# DTMF - Decoding with a 1-bit A/D converter

The purpose of DTMF decoding is to detect sinusoidal signals in the presence of noise. There are plethora of cost effective integrated circuits on the market that do this quite well. In many (most ?) cases, the DTMF decoder IC interfaces with a microcontroller. In these instances, why not use the microcontroller to decode the sinusoids? Well the answer is because the typical microcontroller based decoder requires an A/D converter. Furthermore, the signal processing associated with the decoding is usually beyond the scope of the microcontroller's capabilities. So the designer is forced to use the dedicated IC or upgrade the microcontroller to perhaps a more costly digital signal processor.

However there is yet another way to decode DTMF signals with a microcontroller and a 1-bit A/D converter (i.e. comparator). The theory is quite similar to the "classical" signal processing technique alluded to above. So let's briefly consider what is involved there before we continue.

One brute force way to detect DTMF signals is to digitize the incoming signal and compute 8 DFT's (discrete fourier transforms) centered around the 8 DTMF composite frequencies. DFT's are preferred over FFT's because the frequencies are not equally spaced (in fact they are logarithmically spaced). In its simplest form, the DFT goes something like so:

$$DFT(x) = \sum_{k=0}^{N} x(k) * W(k)$$

where x(k) are the time samples and W(k) is the infamous kernal function:

$$W(k) = e^{j*2*\pi*f*k/N} = \cos(2*\pi*f*k/N) + j*\sin(2*\pi*f*k/N)$$

All this says is that we multiply the samples by sine waves and cosine waves and add them together. And when you're done, you will end up with 8 complex numbers. The magnitudes of these numbers tell us roughly how much energy is present for each frequency of the input signal. In other words, we have computed the frequency spectrum at the 8 DTMF composite frequencies.

The reason this works so well is because of the "orthogonality" of the sine waves. In other words, if you performed the DFT on two sine waves:

$$DFT = \sum \sin(f_1*t) * \sin(f_2*t)$$

You will get a "large" number if the two frequencies are the same and a "small" number or zero if they're different.

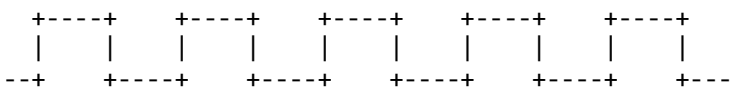
Now, I realize if you have never seen this before then you probably have no idea what I'm talking about and if you have you're probably wandering why left out so much important stuff. But the main point I'm trying to make is to show how the DFT concept can be generalized to non-sinusoidal signals; square waves in particular.

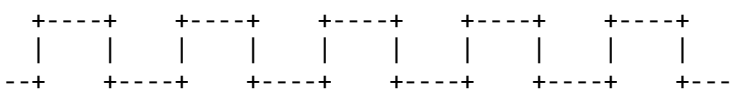
## "DFT" With Square waves:

The orthogonality concept applies equally well to square waves too. In fact, it's even easy to illustrate with

ASCII art! Consider the two examples:

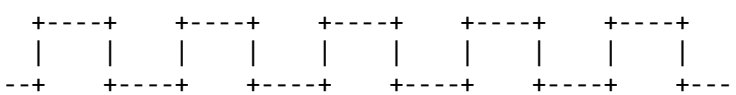
Ex 1

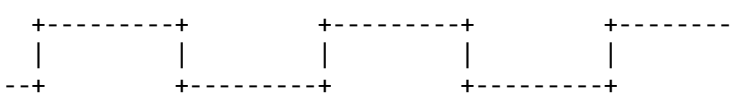




```
+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1
=> 25
```

Ex 2





```
+1+1+1+1-1-1-1-1-1+1+1+1+1+1-1-1-1-1-1+1+1+1+1+1+1-1
=> 2
```

In the first example, the two square waves have the same frequency and phase. When the individual samples are "multiplied" and summed together, you get a large number: 25. In the second case, the square waves differ in frequency by a factor of two. And as expected, when you "multiply" the individual samples and add them up you get a small number: 2.

If you look closely, you'll notice that the multiplication is really an exclusive OR operation.

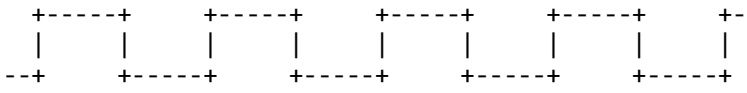
In PIC parlance,

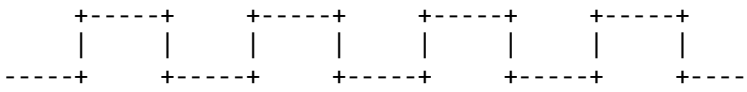
```
sum_of_products    equ  0x20   ;A register variable
input_sample       equ  0x21   ;In LS bit
square_wave_sample equ  0x22   ;In LS bit

        movf    input_sample,W
        xorwf   square_wave_sample,W
        skpnz
         movlw  -2
        addlw   1
        addwf   sum_of_products,F
```

## Quadrature:

In the DFT we used both sine and cosine waves. The two are obviously related by 90 degree phase shift. An analogously shifted square wave is needed for the DTMF decoding too. The reason is that it's possible to end up with a small sum-of-products even if the two waveforms have the same frequency. For example,





```
+1-1-1+1+1-1+1-1-1+1+1-1+1-1-1+1+1-1+1+1-1-1+1+1-1+1-1-1+1+1
=> 0
```

So to protect against this situation, we must perform two sum-of-products operations. One between the digitized input signal and a square at the detection frequency, the other between the digitized input signal and a square wave shifted 90 degrees with respect to the first square wave.

## Dot Product

The DFT operation is a dot product operation. You can imagine the signal and the kernals as vectors whose indices are the sample number. The vectors could be very large, e.g. 4096 samples.

## Signal Strength: 1-norm

If we were computing DFT's with sine and cosines, then the signal strength of a particular frequency is easily ascertained by finding the familiar magnitude:

```
strength ~ sqrt( real(DFT)^2 + imaginary(DFT)^2)
```

In other words, the result of a DFT is a complex number when the complex kernal is used. And the magnitude of a complex number is the square root of the sum of the real part squared plus the imaginary part squared. This is a cumbersome operation that I would rather advoid. But if you wish to consider it, then check out the square root theory and square roots with a PIC pages.

The square root of the sum of the squares normalization is called the "square norm". It is a subset of the general class of normed linear spaces called the "p-norm". In our case, the linear space consists of two components: the real and imaginary parts of the DFT. The p-norm for our case is:

```
strength ~ (abs(real(DFT))^p + abs(imaginary(DFT))^p) ^ 1/p
```

And this is the same as the square norm when p is 2.

Now if p is 1 we up end with an extremely simple formula, the 1-norm:

```
strength ~ abs(real(DFT)) + abs(imaginary(DFT))
```

## Digitization: +1 and -1 OR +1 and 0?

So far we've been digitizing to +1 and -1. However, in a real program we'll probably want to digitize to +1 and 0 since these are the numbers which microcontrollers are most comfortable. (If the subjective feelings of the microcontroller don't sway you, then read on.... there're some real efficiency reasons too). The next question is how does this impact the DFT calculations? Suppose we have two square waves that are digitized to + and - 1. What happens to their dot product if we digitize them to +1 and 0 instead? Let f1 and f2 be the two square waves.

f1 = -1 or +1

f2 = -1 or +1

Convert to 1's and 0's:

q1 = (f1 + 1)/2 => f1 = 2*q1 - 1

q2 = (f2 + 1)/2 => f2 = 2*q2 - 1

So, q1 and q2 represent the re-digitized f1 and f2 square waves. Now the dot product:

```
        ----            ----
DFT =   \     f1*f2 =   \     (2*q1 - 1) * (2*q2 - 1)
        /               /
```

```
      ----          ----
      ----
=     \    (4*q1*q2 - 2*q2 - 2*q1 + 1)
      /
      ----
       ----          ----       ----     ----
= 4 \   q1*q2 - 2 \   q2 - 2 \   q1 + \   1
     /            /          /        /
      ----          ----       ----     ----
```

The last term evaluates N. In other words, 1+1+1+ ... +1 = N. The middle two terms require a closer examination. Assume that f1 and f2 contain no DC component:

```
  N
----
\   f1  = 0
/
----
  0
```

Then the sum of q1 is:

```
  N          N
----       ----
\   q1  =  \    (f1 + 1)/2
/          /
----       ----
  0          0
       = 0 + N/2 = N/2
```

And similarly, the sum of q2 is N/2. Combining these results yields:

```
----              ----
\    f1*f2 =  4 * \    q1*q2 - 2*N/2 - 2*N/2 + N
/                 /
----              ----
                  ----
           =  4 * \    q1*q2 - N
                  /
                  ----
```

So the conclusion is that digitization with 0's and 1's is essentially the same as digitizing with +1's and -1's. The only differences are that a new "DC" term of N has been introduced and the dot product has been scaled.

**Question all assumptions**

I made the assumption that the square waves f1 and f2 have no DC component. But think about how theses square waves are created. One of them say f1 is the digitized DTMF signal, the other is software synthesized. We certainly (should) have control over the synthesized square wave. However, the other one is subject to the errors of reality. First, there's the issue that the comparator introduces an assymetry in the digitization. This can be remedied by a properly design comparator circuit. Second, there's the issue of having a sample window of finite width. For example, if the sampling window is 20 milliseconds wide and there is a 60Hz component present in our input signal we will introduce a DC term. In fact, if there is any frequency present that does not have an integer number of cycles over the sampling window a DC error term is present. There two ways to fix this "problem". The first is to make the window as wide as possible. This unfortunately slows the detection down. The other fix is to measure the DC component.

**Measuring the DC component**

The DC component of a square wave is really easy to measure. And no, you are not going to need an A/D converter. Consider a square wave that has a 50% duty cycle. If we were to sample this at a high rate (e.g. 10 times the frequency of the square wave), we should get the same number of high and low samples. The DC component in this case is one-half of the amplitude of the square wave. If more high samples were accumulated than low samples then the DC component would be larger. In other words, the duty cycle of the square wave is proportional to the DC component.

**Returning to the tone detection application...**

In our application, one of the q's in the dot-product is the 0/1 digitized input. The other is a software generated square wave. There will always be two dot-products for each tone; one for each quadrature.

**Applying the 1-norm to the +1 and 0 Digitization:**

Not complete... It's easy enough to apply the 1-norm formula to the new digitization.

# Error Analysis

A square wave is hardly equivalent to a sine wave. So the question naturally becomes: How much error does this technique introduce? Probably the best way to try to answer this question is by performing a harmonic analysis on the digitization process.

Consider the dual tone signal:

```
g(t) = A1*cos(w1*t) + A2*cos(w2*t)
```

The frequencies w1 and w2 are certainly different (otherwise we don't have a dual tone). The amplitudes are also different, but in many cases they are within a few percent of one another. Let's consider the case where the amplitudes are the same for right now:

```
g(t) = A1*cos(w1*t) + A1*cos(w2*t)
     = A1*(cos(w1*t) + cos(w2*t))
               /w1 + w2    \       /w1 - w2    \
     = A1* cos |------- * t| * cos |------- * t|
               \   2       /       \   2       /
```

This signal is digitized to 1's and 0's by a comparator. The digitization process can be mathematically described by:

```
gd(t) = (1 + sign(g(t)))/2
```

Where sign() is a function that returns the sign of its argument. sign() by itself produces the +1/-1 digitization. g(t) is positive when the two cosine terms have the same sign. Now consider the following sign() identity:

```
sign(f1(t) * f1(t)) = sign(f1(t)) * sign(f2(t))
```

and apply this to our formula

```
                 /    /w1 + w2    \\     /    /w1 - w2    \\
sign(g(t)) = sign|cos |------- * t|| sign|cos |------- * t||
                 \    \   2       //     \    \   2       //
```

The sign() function with a sinusoidal arguement produces a 50% duty cycle square wave with the same frequency as the sinusoid.

Not complete...

The Square Waves page provides even more theory.

And here's more software.

BACK HOME

---

*This page is maintained by Scott Dattalo You can reach me at: scott@dattalo.com.*
*Last modified on 17Dec99.*