A Sigma Delta ADC Simulator in “C”

By Dan P. Bullard
14 May 2011
Adapted from a Maxim Integrated Products Knowledge Base article.

Sigma-Delta ADCs are among the strangest things in the world of electronics. For years I have struggled to understand them. And yet I found that by simulating one and running some signals through it, I suddenly understand what all the fuss is about. I am hoping it will help you too. To follow along with this note it would be best if you were able to use a C compiler and an Excel compatible program to view the waveform files as a chart. You can get a full copy of the program from from me by emailing me at dan@danbullard.com

A (not so) quickie intro

First I highly recommend that you open and read Maxim Applications Note 1870, Demystifying Sigma Delta ADCs
Here is the URL for that apps note: http://www.maxim-ic.com/app-notes/index.mvp/id/1870
You only need to get through the first four pages, that’s good enough for now.
Basically, a Sigma Delta ADC works by converting an analog signal to a digital serial stream with no start, and no stop. You can grab any random chunk of the stream and the data will make sense. There is no MSB, or LSB, it’s just a stream of data, just like a sinewave is a serial stream of information. The stream is really fast compared to the digitized signal, that’s because it’s **oversampled**. That just means that it’s sampled much faster than the Nyquist Criterion. Nyquist found that you had to sample at least twice the rate of the fastest signal. Sigma Delta ADCs sample **much** faster than that, because it only uses a single bit, it has to do something with all the noise that comes from using only one bit. A one bit ADC gives you what I jokingly call CB quality audio (ten-four good buddy!). By oversampling the Sigma Delta ADC has a lot of places to put the noise, but more than that, it’s very architecture causes most of the noise to be moved high into the spectrum, out of the way of our signal. It’s called Noise Shaping. Assume we want to digitize audio for example, all you need to do is sample at 40KHz, since audio only goes to 20KHz, 40KHz satisfies good old Nyquist. However, any noise from the sampling process ends up in the band of DC to 20KHz, that makes it audible, and that is a problem. However, let’s say we **oversample** by 16 times by running our ADC at 640KHz. Our signal is still there at DC to 20KHz, but the ADC shoved most of the noise up above 20KHz so that now none of the noise is audible, to humans anyway (dogs might hear something). This noise shaping is inherent in the design, it is automatic, it comes free, so why not take advantage of it? The guys who figured this out must have been ecstatic when they realized what they had discovered.

Sigma Delta block diagram

Below is the block diagram of a **first order** Sigma Delta ADC stolen from the above named apps note. The term first order just means that you can cascade these things to get better performance, just like you can cascade amplifiers to get more gain. A fourth order ADC isn’t four times harder to understand and isn’t going to take you into the fourth dimension so don’t worry about it.

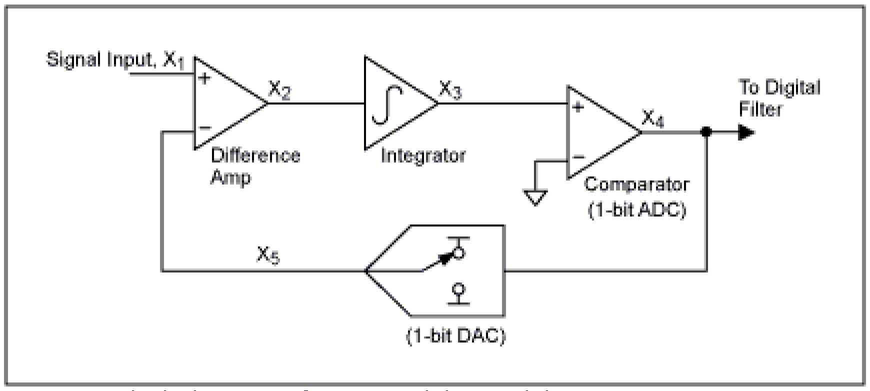
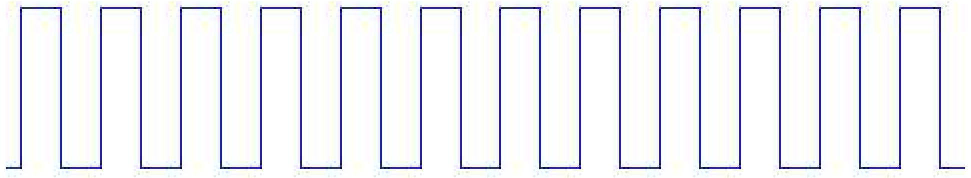


Figure 4. Block diagram of a sigma-delta modulator.

The signal comes into a difference amplifier where the difference between it and the output of a one bit DAC is. Let's assume that the 1 bit DAC is putting out +1V at node X5, and that the input voltage (node X1) is zero. Then the voltage on node X2 is –1v (0.0-1.0 = -1.0). That's going to go to the integrator which is just going to provide a way to accumulate a sum and provide a time delay so things slow down and stick around long enough for this to not be a gigahertz oscillator. The output of the integrator will fall down to minus 1V (–1.0V) at node X3, since 0.0 + (-1.0) is –1.0. (This is not a capacitive integrator, this integrator just sums or integrates over time so we don't have to fool around with 63% of this or that) This –1.0 at node X3 is lower than ground so the comparator puts out a low. That low will tell the DAC to put out –1V. Assuming that the input didn't change and is still 0V, the difference amplifier is going to put out +1V (0.0- (-1.0) = +1.0). Of course that is going to cause the integrator output to go up to 0.0V, since it was at –1V, add 1V you get 0V. The comparator may trip, or may not. If it doesn't trip this time, it sure will next time as the DAC will still be putting out -1V, and 0V – (-1V) is +1V, which will trip the comparator. The practical upshot is this: If you continue with this analysis you will see that when the when the ADC input is at mid-scale (0V in this example) the node at X4 (and X5) just toggles. In other words, it looks like this:

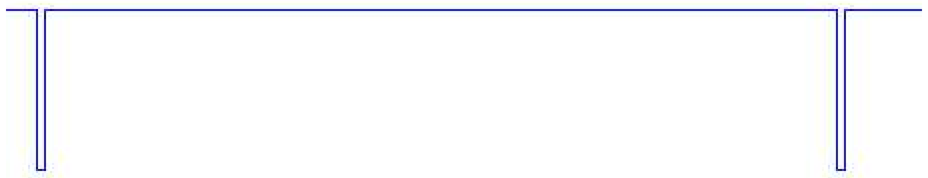


Sigma Delta output with mid scale input.

What does it do when the input voltage is full scale? Again, referring to the diagram above, assume that the output of the integrator is zero, and that the DAC is putting out +1V just like we did last time. Since the difference amp outputs the difference between the input and the DAC, it's output will be zero. That means that 0V is added to the 0V already in the integrator, so it just sits at zero. The comparator gets zero, so it could go either way. If you assume it just continues putting out a high (it had to be high previously, since the DAC was putting out +1V) then nothing changes. The integrator doesn't gain anything, so the comparator continues to put out a high, so the DAC continues putting out +1V, which means that the integrator accumulates no extra voltage, and so on.
If you assume that the comparator doesn't like 0V and flips to putting out a low, that makes the DAC put out –1V, now there is a big difference between the input at +1V and the DAC output at –1V. So the integrator gets the difference of (+1-(-1) = 2V). So now the integrator has +2V going to the comparator,
which is plenty positive to cause it's output to go high, so the DAC puts out a +1V. But again, there is no difference between the input and the +1V. Therefore the integrator neither accumumates extra charge, nor does it lose any charge, it just sits there putting out a constant 2V into the comparator, which continues to put out a high, which causes the DAC to continue to put out +1V. In other words it's a standoff, nothing changes so the output looks like this.

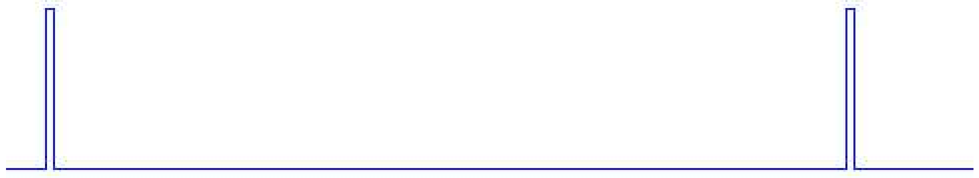
Sigma Delta output with Positive Full Scale in

Pretty boring, huh?
But what happens if the input is just 0.98V, that is 1% lower than positive full scale? (Notice the input of this hypothetical device ranges from +1V to –1V or 2Vpp making 0.98V 1% lower than +FS). Rather than running through a step by step analysis, let me show you.



Sigma Delta output with 99% Full Scale in

What you see here is a digital signal is that is high 99% of the time and low 1% of the time. In other words the duty cycle of the digital output tells you what the input level is. Remember what we got out with 0V in? That's right, a 50% duty cycle waveform. If the input went to –0.98V the output would look like this.



Sigma Delta output with 99% of Negative Full Scale in

This digital signal is high for 1% of the time, and low 99% of the time. Amazing huh? Now, before we put any real signals into our Sigma Delta ADC, let's see how it's implemented in C.

Simulating a Sigma Delta ADC

Remember the diagram for the Sigma Delta ADC?

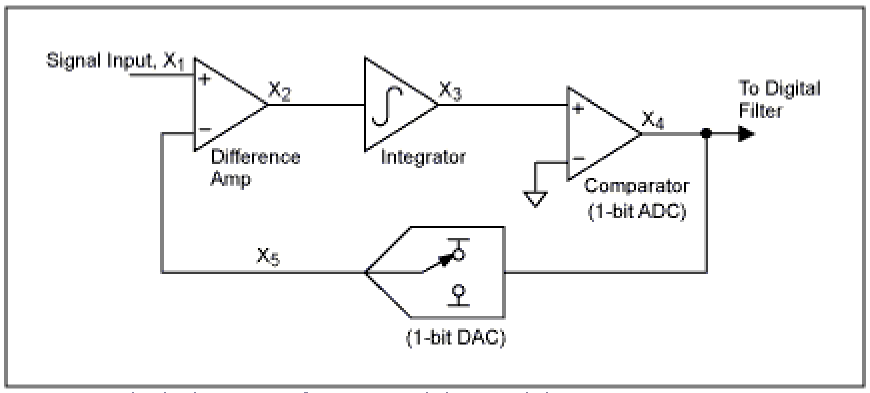


Figure 4. Block diagram of a sigma-delta modulator.

What does that look like in C code? Get ready, it's really simple:

```
void sigma_delta(float *wavein, float *waveout, int mysamples)
{
     int i; float integrator = wavein[0];
     for(i=0; i<mysamples; i++)
     {
          if(i > 0) integrator += (wavein[i] - waveout[i-1]);
          if(integrator > 0.0) waveout[i] = 1.0; else waveout[i] = -1.0;
          if(i < 20) printf("integrator = %f\n", integrator);
     }
}
```

I implemented it with a function that takes three parameters, the first is the input waveform, the second is the output waveform, and the last is just the number of samples in the input waveform. We start off by setting the integrator to the value in the first element of wavein. The integrator is supposed to be accumulating history, but we have none, so the best we can do is set it to the first sample.
Now we drop into a loop. For sample zero we skip the actual integration process, because we can't look back into a non-existent history. Now, we implement the comparator function with an **if** statement. Notice that I combined the comparator and 1 bit DAC into a single function, the only reason it's two different functions in the block diagram is that the comparator's output levels might not be compatible with my ADC input range. In code, it's just easier to make them the same. That means in my code, a "1" is +1 and a "0" is –1.
Last in the loop I placed a printf so when the program is run we can watch the integrator in action. This is quite interesting to watch, at least until you believe. On the second pass we do the job of the differencing amplifier and the integrator, subtracting the last value of waveout (waveout[i-1]) from the current value of wavein (wavein[i]) and adding that to the value of integrator. Now the integrator has a history, and each step of the way it will accumulate the "sum of the differences" between the input and the last output. If you run the program with 0.0V going in, the printf shows the integrator in action.

```
integrator = 0.000000
integrator = 1.000000
integrator = 0.000000
integrator = 1.000000
integrator = 0.000000
integrator = 1.000000
integrator = 0.000000
integrator = 1.000000
integrator = 0.000000
```

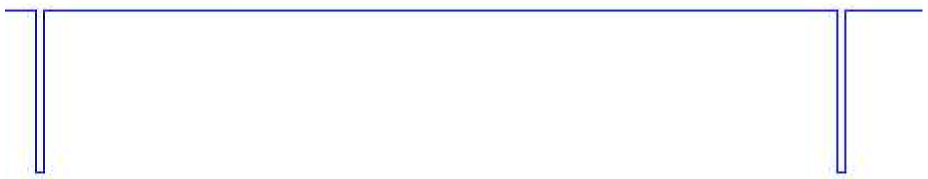
But watch what happens when I put in 0.98 (1% lower than full scale).

```
integrator = 0.980000
integrator = 0.960000
integrator = 0.940000
integrator = 0.920000
integrator = 0.900000
integrator = 0.880000
integrator = 0.860000
integrator = 0.840000
integrator = 0.820000
```

You can see the difference amplifier in action, subtracting the difference between the input and the +1V that is more than likely coming out of the DAC.The sum loses 0.02 (the difference) each iteration of the loop. After 50 or so iterations, the integrator output is going to go below zero isn't it? Let's look at what happens then. Alongside I logged the value of waveout:

```
integrator = 0.080001, waveout[45] = 1.000000
integrator = 0.060001, waveout[46] = 1.000000
integrator = 0.040001, waveout[47] = 1.000000
integrator = 0.020001, waveout[48] = 1.000000
integrator = 0.000001, waveout[49] = 1.000000
integrator = -0.019999, waveout[50] = -1.000000  here is a single low!
integrator = 1.960001, waveout[51] = 1.000000
integrator = 1.940001, waveout[52] = 1.000000
integrator = 1.920001, waveout[53] = 1.000000
integrator = 1.900001, waveout[54] = 1.000000
```

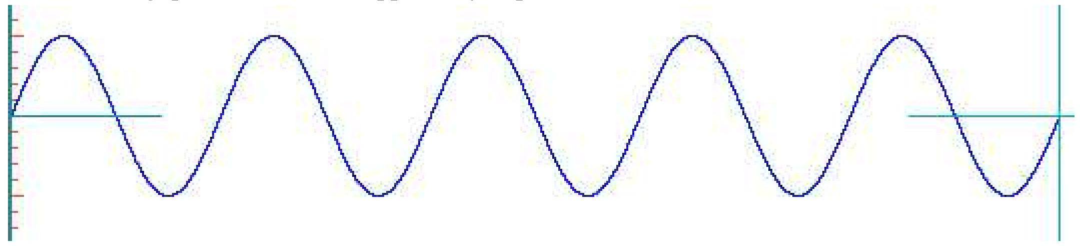
Remember the waveform we saw with 0.98V in?



The digital output was high 99% of the time, then it went low once, then stayed high for another 99 counts. The only reason mine went low after only 50 counts is because I didn't have that accumulated history (remember?). But after it gets going, the pattern continues forever, until the signal starts to move, then things get more interesting.
By the way, you might notice that the output of a Sigma Delta ADC can be it's own DAC. If you simply placed a low pass filter on the output of any of these signals, you would get the signal back. Or you can just use a digital filter, such as a simple averager. What's the average value of a signal that's at +1V for 99% of the time and –1V for 1% of the time? That's right, 0.98V!

Moving on to moving signals

So now the big question is, what happens if you put a sine wave into one. Well, first let's look at the sine wave.

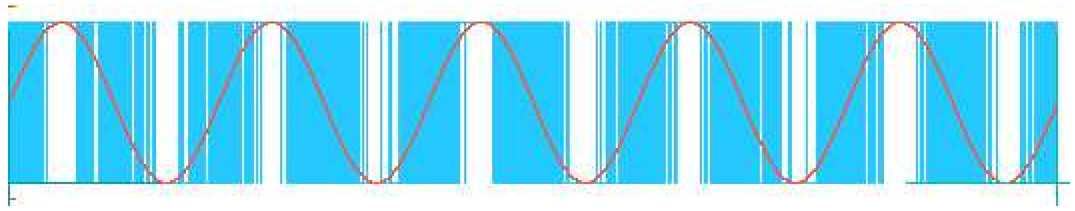


Five Cycle Sine Wave

This sine wave has five cycles. I made it using a simple formula below. Don't worry about the markers at the start and end, that's just part of the Credence AWT tool.
The sine wave was created with the code below, it's pretty simple, but you have to remember to include **math.h** otherwise the **sin** function isn't available. Remember that the sin functions returns values ranging from +1 to –1 and works in **radians** (hence the **2.0*PI**, a define). **Samples** is a define set to **65536** in this example. The **5.0** in the formula defines that I will get a **5 cycle** sine wave. If I wanted **seventeen cycles** I would put a **17.0** in the place of the 5.0.

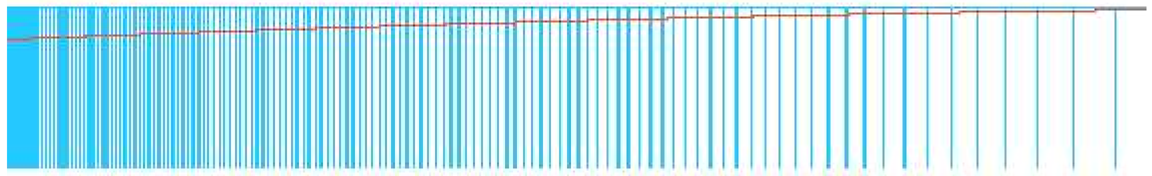
```
float mywave_real[samples]; /* generate a 5 cycle sinewave, whatever the
numerator is in the
formula controls the number of cycles */ for(i = 0; i<samples; i++) {
mywave_real[i] = (float)sin(2.0*PI*i*5.0/samples);
}
```

Now let's digitize it with the Sigma Delta convertor. Ready?



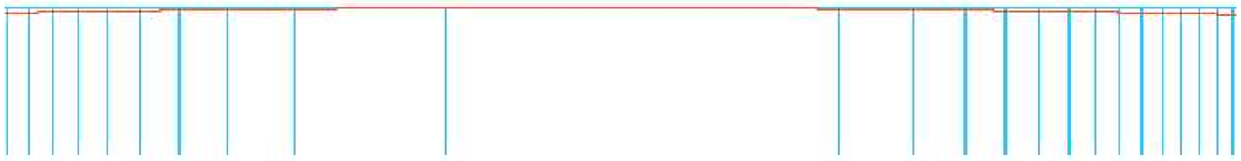
Sine wave Digitized by the Sigma Delta ADC (with sine wave overlaid in red)

The light blue waveform is the Sigma Delta waveform, and I have overlaid the sine wave in red. Notice that when the sine wave gets near the peaks, the digitized version goes nearly all high (for the high sine wave peaks) or nearly all low (for the low sine wave peaks). You might wonder what is happening in between. Let's look.



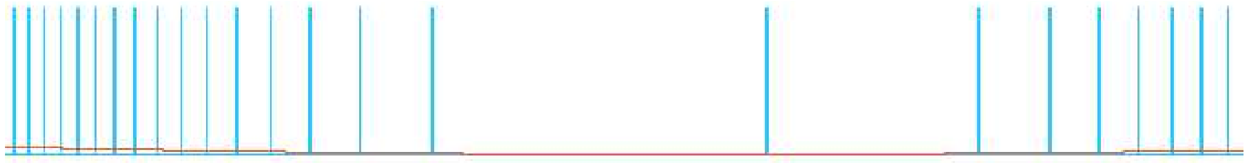
Close up of the overlaid waveforms

As the sine wave increases in amplitude the Sigma Delta output goes from having about half lows and half highs to having more and more highs. As we near the peak of the sine wave (in red still) we see even fewer lows, mostly just lots of highs.



Close up of the sine wave positive peak

Now as the sine wave descends you see the lows start to become more prevalent, and as we near a negative peak of the sine wave, we see the lows take over, at the expense of the highs.



Close up of the sine wave negative peak

So, the Sigma Delta ADC is just busily digitizing the signal, hardly aware that it's moving. The difference amplifier notices that the input voltage changed slightly, but it can easily keep up since it's digitizing far faster than Nyquist requires.

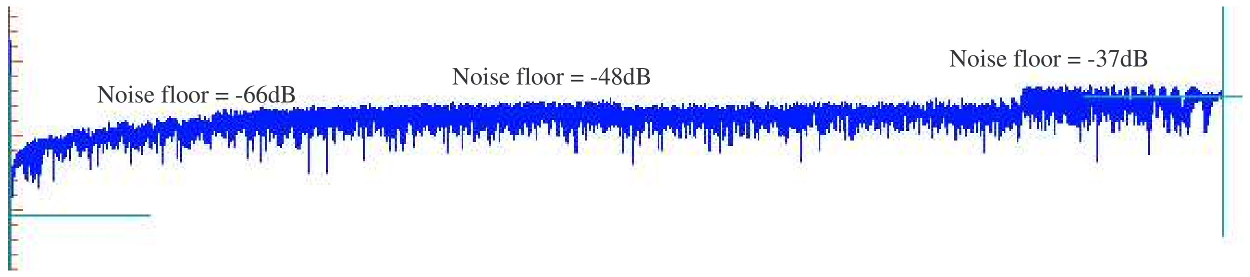
Yeah, I get it, so what?

OK, so the output of a Sigma Delta ADC is no longer a mystery, good for you. Why is it such a big deal? One of the biggest reasons why you would want to use a single bit ADC is that it is **linear** by definition. A single bit ADC has only one comparator, so it has to be linear. Compare that to a typical flash ADC which has a bunch of resistors and comparators, any mismatch of any resistor or any comparator and linearity is shot. But a single bit ADC is linear in both **Integral Linearity** and **Differential Linearity**. Each step is always the same and the +FS to –FS transfer curve is absolutely straight. They only problem is the **quantization** noise. By definition a single bit ADC has only 6.02dB + 1.72dB of SNR and an abysmal noise floor. If it weren't for that one fact a single bit ADC would be great.

However, this is kindred to saying that a car with four flat tires would be better if one tire was good. But don't give up hope, we haven't looked at the spectrum yet.

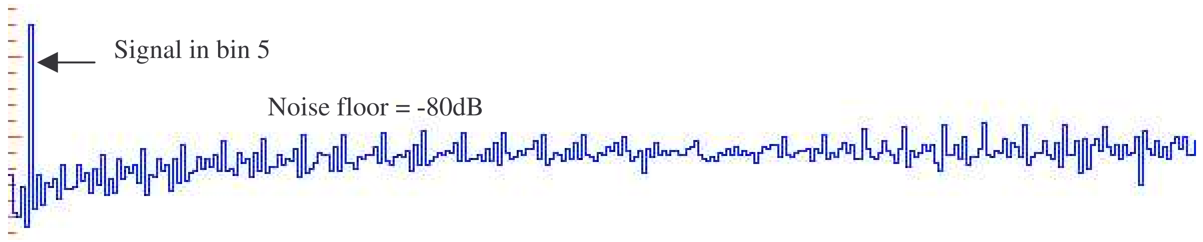
Noise Shaping and the signal within

Let's just take a look at the spectrum of the Sigma Delta digitized 5 cycle sine wave.



Sigma Delta ADC sine wave spectrum

Here is the spectrum, and at the far left is the signal, which is nearly invisible. But now you can see the **noise shaping**. Notice how the noise is much lower on the left side of the spectrum, and it gets higher and higher the further we go up in frequency. This is the noise the dogs would hear if we go back to our audio ADC example. People wouldn't be able to hear much noise because human ears act like a low pass filter. Let's look lower in the spectrum and see what the noise looks like down there:

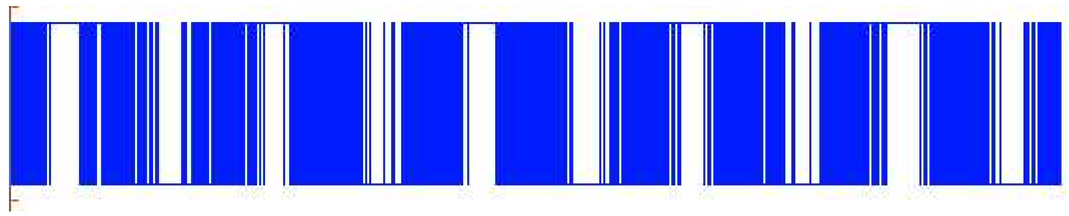


Sigma Delta ADC sine wave spectrum close up

Down in this area you can see that the noise floor is nearly CD quality (better than 90dB) rather than CB quality, despite the fact that we only used a single bit ADC. Where is all the noise? It got shoved up higher into the spectrum, where only dogs can hear it. If we can chop the spectrum down to only this area, we don't have to worry about it. One way to do that is to filter the waveform.

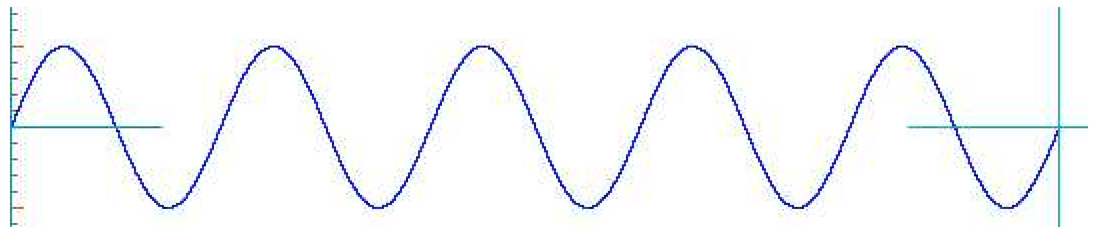
Now for my final trick!

So what happens if we filter the original Sigma Delta waveform? Ready:



Original five cycle Sigma Delta digitized sine wave

Here is original five cycle Sigma Delta digitized sine wave, and, **ta-da**:



Filtered Sigma Delta digitized sine wave

If you just apply a low pass filter to the waveform, it reconstructs the signal, just like a low pass filter reconstructed the original DC levels early on. The filter can be a real RC filter, or it can be a digital filter.
Along with just filtering, we can also use filters to reduce the data rate. Anytime you reduce the data rate it's called **decimation**, and since the Sigma Delta ADC sample rate was very high, the **Decimation filter** reduces the data, and thus the data rate, and it filters out all that noise that was sent high into the dog spectrum.

Conclusion
Sigma Delta ADCs can be scary just like any new technology, but hopefully you understand more about it now. If you do your own experiments using the program (below) you can discover how it works by yourself. The program creates two files, sinewave.csv and sigmawave.csv. Either can be opened with Excel or any Excel compatible program that can plot data.

If you have any questions please let me know at dan@danbullard.com

```
/*
 * File:   main.cpp
 * Author: DanBullard
 *
 * Created on April 24, 2011, 12:13 PM
 */

#include <cstdlib>
#include <stdio.h>
#include <iostream>
#include "math.h"
#include "string.h"

#define PI 3.14159265
#define samples 256

using namespace std;

/* this function writes the waveform to a CSV file, compatible with Excel */

void write_to_csv(float *wave, int size, char *name)

{
     FILE *csv_file;
     int i;
     char filename[80];
     strcpy(filename, name);
     strcat(filename, ".csv");
     if( (csv_file = fopen(filename, "w")) == NULL)
     {
          printf("couldn't open file %s\n", filename);
          return;
     }
     for(i=0; i<size; i++)
     {
             if((i+1)%256 == 0)
          fprintf(csv_file, "%e\n", wave[i]);
            else
                 fprintf(csv_file, "%e,", wave[i]);
     }
     fclose(csv_file);
}

// dans attempt to code a simple sigma delta convertor
// This mimicks the block diagram on page 3 of Maxim app note 1870

void sigma_delta(float *wavein, float *waveout, int mysamples)
{
     int i;
     float integrator = wavein[0];

     for(i=0; i<mysamples; i++)
     {
          if(i > 0) integrator += (wavein[i] - waveout[i-1]);
          if(integrator > 0.0)
               waveout[i] = 1.0; else waveout[i] = -1.0;
     }
}

int main(int argc, char* argv[])
{
     unsigned i;

     float mywave_real[samples], sigma_delta_wave[samples];
     /* generate a 5 cycle sinewave, whatever the numerator is in the formula
controls the number of cycles */

     for(i = 0; i<samples; i++)
     {
          mywave_real[i]  = (float)sin(2.0*PI*i*5.0/samples);
     }

     sigma_delta(mywave_real, sigma_delta_wave, samples);
     char mystring[50] = "sinewave";
     write_to_csv(mywave_real, samples, mystring);
     strcpy(mystring,"sigmawave");
     write_to_csv(sigma_delta_wave, samples, mystring);
     printf("Hit <RETURN> to exit ");
     cin.get();
     return 0;
}
```