any signal detected at the ADC input, such as a new DTMF tone, will restart the timeout and will not be interpreted as a valid DTMF tone. DTMF detection can only be re-enabled by the Timer 0 Interrupt Service Routine (ISR) after 20 ms of silence at the ADC input. The 20 ms timeout can be varied by changing the Timer 0 reload value.

4.3. Block 3—Gain Control/DC Bias Removal

The gain control block applies a dynamic gain to ADC input to offset variations present in a typical telephone line and ensure that the input signal utilizes the full dynamic range of the ADC (0 V to V_{REF}). The dynamic gain is determined by finding the peak amplitude of the first 70 samples then scaling the rest of the samples such that the peak value is equal to V_{REF}. The gain is recalculated at the beginning of every tone.

The gain control block also removes any dc bias from the input signal by recording the minimum value of the first 70 samples. Knowing the maximum and minimum values of the input signal, the mean value is computed and subtracted from future samples. Removing the dc bias from the input signal provides consistent power measurements and minimizes errors. The dc bias is recalculated at the beginning of every tone.

4.4. Block 4A—Goertzel Filters

The Goertzel filter block contains 16 filters to determine the energy present at the 8 DTMF frequencies and their second harmonics. At 125 µs of CPU time per sample, it is not possible to execute all 16 filters concurrently in real-time. Instead, the filters are divided into two groups of 8. The first group is executed on the first **N** samples received and determine the presence of energy at the low and high fundamental frequencies. The second group is executed on a second set of **N** samples and are used to determine the presence of energy at the second harmonics. All samples are processed as they arrive and the value of **N** may vary between the two filter groups.

4.5. Block 4B—Goertzel Filter Output (Signal Power Calculation)

The Goertzel filter output stage computes the signal power for each of the 16 tested frequencies and stores them for use by the next block.

4.6. Block 5—DTMF Decode

The DTMF Decode block determines which of the 16 frequencies have enough energy to contain a signal. The signal power of each is compared to the remaining signals in the frequency group (low or high). If the signal power is K times (K = 8 in this implementation) greater than the sum of signal power from the other three signals, the input signal is determined to contain a sinusoid at the tested frequency. The results are stored in memory and used by the next block.

4.7. Block 6—DTMF Output

The DTMF Output block uses the result of the previous block to determine if a valid DTMF Tone corresponding to a unique character has been received. If multiple frequencies from the same group are detected, or the combined energy from the second harmonics exceeds a threshold, the input signal is rejected. The rest of the software in this block is application dependant. In this implementation, the detected character is printed to the UART and a 7-segment LED display is updated to display the decoded character.

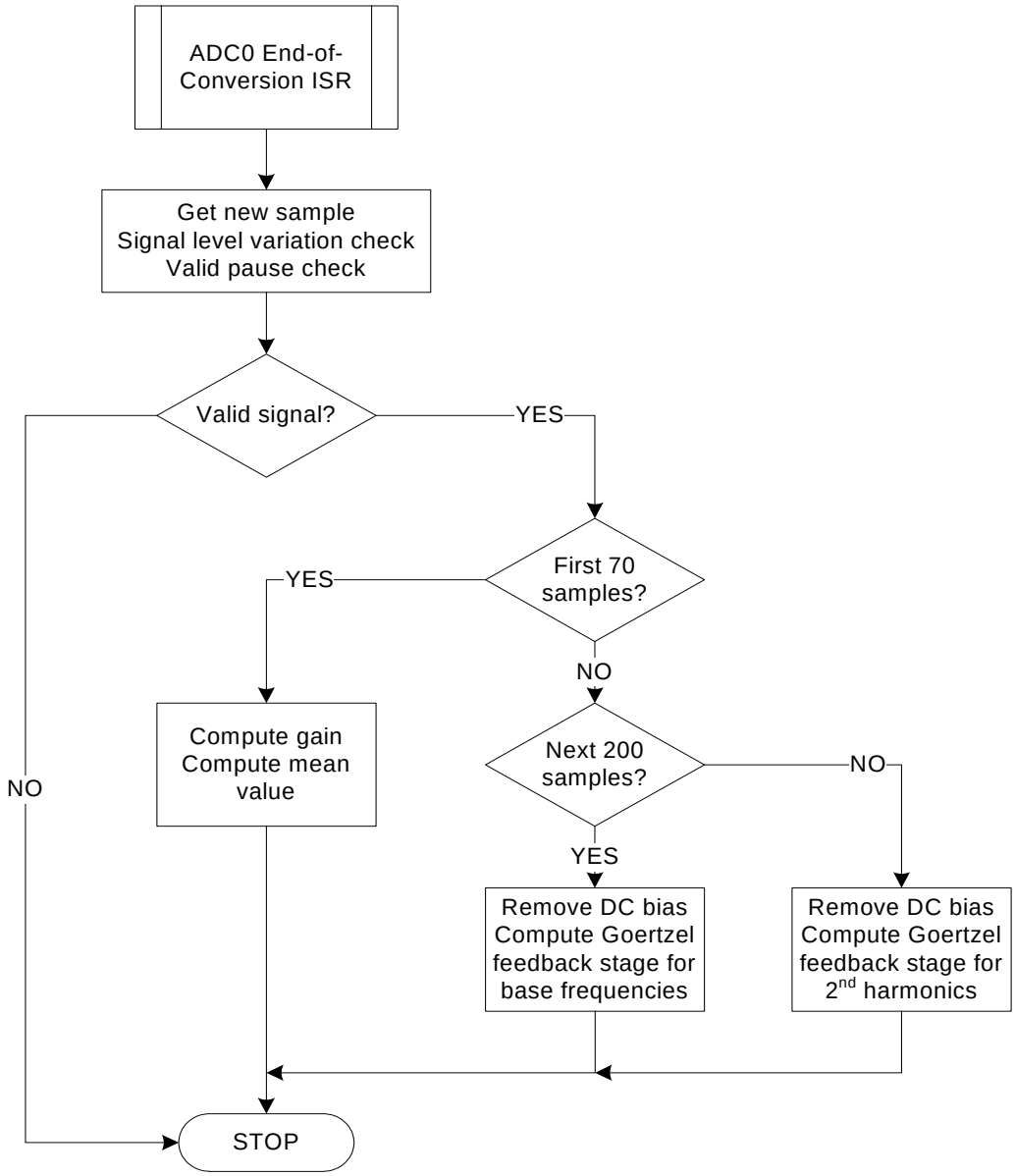

Figure 3. ADC0 End-of-Conversion ISR Flowchart

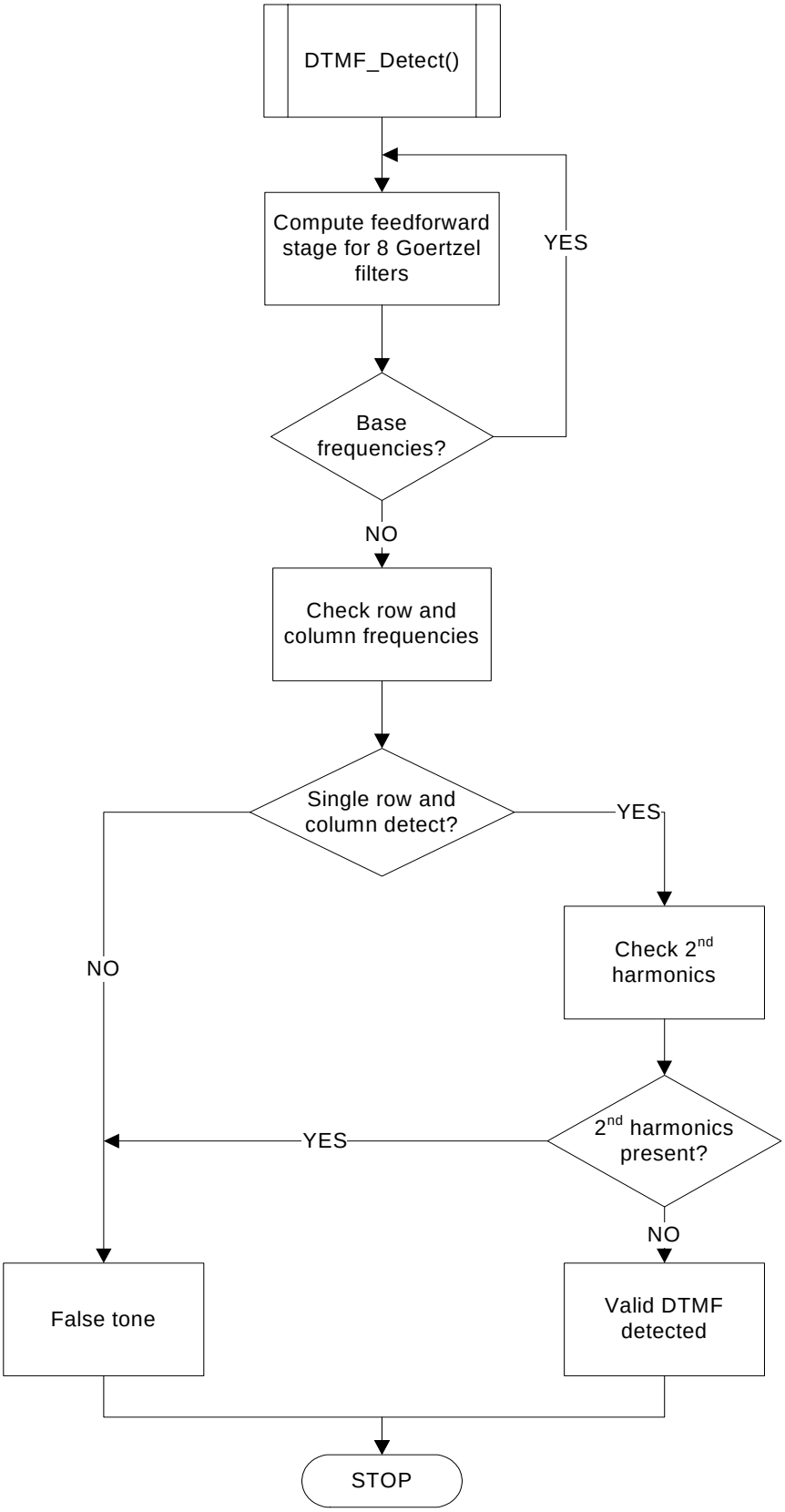

Figure 4. DTMF_Detect() Function Flowchart

5. Testing

The software implementation of the Goertzel algorithm has been tested for talk-off errors using the Bellcore™ digit simulation test tape. For a 4 hour talk-off test with a calibration tone voltage of 72.5 mV_{RMS}, no false hits were detected. In another 4 hour test with a calibration tone voltage of 135.5 mV_{RMS}, only 11 false hits were detected. The software implementation in this reference design, running on the DTMF Decoder Evaluation Board, performs very well with respect to the limits specified by the Bellcore™ test tapes.

6. References

[1] K. Banks, “The Goertzel Algorithm,” Embedded.com, August 28, 2002, <http://www.embedded.com/showArticle.jhtml?articleID=9900722>

[2] Oppenheim, Schafer, *Discrete Time Signal Processing*, Prentice-Hall, 1989

7. Software Example

```
//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories, Inc.
//
// FILE NAME      : dtmf.c
// TARGET DEVICE  : C8051F300
// CREATED ON     : 30.04.2004
// CREATED BY     : SYRO
//
// Revision 1.0
// This file contains the source code of the DTMF decoder


//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
#include <stdio.h>
#include “c8051f300.h”                    // SFR declarations
#include “dtmf.h”


sbit DATA = P0^5;                          // SIPO buffer data line
sbit CLK = P0^7;                           // SIPO buffer clock line


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
//
void main(void)
{
    EA = 0;                                // All interrupts disabled
    PCA0MD &= ~0x40;                       // Clear watchdog timer

    SYSCLK_Init();                         // Configure system clock
    PORT_Init();                           // Condigure I/O port
    Timer0_Init(SYSCLK/12/50);             // Configure Timer0
    Timer2_Init(SYSCLK/12/SAMPLE_RATE);    // Configure Timer2
    ADC0_Init();                           // Configure ADC0
    CP0_Init();                            // Configure Comparator0
    UART0_Init();                          // Configure UART0
    PCA0_Init();
    Interrupt_Init();                      // Initialize interrupts
    DTMF_Init();                           // Variable initializations

    while(1)
    {
        Idle();                            // Switch to Idle Mode

        while(!done);                      // Wait the feedback stage of
                                           // Goertzel filters for 2nd
                                           // harmonics to complete
```

```
        DTMF_Detect();                      // Compute feedforward stage of
                                            // Goertzel filters and decode
                                            // the DTMF character
    }
}


//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use the internal oscillator
// as its clock source.  Also enables missing clock detector reset.
//
void SYSCLK_Init (void)
{
    unsigned int i;

    OSCICN = 0x07;                          // configure internal oscillator for
                                            // its highest frequency

    for(i=0;i<255;i++);
    while(!(OSCXCN | 0x80));
    OSCXCN = 0x61;

    RSTSRC = 0x06;                          // Enable Missing Clock Detector
                                            // and VDD Monitor
}


//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports.
// P0.0 - ADC0 input
// P0.1 - CP0 Negative Input
// P0.2 - External Oscillator
// P0.3 - External Oscillator
// P0.4 - TX
// P0.5 - RX
// P0.6 - CP0 Positive Input
// P0.7 - C2D
//
void PORT_Init (void)
{
    XBR0     = 0x4F;                    // skip P0.0, P0.1, P0.2, P0.3
                                        // and P0.6

    XBR1     = 0x01;                    // UART TX selected

    XBR2     = 0xC0;                    // Enable crossbar and disable
                                        // weak pull-ups
```

```
    P0MDIN &= ~0x4F;                    // P0.0, P0.1, P0.2, P03 and P0.6
                                        // configured as analog inputs

    P0MDOUT = 0xFF;                     // All output pins are push-pull
                                        // except for P0.6
}


//-----------------------------------------------------------------------------
// Timer0_Init
//-----------------------------------------------------------------------------
//
// This routine configures Timer0
//
void Timer0_Init(int counts)
{
    CKCON &= ~0x0B;                       // Timer0 clock source = SYSCLK/12

    TMOD &= ~0x0E;                        // Timer0 in 16 bit mode
    TMOD |= 0x01;

    TH0 = -counts >> 8;                   // Load Timer0 registers
    TL0 = -counts & 0xFF;
    TMR0RLH = TH0;                        // Save values of Timer0 registers
    TMR0RLL = TL0;                        // for reload
}


//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// Timer2 is configured in 16-bit autoreload mode. Its overflows will trigger
// the ADC0 conversions.
//
void Timer2_Init (int counts)
{
    TMR2CN = 0x00;                        // Timer2 configured for 16-bit
                                          // auto-reload, low-byte interrupt
                                          // disabled

    CKCON &= ~0x20;                       // Timer2 clock source = SYSCLK/12

    TMR2 = -counts;                       // Load Timer2 registers
    TMR2RL = -counts;
}


//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// ADC0 is configured in Single-Ended mode. Conversions are triggered by
// Timer2 overflows.
```

```
//
void ADC0_Init(void)
{
   ADC0CN = 0x02;                          // ADC0 disabled; ADC0 conversions
                                           // are initiated on overflow of
                                           // Timer2

   AMX0SL = 0xF0;                          // Select P0.0 as ADC0 input
                                           // ADC0 in Single-Ended Mode

   ADC0CF = (SYSCLK/5000000) << 3;         // ADC conversion clock < 5.0MHz
   ADC0CF |= 0x01;                         // PGA gain = 1

   REF0CN = 0x0A;                          // VREF = VDD,
                                           // Bias generator is on.

   AD0EN = 1;                              // Enable ADC0
}


//-----------------------------------------------------------------------------
// CP0_Init
//-----------------------------------------------------------------------------
//
// Configure Comparator0
//
void CP0_Init(void)
{
    CPT0CN = 0x8C;                         // Comparator0 enabled
                                           // 20 mV positive hysteresis

    CPT0MX = 0x03;                         // P0.1 negative input
                                           // P0.6 positive input

    CPT0MD = 0x03;                         // CP0 response time 1050ns
}


//-----------------------------------------------------------------------------
// UART_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1 for <BAUDRATE> and 8-N-1.
//
void UART0_Init(void)
{
    SCON0 = 0x10;                          // SCON0: 8-bit variable bit rate
                                           //        level of STOP bit ignored
                                           //        RX enabled
                                           //        ninth bits are zeros
                                           //        clear RI0 and TI0 bits
    if (SYSCLK/BAUDRATE/2/256 < 1)
    {
        TH1 = -(SYSCLK/BAUDRATE/2);
```

```
        CKCON |=  0x10;                     // T1M = 1; SCA1:0 = xx
    }
    else if (SYSCLK/BAUDRATE/2/256 < 4)
    {
        TH1 = -(SYSCLK/BAUDRATE/2/4);
        CKCON &= ~0x13;                     // T1M = 0; SCA1:0 = 01
        CKCON |=  0x01;
    }
    else if (SYSCLK/BAUDRATE/2/256 < 12)
    {
        TH1 = -(SYSCLK/BAUDRATE/2/12);
        CKCON &= ~0x13;                     // T1M = 0; SCA1:0 = 00
    }
    else
    {
        TH1 = -(SYSCLK/BAUDRATE/2/48);
        CKCON &= ~0x13;                     // T1M = 0; SCA1:0 = 10
        CKCON |=  0x02;
    }

    TL1 = TH1;                              // Initialize Timer1
    TMOD &= ~0xF0;                          // TMOD: Timer1 in 8-bit autoreload
    TMOD |=  0x20;
    TR1 = 1;                                // START Timer1
    TI0 = 1;                                // Indicate TX0 ready
}


//-----------------------------------------------------------------------------
// PCA0_Init
//-----------------------------------------------------------------------------
//
// Configure PCA0
//
void PCA0_Init(void)
{
    PCA0MD = 0x0B;                          // PCA0 timer uses external clock
                                            // divided by 8
                                            // PCA0 timer overflow interrupt
                                            // enabled

    CR = 1;                                 // Start PCA0 timer
}


//-----------------------------------------------------------------------------
// Interrupt_Init
//-----------------------------------------------------------------------------
//
// Enables the interrupts and sets their priorities
//
void Interrupt_Init(void)
{
    IE = 0;                                 // All interrupts disabled
```

```
    EIE1 = 0;

    PT0 = 0;                                // Timer0 interrupt low priority
    ET0 = 1;                                // Timer0 interrupt enable

    EIP1 |= 0x04;                           // ADC0 interrupt high priority
    EIE1 |= 0x04;                           // ADC0 interrupt enable

    EIP1 &= ~0x08;                          // PCA0 interrupt low priority
    EIE1 |= 0x08;                           // PCA0 interrupt enable

    EA = 1;                                 // Enable interrupts
}


//-----------------------------------------------------------------------------
// ADC0 Conversion Complete Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// ADC0 ISR implements the signal detection, automatic gain control,
// DC bias removal and the feedback phase of the Goertzel filters
//
//
void ADC0_ISR(void) interrupt 8 using 1
{
    AD0INT = 0;                             // Clear the interrupt flag

    x = ADC0;                               // Get a new sample


//-------------------------------------------------------------------------//
//                                                                         //
//                              BLOCK 2                                    //
//                          Signal Detection                               //
//                                                                         //
//-------------------------------------------------------------------------//

    delta_x = x - x_old;

    // The signal variation is big enough
    if( ( delta_x > XMIN ) || ( delta_x < -XMIN ) )
    {
        if(new_tone)                        // The required pause between two
        {                                   // tones has passed

            ET0 = 0;                        // Disable Timer0 interrupt
            new_tone = 0;                   // Reset new_tone, only way for this
                                            // code to execute again is if there
                                            // is a 20ms gap in the signal and
                                            // Timer0 overflows.

            start_goertzel = 1;             // A new tone has been detected,
                                            // so start the DTMF detect.
        }
```

```
        TH0 = TMR0RLH;                  // Reload Timer0 registers
        TL0 = TMR0RLL;
    }

    x_old = ADC0;
//-----------------------------------------------------------------------------//
//                                                                             //
//                               BLOCK 3                                       //
//                         Automatic Gain Control                              //
//                                                                             //
//-----------------------------------------------------------------------------//

    // Initially, the signal input is passed through the following gain
    // calculation routine.  The gain is produced by dividing the maximum
    // possible signal magnitude by the greatest input magnitude of the
    // current tone.  This produces a gain that when multiplied with the
    // incoming signal will increase the signal amplitude to nearly full
    // scale. Also the routine finds the lowest input magnitude, which will
    // be used to compute the average value of the signal for DC bias removal.

    if(start_goertzel)                  // New tone detected
    {
        if(gain_calc)                   // Gain calculation phase
        {
            if(x > high_x)
                high_x = x;             // Find maximum value
            else
            if(x < low_x)
                low_x = x;              // Find minimum value
            gain_cnt++;
            if(gain_cnt >= 70)
            {
                gain_cnt = 0;           // Reset gain counter
                gain_calc = 0;          // Reset gain calculation flag

                // compute gain
                gain = ( 256 / (high_x - low_x) );

                // low_x will contain the average value
                low_x += (high_x - low_x)>>1;
            }
        }
        else                            // Gain calculation completed
        {
            x = (x - low_x) * gain;     // Scale input to nearly full scale
                                        // of ADC0, and remove DC bias

//-----------------------------------------------------------------------------//
//                                                                             //
//                               BLOCK 4A                                      //
//                 Feedback Phase of Goertzel Filters                          //
//                                                                             //
//-----------------------------------------------------------------------------//
```

```
//Filter 1 Sample and Feedback
Qhold.l = (long)coef1[base_freq]*(long)Q1[1].i;
Q1[0].i = x;
Q1[0].i += Qhold.hold.intval;
Q1[0].i -= Q1[2].i;
Q1[2].i = Q1[1].i;
Q1[1].i = Q1[0].i;
//End Filter 1

//Filter 2 Sample and Feedback
Qhold.l = (long)coef2[base_freq]*(long)Q2[1].i;
Q2[0].i = x;
Q2[0].i += Qhold.hold.intval;
Q2[0].i -= Q2[2].i;
Q2[2].i = Q2[1].i;            // Feedback part of filter
Q2[1].i = Q2[0].i;
//End Filter 2

//Filter 3 Sample and Feedback
Qhold.l = (long)coef3[base_freq]*(long)Q3[1].i;
Q3[0].i = x;
Q3[0].i += Qhold.hold.intval;
Q3[0].i -= Q3[2].i;
Q3[2].i = Q3[1].i;            // Feedback part of filter
Q3[1].i = Q3[0].i;
//End Filter 3

//Filter 4 Sample and Feedback
Qhold.l = (long)coef4[base_freq]*(long)Q4[1].i;
Q4[0].i = x;
Q4[0].i += Qhold.hold.intval;
Q4[0].i -= Q4[2].i;
Q4[2].i = Q4[1].i;            // Feedback part of filter
Q4[1].i = Q4[0].i;
//End Filter 4

//Filter 5 Sample and Feedback
Qhold.l = (long)coef5[base_freq]*(long)Q5[1].i;
Q5[0].i = x;
Q5[0].i += Qhold.hold.intval;
Q5[0].i -= Q5[2].i;
Q5[2].i = Q5[1].i;            // Feedback part of filter
Q5[1].i = Q5[0].i;
//End Filter 5

//Filter 6 Sample and Feedback
Qhold.l = (long)coef6[base_freq]*(long)Q6[1].i;
Q6[0].i = x;
Q6[0].i += Qhold.hold.intval;
Q6[0].i -= Q6[2].i;
Q6[2].i = Q6[1].i;            // Feedback part of filter
Q6[1].i = Q6[0].i;
//End Filter 6
```

```
//Filter 7 Sample and Feedback
Qhold.l = (long)coef7[base_freq]*(long)Q7[1].i;
Q7[0].i = x;
Q7[0].i += Qhold.hold.intval;
Q7[0].i -= Q7[2].i;
Q7[2].i = Q7[1].i;            // Feedback part of filter
Q7[1].i = Q7[0].i;
//End Filter 7

//Filter 8 Sample and Feedback
Qhold.l = (long)coef8[base_freq]*(long)Q8[1].i;
Q8[0].i = x;
Q8[0].i += Qhold.hold.intval;
Q8[0].i -= Q8[2].i;
Q8[2].i = Q8[1].i;            // Feedback part of filter
Q8[1].i = Q8[0].i;
//End Filter 8

if(sample_no == max_sample)   // Feed forward and output...
{
    sample_no = 0;            // Reset sample counter
    done = 1;                 // Set done flag so magnitude calculation
                              // can run

   // The following code essentially scales each element down by 8 bits.

    // Filter1 Element Update
    Q1[1].b[1] = Q1[1].b[0];  // shift 8 bits right
    Q1[2].b[1] = Q1[2].b[0];

    // sign extension
    if(Q1[1].b[0]<0) Q1[1].b[0] = 0xFF;
    else Q1[1].b[0] = 0x00;

    if(Q1[2].b[0]<0) Q1[2].b[0] = 0xFF;
    else Q1[2].b[0] = 0x00;

    // Filter2 Element Update
    Q2[1].b[1] = Q2[1].b[0];
    Q2[2].b[1] = Q2[2].b[0];

    // sign extension
    if(Q2[1].b[0]<0) Q2[1].b[0] = 0xFF;
    else Q2[1].b[0] = 0x00;

    if(Q2[2].b[0]<0) Q2[2].b[0] = 0xFF;
    else Q2[2].b[0] = 0x00;

    // Filter3 Element Update
    Q3[1].b[1] = Q3[1].b[0];
    Q3[2].b[1] = Q3[2].b[0];

    // sign extension
```

```
if(Q3[1].b[0]<0) Q3[1].b[0] = 0xFF;
else Q3[1].b[0] = 0x00;

if(Q3[2].b[0]<0) Q3[2].b[0] = 0xFF;
else Q3[2].b[0] = 0x00;

// Filter4 Element Update
Q4[1].b[1] = Q4[1].b[0];
Q4[2].b[1] = Q4[2].b[0];

// sign extension
if(Q4[1].b[0]<0) Q4[1].b[0] = 0xFF;
else Q4[1].b[0] = 0x00;

if(Q4[2].b[0]<0) Q4[2].b[0] = 0xFF;
else Q4[2].b[0] = 0x00;

// Filter5 Element Update
Q5[1].b[1] = Q5[1].b[0];
Q5[2].b[1] = Q5[2].b[0];

// sign extension
if(Q5[1].b[0]<0) Q5[1].b[0] = 0xFF;
else Q5[1].b[0] = 0x00;

if(Q5[2].b[0]<0) Q5[2].b[0] = 0xFF;
else Q5[2].b[0] = 0x00;

// Filter6 Element Update
Q6[1].b[1] = Q6[1].b[0];
Q6[2].b[1] = Q6[2].b[0];

// sign extension
if(Q6[1].b[0]<0) Q6[1].b[0] = 0xFF;
else Q6[1].b[0] = 0x00;

if(Q6[2].b[0]<0) Q6[2].b[0] = 0xFF;
else Q6[2].b[0] = 0x00;

// Filter7 Element Update
Q7[1].b[1] = Q7[1].b[0];
Q7[2].b[1] = Q7[2].b[0];

// sign extension
if(Q7[1].b[0]<0) Q7[1].b[0] = 0xFF;
else Q7[1].b[0] = 0x00;

if(Q7[2].b[0]<0) Q7[2].b[0] = 0xFF;
else Q7[2].b[0] = 0x00;

// Filter8 Element Update
Q8[1].b[1] = Q8[1].b[0];
Q8[2].b[1] = Q8[2].b[0];
```

```
// sign extension
if(Q8[1].b[0]<0) Q8[1].b[0] = 0xFF;
else Q8[1].b[0] = 0x00;

if(Q8[2].b[0]<0) Q8[2].b[0] = 0xFF;
else Q8[2].b[0] = 0x00;

// Save the bottom two filter elements for magnitude calculation
Qt1[0].i = Q1[1].i;
Qt1[1].i = Q1[2].i;
Qt2[0].i = Q2[1].i;
Qt2[1].i = Q2[2].i;
Qt3[0].i = Q3[1].i;
Qt3[1].i = Q3[2].i;
Qt4[0].i = Q4[1].i;
Qt4[1].i = Q4[2].i;
Qt5[0].i = Q5[1].i;
Qt5[1].i = Q5[2].i;
Qt6[0].i = Q6[1].i;
Qt6[1].i = Q6[2].i;
Qt7[0].i = Q7[1].i;
Qt7[1].i = Q7[2].i;
Qt8[0].i = Q8[1].i;
Qt8[1].i = Q8[2].i;


Q1[1].i = 0;
Q1[2].i = 0;
Q2[1].i = 0;
Q2[2].i = 0;
Q3[1].i = 0;
Q3[2].i = 0;
Q4[1].i = 0;
Q4[2].i = 0;
Q5[1].i = 0;
Q5[2].i = 0;
Q6[1].i = 0;
Q6[2].i = 0;
Q7[1].i = 0;
Q7[2].i = 0;
Q8[1].i = 0;
Q8[2].i = 0;

// Each window has 8 filters detecting 8 different frequencies.
// If the second window has completed, then the flag to run
// Goertzel is reset, the flag to calculate gain is reset,
// Timer0 interrupt is reenabled, and the Timer0 high and low
// bytes are reloaded with a 20ms delay.
if(base_freq == 0)        // 2nd harmonics filters completed
{
    ET0 = 1;              // Enable Timer0 interrupt
    TH0 =  TMR0RLH;       // Reload Timer0
    TL0 =  TMR0RLL;
    start_goertzel = 0;   // Clear start of decoding flag
```

```
                    gain_calc = 1;        // Set gain calculation flag
                    low_x = 255;          // Initialize minimum and maximum
                    high_x = 0;           // signal values
                    max_sample = 200;     // Number of samples for base
                                          // frequencies
                }
                else                      // Base frequencies filters completed
                {
                    max_sample = 250;     // Number of samples for 2nd harmonics
                }

                // Each time a window completes, this flag is set or reset to
                // indicate which window is next.
                base_freq^=1;
            }

            // Increment the number of iterations and move on until there
            // have been max_sample iterations.
            else
                sample_no++;
        }
    }
}


//-----------------------------------------------------------------------------
// Timer0 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// Timer0 is set to overflow after 20ms with no signal.  If Timer 0 overflows
// then the next tone received is a new tone.
//
void Timer0_ISR (void) interrupt 1 using 2
{
    // If Timer0 overflows, then there has been at least 20ms
    // with no signal, that means the next tone will be a new one.
    new_tone = 1;
    done = 1;
}


//-----------------------------------------------------------------------------
// Comparator0 Rising Edge Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// Comparator0 interrupts are used to wake-up the system from Idle Mode.
// This routine switches the system clock to internal 24.5 MHz oscillator
// and re-enables the peripherals disabled by Idle() routine.
//
void CP0RE_ISR (void) interrupt 11 using 3
{
    OSC_Change(INT_OSC);                  // Switch to internal oscillator

    CPT0CN &= ~0x20;                      // Clear CP0 rising edge interrupt
```

```
                                        // flag

    EIE1 &= ~0x20;                      // Disable CPO rising edge interrupt

    TR0 = 1;                            // Start Timer0
    TR1 = 1;                            // Start Timer1
    TR2 = 1;                            // Start Timer2

    AD0EN = 1;                          // Enable ADC0
}


//-----------------------------------------------------------------------------
// PCA0 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// PCA0 Timer is used to measure the elapsed time since last reset. The
// interrupt is triggered once every 16 seconds.
//
void PCA0_ISR (void) interrupt 9
{
    CF = 0;

    second += 16;
    if(second >= 60)
    {
        second -= 60;
        minute++;

        if(minute >= 60)
        {
            minute = 0;
            hour++;

            if(hour >= 5)
            {
                hour = 0;
            }
        }
    }
}


//-----------------------------------------------------------------------------
// DTMF_Detect
//-----------------------------------------------------------------------------
//
// This routine implements the feedforward stage of the Goertzel filters
// (the calculation of the squared magnitude) and the decoding of the
// received DTMF character. If a valid tone has been detected, the
// character is sent on UART0.
//
void DTMF_Detect(void)
{
```

```
    char temp_sig;                      // All of these temp variables are used
    int temp_dtmf_index;                // to store the values generated by the
    unsigned char temp_set1 = 0;        // first set of magnitude calculations
    unsigned char temp_set2 = 0;        // when the second set of calculations
                                        // occurs.
    unsigned int pca_val;
    unsigned int pca_temp;

    char coef_index;                    // The index for coefficients array

    unsigned char i;

    coef_index = base_freq^1;

    // Reset the done flag so that the next time Timer4 overflows,
    // it spins until the sample and feedback stage is complete.
    done = 0;


//-----------------------------------------------------------------------------//
//                                                                             //
//                               BLOCK 4B                                      //
//                  Feedforward Phase / Output Calculation                     //
//                                                                             //
//-----------------------------------------------------------------------------//

// Here the feedforward and magnitude stages are calculated.
// Notice that the feedforward phase is completed first and stored
// in Qthold.l.  Then the magnitude is calculated using that result
// and the final two filter elements.

    //Filter 1 Output
    Qthold.l = (long)Qt1[0].i * (long)Qt1[1].i * (long)coef1[coef_index];
    mag_squared1 = ((Qt1[0].i*Qt1[0].i)
                 + (Qt1[1].i*Qt1[1].i)
                 - (Qthold.hold.intval));


    //Filter 2 Output
    Qthold.l = (long)Qt2[0].i* (long)Qt2[1].i* (long)coef2[coef_index];
    mag_squared2 = ((Qt2[0].i*Qt2[0].i)
                 + (Qt2[1].i*Qt2[1].i)
                 - (Qthold.hold.intval));

    //Filter 3 Output
    Qthold.l = (long)Qt3[0].i * (long)Qt3[1].i * (long)coef3[coef_index];
    mag_squared3 = ((Qt3[0].i*Qt3[0].i)
                 + (Qt3[1].i*Qt3[1].i)
                 - (Qthold.hold.intval));

    //Filter 4 Output
    Qthold.l = (long)Qt4[0].i * (long)Qt4[1].i * (long)coef4[coef_index];
    mag_squared4 = ((Qt4[0].i*Qt4[0].i)
                 + (Qt4[1].i*Qt4[1].i)
```

```
                - (Qthold.hold.intval));

    //Filter 5 Output
    Qthold.l = (long)Qt5[0].i * (long)Qt5[1].i * (long)coef5[coef_index];
    mag_squared5 = ((Qt5[0].i*Qt5[0].i)
                + (Qt5[1].i*Qt5[1].i)
                - (Qthold.hold.intval));

    //Filter 6 Output
    Qthold.l = (long)Qt6[0].i * (long)Qt6[1].i * (long)coef6[coef_index];
    mag_squared6 = ((Qt6[0].i*Qt6[0].i)
                + (Qt6[1].i*Qt6[1].i)
                - (Qthold.hold.intval));

    //Filter 7 Output
    Qthold.l = (long)Qt7[0].i * (long)Qt7[1].i * (long)coef7[coef_index];
    mag_squared7 = ((Qt7[0].i*Qt7[0].i)
                + (Qt7[1].i*Qt7[1].i)
                - (Qthold.hold.intval));

    //Filter 8 Output
    Qthold.l = (long)Qt8[0].i * (long)Qt8[1].i * (long)coef8[coef_index];
    mag_squared8 = ((Qt8[0].i*Qt8[0].i)
                + (Qt8[1].i*Qt8[1].i)
                - (Qthold.hold.intval));


//-------------------------------------------------------------------------//
//                                                                         //
//                              BLOCK 5                                    //
//                            DTMF Decode                                  //
//                                                                         //
//-------------------------------------------------------------------------//

    if(base_freq)
        temp_sig=sig_present;

    // All of the values calculated in the previous function call
    // are saved here into temporary variables.
    temp_sig = sig_present;
    temp_dtmf_index = dtmf_index;
    temp_set1 = set1;
    temp_set2 = set2;

    // Reset these guys for the decoding process.
    dtmf_index = 0;
    sig_present = 0;
    set1 = 0;
    set2 = 0;


    if(!base_freq)                      // Base frequencies calculation
    {                                   // complete
```

```
// If the energy of a given frequency is eight times greater than
// the sum of the other frequencies from the same group, then it is
// safe to assume that the signal contains that frequency.
// If that frequency is present then a unique bit in sig_present
// is set, dtmf_index is modified so that the correct character
// in the dtmfchar array will be accessed, and the set1 or set2
// flag is incremented to indicate that a low group or high
// group frequency has been detected.
if(mag_squared1 > (mag_squared2+mag_squared3+mag_squared4+8)<<3)
{
    sig_present |= 0x01;
    dtmf_index += 0;
    set1 += 1;
}
else
{
    sig_present &= ~0x01;         // These elses are unnecessary
}

if(mag_squared2 > (mag_squared1+mag_squared3+mag_squared4+2)<<3)
{
    sig_present |= 0x02;
    dtmf_index += 4;
    set1 += 1;
}
else
    sig_present &= ~0x02;

if(mag_squared3 > (mag_squared1+mag_squared2+mag_squared4+2)<<3)
{
    sig_present |= 0x04;
    dtmf_index += 8;
    set1 += 1;
}
else
    sig_present &= ~0x04;

if(mag_squared4 > (mag_squared1+mag_squared2+mag_squared3+2)<<3)
{
    sig_present |= 0x08;
    dtmf_index += 12;
    set1 += 1;
}
else
    sig_present &= ~0x08;

if(mag_squared5 > (mag_squared6+mag_squared7+mag_squared8+1)<<3)
{
    sig_present |= 0x10;
    dtmf_index += 0;
    set2 += 1;
}
else
    sig_present &= ~0x10;
```

```
        if(mag_squared6 > (mag_squared5+mag_squared7+mag_squared8+1)<<3)
        {
            sig_present |= 0x20;
            dtmf_index += 1;
            set2 += 1;
        }
        else
            sig_present &= ~0x20;

        if(mag_squared7 > (mag_squared5+mag_squared6+mag_squared8+1)<<3)
        {
            sig_present |= 0x40;
            dtmf_index += 2;
            set2 += 1;
        }
        else
            sig_present &= ~0x40;

        if(mag_squared8 > (mag_squared5+mag_squared6+mag_squared7+1)<<3)
        {
            sig_present |= 0x80;
            dtmf_index += 3;
            set2 += 1;
        }
        else
            sig_present &= ~0x80;
    }
    else                                    // 2nd harmonics calculation complete
    {
        // If the sum of all 2nd harmonics energy is greater than a threshold
        // value, we assume that the signal wasn't a pure DTMF.
        if(mag_squared1 + mag_squared2 + mag_squared3 + mag_squared4 +
           mag_squared5 + mag_squared6 + mag_squared7 + mag_squared8 > 125)
        {
            sig_present = 1;
        }
    }

//-----------------------------------------------------------------------------//
//                                                                             //
//                                BLOCK 6                                      //
//                              DTMF Output                                    //
//                                                                             //
//-----------------------------------------------------------------------------//

    // If both windows have completed, there is 1 and only one of
    // the row tones and 1 and only one of the column tones,
    // and there are no harmonic frequencies present, then print
    // the proper character to the terminal.
    if((sig_present == 0)&&(base_freq)&&(temp_set1 == 1)&&(temp_set2 == 1))
    {
        CR = 0;                             // Stop PCA0 timer
        pca_val = PCA0L;                    // Read PCA0 value
```

```
        pca_val += PCA0H*256;
        CR = 1;                             // Start PCA0 Timer



        // Compute the timpe elapsed since last reset

        temp_sec = second;
        temp_min = minute;
        temp_hour = hour;

        pca_temp = pca_val / 4096;
        temp_sec += pca_temp;
        pca_temp = pca_val - pca_temp*4096;
        hundredth = ((unsigned long)pca_temp*100)/4096;

        if(temp_sec >= 60)
        {
            temp_sec -= 60;
            temp_min++;

            if(temp_min >= 60)
            {
                temp_min = 0;
                temp_hour++;

                if(temp_hour >= 5)
                {
                    temp_hour = 0;
                }
            }
        }

        // Send the decoded character and the time elapsed since laste reset
        // on UART
        UART_display(dtmfchar[temp_dtmf_index]);

        for(i=0; i<100; i++)
            Delay();

        // Display the decoded character on the 7-segment cell
        Display(display_codes[temp_dtmf_index]);
    }
}


//-----------------------------------------------------------------------------
// DTMF_Init
//-----------------------------------------------------------------------------
//
// Various variable initializations
//
void DTMF_Init(void)
{
```

```
    new_tone = 1;
    done = 0;
    base_freq = 1;
    gain_cnt = 0;
    gain_calc = 1;
    sample_no = 0;
    set1 = 0;
    set2 = 0;
    low_x = 255;
    high_x = 0;
    max_sample = 200;

    TR0 = 1;                                // Start Timer0
    TR2 = 1;                                // Start Timer2

    Display(0x01);                          // Display the decimal point
}


//-----------------------------------------------------------------------------
// OSC_Change
//-----------------------------------------------------------------------------
//
// Switches system clock to internal or external oscillator
//
void OSC_Change(char n)
{
    unsigned int i;

    RSTSRC &= ~0x04;                        // Disable Missing Clock Detector

    if(n==EXT_OSC)                          // Switch to external oscillator
    {
        for(i=0;i<255;i++);                 // Wait for external osc. to stabilize
        while(!(OSCXCN | 0x80));
        OSCICN |= 0x08;                     // System clock = external oscillator
        OSCICN &= ~0x04;                    // Turn off internal osc.
    }
    else                                    // Switch to internal oscillator
    {
        OSCICN |= 0x04;                     // Turn on internal osc.
        while(!(OSCICN & 0x10));
        OSCICN &= ~0x08;                    // System clock = internal oscillator
    }

    RSTSRC |= 0x04;                         // Enable Missing Clock Detector
}


//-----------------------------------------------------------------------------
// Idle
//-----------------------------------------------------------------------------
//
//  This routine turns off all active digital and analog peripherals (except for
```

```
//  Comparator0, used to wake-up the system), switches the system clock to external
//  crystal and puts the microcontroller in Idle Mode.
//
void Idle()
{
    EA = 0;                                 // Disable all interrupts

    TR0 = 0;                                // Stop Timer0
    TR1 = 0;                                // Stop Timer1
    TR2 = 0;                                // Stop Timer2

    AD0EN = 0;                              // Disable ADC0

    EIE1 |= 0x20;                           // Enable CPO rising edge interrupt

    OSC_Change(EXT_OSC);                    // Switch to external oscillator

    EA = 1;                                 // Enable all interrupts

    PCON |= 0x01;                           // Switch to idle mode
}



//-----------------------------------------------------------------------------
// Display
//-----------------------------------------------------------------------------
//
// Send a character to the Serial Input Parallel Output buffer that drives the
// 7-segment LED display
//
void Display(unsigned char char_code)
{
    unsigned char i;

    for(i=0x01; i!=0x00; i=i<<1)
    {
        CLK = 0;                            // Hold clock low
        if( (char_code & i) != 0 )
            DATA = 1;                       // Send a 1 to data line
        else
            DATA = 0;                       // Send a 0 to data line

        Delay();                            // Wait 250us
        CLK = 1;                            // Clock high
        Delay();                            // Wait 250us
    }

    DATA = 1;                               // After the character is sent, pull
    CLK = 1;                                // clock and data high
}


//-----------------------------------------------------------------------------
```

```
// UART_display
//-----------------------------------------------------------------------------
//
// Sends on UART the decoded character and the time elapsed since last reset.
//
void UART_display(unsigned char character)
{
    unsigned char temp;

    putchar(character);                     // Send the DTMF character
    putchar(‘ ‘);

    putchar(temp_hour+’0’);                 // Send hours
    putchar(‘:’);

    temp = temp_min/10;                     // Send minutes
    putchar(temp + ‘0’);
    temp = temp_min - temp*10;
    putchar(temp + ‘0’);
    putchar(‘:’);

    temp = temp_sec/10;                     // Send seconds
    putchar(temp + ‘0’);
    temp = temp_sec - temp*10;
    putchar(temp + ‘0’);
    putchar(‘.’);

    temp = hundredth/10;                    // Send hundredths of seconds
    putchar(temp + ‘0’);
    temp = hundredth - temp*10;
    putchar(temp + ‘0’);

    putchar(‘ ‘);
    putchar(‘ ‘);
}

//-----------------------------------------------------------------------------
// Delay
//-----------------------------------------------------------------------------
//
// Wait for about 250us
//
void Delay(void)
{
    TR0 = 0;                                // Stop Timer0
    TL0 = 0;                                // Reset Timer0
    TH0 = 0;
    TR0 = 1;                                // Start Timer0

    while(TH0 < 2);                         // Wait 250us

    TR0 = 0;                                // Stop Timer0
    TL0 = TMR0RLL;                          // Reload Timer0
    TH0 = TMR0RLH;
```

```
    TR0 = 1;                                // Start Timer0
}


//-----------------------------------------------------------------------------
// end of dtmf.c
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Copyright 2004 Silicon Laboratories, Inc.
//
// FILE NAME      : dtmf.h
// TARGET DEVICE  : C8051F300
// CREATED ON     : 30.04.2004
// CREATED BY     : SYRO
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------
#define SYSCLK       24500000            // SYSCLK frequency in Hz
#define XTALCLK      32768               // XTALCLK frequency in Hz
#define BAUDRATE     9600                // UART0 baud rate
#define SAMPLE_RATE  8000                // Sample rate for ADC0 conversions
#define XMIN         20                  // Threshold value for the difference
                                         // between two succesive samples

#define EXT_OSC      0                   // Constants used by OSC_Change
#define INT_OSC      1                   // routine


//-----------------------------------------------------------------------------
//Structures and Types
//-----------------------------------------------------------------------------
struct ltype                             // In order to avoid truncation in the
{                                        // filter feedback stage, a long
    char space;                          // multiply and a divide are necessary.
    int intval;                          // This structure is used to avoid the
    char trunc;                          // long division.
};

typedef union ULONG                      // When the filter terms are calculated,
{                                        // the desired result ends up in the two
    long l;                              // middle bytes of the long variable.
    struct ltype hold;                   // Rather than divide by 256, the code
} ULONG;                                 // accesses the middle two bytes directly.


typedef union UINT                       // This type is used in much the same way
{                                        // as the previous one.  Rather than
    int i;                               // dividing the value to scale it down,
    char b[2];                           // the code accesses the high byte, which
} UINT;                                  // is equivalent to divide by 256.
```

```
typedef union LNG                       // Same as UINT but modified to access
{                                       // the high 16 bits of a long result.
   long l;
   int i[2];
} LNG;


//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F30x
//-----------------------------------------------------------------------------
sfr16 TMR2     = 0xcc;                  // Timer2 counter
sfr16 TMR2RL   = 0xca;                  // Timer2 reload value


//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------
char code dtmfchar[16] =                // DTMF characters
{
    '1','2','3','A',
    '4','5','6','B',
    '7','8','9','C',
    '*','0','#','D'
};


char code display_codes[16] =           // Character codes for the 7-segment
{                                       // LED display
    0x60, 0xDA, 0xF2, 0xEE,
    0x66, 0xB6, 0xBE, 0x3E,
    0xE0, 0xFE, 0xF6, 0x9C,
    0x6E, 0xFC, 0x3A, 0x7A
};


code int coef1[2] = { 235, 437};        // Goertzel filter coefficients
code int coef2[2] = { 181, 421};
code int coef3[2] = { 118, 402};
code int coef4[2] = {  47, 378};
code int coef5[2] = {-165, 298};
code int coef6[2] = {-258, 255};
code int coef7[2] = {-349, 204};
code int coef8[2] = {-429, 146};


ULONG Qhold;                            // Temporary storage for fiter
                                        // calculations

ULONG Qthold;                           // Saves previous value for magnitude
                                        // calculations

UINT Q1[3];                             // These are the elements of the 8
```

```
UINT Q2[3];                             // Goertzel filters.  The filters are
UINT Q3[3];                             // 2 pole IIR filters and so require
UINT Q4[3];                             // 3 terms each.
UINT Q5[3];
UINT Q6[3];
UINT Q7[3];
UINT Q8[3];

idata UINT Qt1[2];                      // These 2 element arrays are used for
idata UINT Qt2[2];                      // storing the low two elements of the
idata UINT Qt3[2];                      // filter elements after N filter
idata UINT Qt4[2];                      // iterations.
idata UINT Qt5[2];
idata UINT Qt6[2];
idata UINT Qt7[2];
idata UINT Qt8[2];

idata int mag_squared1, mag_squared2;   // Store output of the Goertzel filters.
idata int mag_squared3, mag_squared4;
idata int mag_squared5, mag_squared6;
idata int mag_squared7, mag_squared8;

unsigned char TMR0RLH, TMR0RLL;         // Timer0 reload value

bit new_tone;                           // Flag for valid pause between tones

bit start_goertzel;                     // Flag for start of the decoding
                                        // process

bit gain_calc;                          // Flag for gain computing

bit done;                               // Flag for starting character decoding

char base_freq;                         // Flag for base frequencies /
                                        // 2nd harmonics

int x;                                  // Current signal sample

int x_old;                              // Former signal sample

int high_x, low_x;                      // Minimum and maximum value of the
                                        // signal
int delta_x;

int gain;                               // Gain computed in AGC block

unsigned char gain_cnt;                 // Gain stage sample counter

unsigned char sample_no;                // Sample counter

unsigned char max_sample;               // Total no. of samples for base freqs
                                        // and 2nd harmonics

unsigned char sig_present;              // For every frequency detected, a bit
```

```
                                        // in sig_present is set

unsigned char dtmf_index;               // Index for dtmfchar array

unsigned char set1;                     // Hold the no. of freqs. detected
unsigned char set2;


unsigned char hour;
unsigned char minute;
unsigned char second;
unsigned char hundredth;

unsigned char temp_hour;
unsigned char temp_min;
unsigned char temp_sec;


//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------
void SYSCLK_Init(void);                 // System clock configuration

void PORT_Init(void);                   // I/O port configuration

void Timer0_Init(int counts);           // Timer0 configuration

void Timer2_Init(int counts);           // Timer2 configuration

void ADC0_Init(void);                   // ADC0 configuration

void CP0_Init(void);                    // Comparator0 configuration

void UART0_Init(void);                  // UART0 configuration

void PCA0_Init(void);                   // PCA0 configuration

void Interrupt_Init(void);              // Interrupt configuration

void DTMF_Detect(void);                 // Compute signal energy and
                                        // decode DTMF characters

void DTMF_Init(void);                   // Variable initializations

void Idle(void);                        // Switches to Idle Mode

void OSC_Change(char n);                // Changes the system clock

void Display(unsigned char char_code);  // Displays a character on the
                                        // 7-segment LED display

void UART_display(unsigned char character);

void Delay(void);                       // Waits for 250us
```

```
//-----------------------------------------------------------------------------
// end of dtmf.h
//-----------------------------------------------------------------------------
```

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.

4635 Boston Lane
Austin, TX 78735
Email: MCUinfo@silabs.com
Internet: www.silabs.com

SILICON LABORATORIES