Introduction

Analog signal generation is still a mainstay technology of electronics even in an increasingly digital world. Analog signals find use in communication and radio systems, electronic test equipment and musical applications to name just a few application areas. Analog signals that were once created using discrete electronic components are more often than not produced digitally these days. It is no wonder analog signal generation has gone digital. Problems with component matching, temperature sensitivity and component value drift requiring periodic calibration of the analog circuitry are gone. Of course digital signal generation has its own set of issues including quantization and quantization noise but techniques for dealing with these issues are well understood.

Direct digital synthesis or DDS is a method for digitally generating analog signals. With DDS one creates a digital representation of the desired analog signal and then uses digital-to-analog conversion to produce it. DDS systems allow quick switching between output frequencies, fine frequency resolution and operation over a wide range of frequencies.

Although hardware DDS chips do exist, here we will be using software running on Arduino compatible microcontrollers to show DDS in operation. In this article we will explore a musical application of DDS by building an electronic music box. First some background on DDS to get us started.

A typical system for digital signal generation is shown in Figure One. Here, some integral number of cycles of the desired signal are stored as samples in a wavetable. Every sample clock increments the address counter which provides the address of the next sample in the wavetable. That sample is output to a digital-to-analog converter where it is converted to an analog voltage. A low pass filter (LPF) is generally employed to remove high frequency artifacts from the analog output that could result in aliasing. With this system, the frequency of the output waveform is controlled by the frequency of the sample clock; the faster the sampling clock the higher the frequency. This, however, is inconvenient because the cutoff frequency of the LPF would need to change with the change in the sampling clock, complicating the design.

Figure Two illustrates a basic DDS system which can produce different output frequencies without changing the frequency of the sample clock. It shares a lot of functionality with the previous technique though it differs in how the addresses for the wavetable are generated. Every sample clock the content of the frequency register is added to contents of the phase accumulator which in turn is used to address the wavetable. The output frequency of this configuration is controlled by the DDS tuning equation:

$$F_{out} = M * F_{SampleRate} / 2^{n}$$

where **M** is the content of the frequency register and **n** is the number of bits which make up the phase accumulator. The bit width of the phase accumulator is important because it determines the frequency resolution of a DDS system and is described by the following equation:

$$F_{SampleRate} / 2^{n}$$

If the phase accumulator is 32 bits wide the frequency resolution is one part in 4 billion allowing for very precise frequency control. What the factor M represents can best be seen in Figure Three where the circumference of the circle represents one complete pass through the wavetable typically containing one cycle of the desired output waveform. If M = 1, n = 32 and the sampling rate is 32000 samples/second, the output frequency would be 7.45×10^{-6} Hz.

In a practical system the size of the wavetable would be substantially smaller than the 32 bit width of the phase accumulator used to address it. In practice, a significant number of the least significant bits of the phase accumulator would be truncated resulting in a much smaller address space. Theory shows that this truncation does not affect frequency resolution but does adds a small amount of phase noise to the output.

If we have a 256 entry wavetable (requiring 8 bits of address) containing one cycle of a sin waveform and we are sampling at 32000 samples/second we have a base or fundamental frequency of 32000/256 or 125 Hz. If we wish to produce a 2200 Hz sin wave as output with n = 32 bits, the value of M would be expressed as:

$$2^{32} * \text{output frequency} / \text{sample rate}$$

or the rounded value of M would be 295,279,002 which in fact is a fixed point representation of the fractional value.

To generate a 2200 Hz sin wave in software we would need an interrupt service routine (ISR) running at the 32000 Hz sampling rate. Each time through the ISR, the value 295,279,002 would be added to the phase accumulator and then the phase value would be shifted to the right by 24 bits and the remaining 8 most significant bits would be used as the address into the wavetable for returning the sample to be sent to the digital-to-analog converter. With this number of bits and the sample rate specified our DDS oscillator would have a frequency resolution of 0.00000745058 Hz.

Another thing to note about DDS is that phase is preserved when the frequency is changed. This is especially important in musical applications where a major discontinuity in phase may be audible when output frequency or pitch is altered.

Digital to Analog Conversion

Once digital samples are available they must be converted into analog with some sort of digital to analog conversion. Numerous techniques exists for doing this including:

1. Using a discrete D to A converter chip
2. Building a D to A converter using an R2R resistor ladder network
3. Using pulse width modulation (PWM) with filtering

The technique employed depends upon the application. We will use technique number three for the electronic music box.

The Electronic Music Box

DDS can be used for the generation of periodic or non periodic analog signals. So let's have some fun and build an electronic music box. Our music box will emulate the sound of a mechanical music box which uses tuned metal tines plucked by pegs on a revolving drum. This arrangement produced single notes and multi-note chords which have a high harmonic content after being plucked but become more sine wave like over time. Also these notes decay in amplitude quickly giving music boxes their distinctive plucked sound.

We will build the electronic music box using just three components connected as shown in Figure Five:

1. An Arduino compatible microcontroller (Figure Four).
2. A 100 ohm 1/4 watt 5% resistor
3. A small 100 ohm speaker

This configuration works without the use of filtering because the frequency of the PWM signal and the chosen sample rate are so far above the frequency response of the speaker they cannot be heard. The volume the direct drive approach produces may be insufficient for some applications. If so, an amplifier can be added as also shown on the schematic.

Our music box can play three songs: Fur Elise, Twinkle, Twinkle Little Star and Greensleeves and is capable of playing three notes simultaneously. See Resources for the Arduino sketches. The sketch/code is to long to print so we will just discuss how DDS makes the electronic music box possible.

Truth be told, the idea of an electronic music box wasn't mine. I came upon the idea at http://elm-chan.org/works/mxb/report.html while doing DDS research on the Internet. While I did adopt techniques presented there, the implementation I provide with this article is entirely my own.

For something so simple in concept, the electronic music box code is surprisingly complex and took time to get working. Implementing three simultaneous voices (called sound generators in the code) each with their own attack, sustain and decay characteristics stretches the 8 bit micro controller to its real time limit. More voices would be possible if the code were written in assembler but I didn't want to go there just for a demo program.

Hardware Setup

The music box code takes advantage of hardware built into the ATMega chips. See Table One.

Table One – Electronic Music Box Hardware Usage

Micro Controller	Music Box Sketch	Timer 1	Timer 2	Timer 4
ATMega328 as used in the Arduino Uno	MusicBox.ino	Used to generate an interrupt at the 32000 Hz sample rate	Fast 8 bit PWM running at 62,500 Hz	Not Available
ATMega32U4 as used in SparkFun's Pro Micro	MusicBoxProMicro.ino	Used to generate an interrupt at the 32000 Hz sample rate	Unused	Fast 8 bit PWM running at 187,500 Hz

The timers are configured for operation in the ***setup()*** portion of the Arduino sketches.

Music Box Data

The music box requires lots of data. Most of this data was generated by a series of Java tools I wrote specifically for this purpose. See Resources. Table Two describes the music box data. NOTE: there is to much data to fit into the small amount of RAM available on these microcontrollers. For this reason much of the data is stored in program memory which requires special handling to access. See the code for details.

Table Two – Music Box Data Arrays

Data Array	Data Type	Usage Description
Song1Data in PROGMEM	uint16_t	Song data for Fur Elise. All song data was processed from MIDI files by MidiParser.java. Each note consists of two data items: note start time and midi note number. A note start time of zero indicates end of song.
Song2Data in PROGMEM	uint16_t	Song data for Twinkle, Twinkle, Little Star
Song3Data in PROGMEM	uint16_t	Song data for Greensleeves
MidiPitchData in RAM	uint16_t	The M values corresponding to the frequencies of the MIDI note numbers 0..127 in fixed point 8.8 format. Data generated by ScaleGenerator.java.
EnvelopeData in PROGMEM	uint8_t	8.8 fixed point numbers representing an exponentially decaying value (Figure Seven). This is used to add the decay dynamic to each music box note. Envelope data generated by

Data Array	Data Type	Usage Description
		EnvelopeGenerator.java. An envelope value of zero indicates end of decay envelope.
WaveTableData in PROGMEM	int8_t	Signed sample data for the attack (Figure Six) and sustain (Figure Eight) portions of the music box note waveform. Attack data generated by AttackGenerator.java; sustain data by SustainGenerator.java.

The music box uses other data as well. Most importantly the data that defines the sound generators. Three sound generators are required to play three simultaneous notes. A sound generator is defined by three variables:

1. The ***m*** value which contains M for the sound generator that defines which frequency is being produced.
2. The ***phaseAccumulator*** value to which ***m*** is added each sample time and which, after shifting, provides the address to look up in the wavetable.
3. The ***envelopeIndex*** value which controls the amplitude decay of the note a sound generator is playing.

A sound generator is selected for each note in a song. If the selected sound generator is busy playing a previous note, note playback is terminated and the new note sounds. Premature note termination can sometimes be heard especially when low frequency notes are being played. Increasing the number of sound generators helps with this problem but you soon run up against the real time processing limit of the processor.

Timing

Accurate timing is extremely important for music reproduction as the human ear is very sensitive to tempo and timing issues. This is true whether we are talking about a musician playing a real musical instrument or our music box playing a song. To this end the music box code establishes a time base based upon the execution of the 32000 Hz sample rate interrupt triggered by a hardware timer. Each time through this ISR a volatile 32 bit variable ***sampleCount*** is incremented. From ***sampleCount***, the program derives ***tickCount*** which controls the tempo of the song played. More on this in a moment.

Music Box Operation

With the majority of the data the music box uses described above we can now talk about how the program actually works. We will do this by describing the foreground and background processes separately.

The Foreground Process

The foreground process is the code that runs in the sketch's ***loop()*** function which repeats forever. This code is concerned with playing songs on a note by note basis and is therefore interested in which song is currently being played, when the next note needs to be scheduled and the frequency of the next note.

Basically the next note is fetched from a song's data array and its start time examined. If zero, the end of the current song has been reached and setup for the next song occurs. If non zero, the start time is compared to ***tickCount.*** If it is not yet time for the note to sound, the code loops waiting for the proper start time.

Once the note's start time is reached, a sound generator is assigned and initialized for this note's reproduction. Initialization consists of writing the M value for the MIDI note into the sound generator's ***m*** variable and clearing both its ***phaseAccumulator*** and ***envelopeIndex*** values. After the sound generator assignment is made the loop repeats waiting for the next note.

The Background Process

The background process is the real time code which runs in the ISR at the 32000 Hz sample rate. In this code every cycle is important and keeping time consuming operations like multiplications and divisions to a minimum are absolutely necessary. Fixed point calculations are used in the ISR because floating point operations on the fractional values would be to costly.

The first order of business in the ISR is the processing of the three sound generators. For each sound generator the ***phaseAccumulator*** value is fetched and shifted to produce the address of the next sample in the wavetable. Next the ***m*** value for this sound generator is added to the ***phaseAccumulator*** and a test is made to determine if all of the values in the wavetable have already been output. If so, the ***phaseAccumulator*** value is reset so that the sustain portion of the wavetable is set to repeat next sample time and a variable called ***decaying*** is set which controls note amplitude decay.

The sample from the wavetable and the value of the decay envelope are then fetched from program memory. These values are multiplied together and the result added to that of the other sound generators. Finally if the note is decaying, the index into the envelope data is incremented for next time.

After all the sound generators have been processed, the sum of all sample values is scaled into the appropriate range and set into the PWM timer's count register.

Some notes about sound generators are in order here.

1. Sound generators only run during the ISR
2. As a sound generator steps through the wavetable it produces a waveform that is initially high in harmonics (see Figure Six) but gets more sine wave like at the end of the attack period.
3. The sustain portion of the wavetable is repeated over and over unless a new note is assigned to the sound generator. Programmatic repetition of the sustain portion of the wavetable data was implemented to reduce the overall size of the wavetable.

4. The multiplication of the wavetable sample with the value of the envelope controls note decay. Note decay begins after the first full pass through the wavetable is completed and the ***decaying*** variable is set. Once the envelope data reaches zero the sound generator no longer contributes to the final PWM sample.

Conclusion

DDS is a powerful technique for analog signal generation that can be used in a variety of applications and which can be implemented in hardware or software. In this article we described the basic theory behind DDS and then went on to implement an electronic music box as an example application. In the second article in this series we will use DDS to build a multi-waveform audio frequency function generator. Until then, enjoy your new electronic music box.

Figures

Figure One – Basic Wavetable System

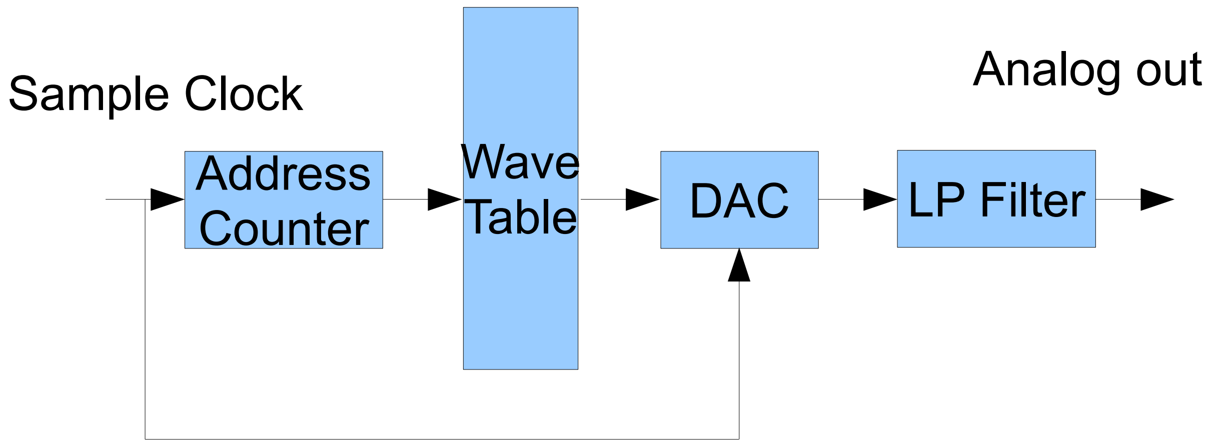


Figure Two – Basic DDS System

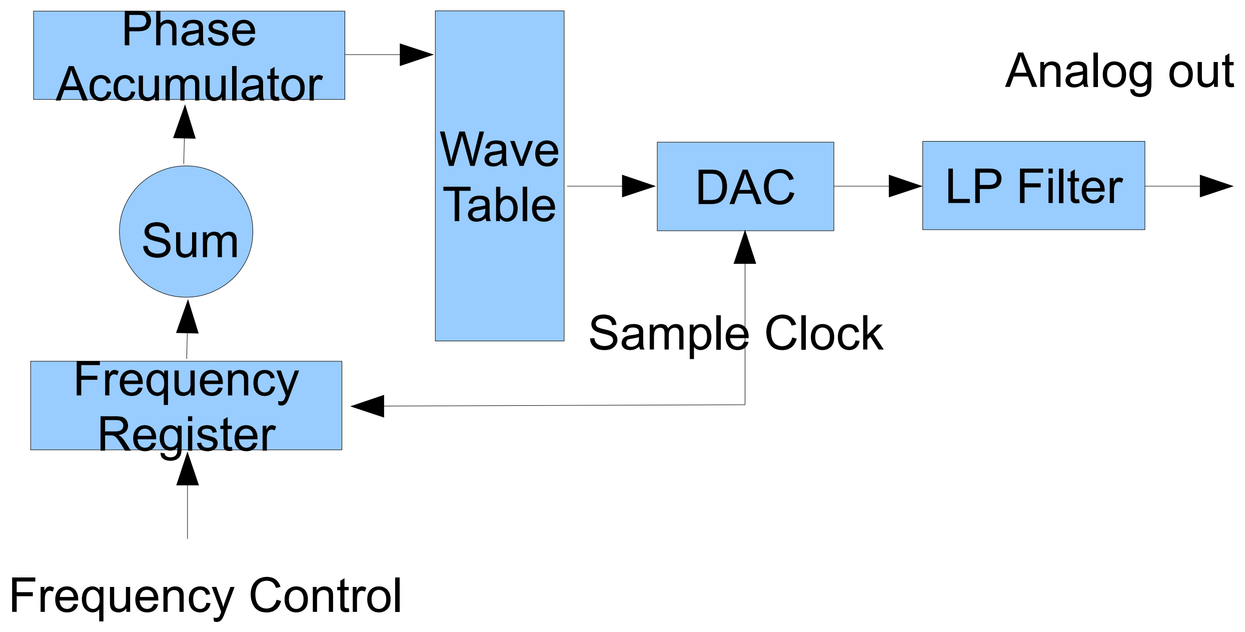


Figure Three – The value of M determines how fast we move around the circle

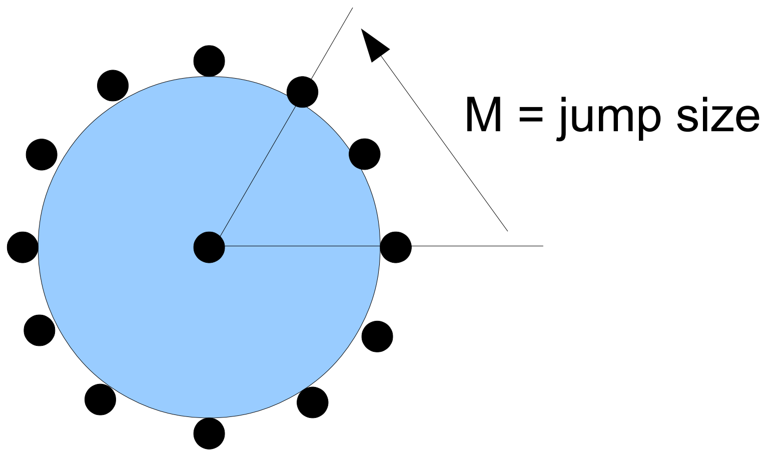


Figure Four – SparkFun's Pro Micro board based on the ATMega32U4 micro controller
Dimensions approximately 1.25” x .75” which makes for a small electronic music box

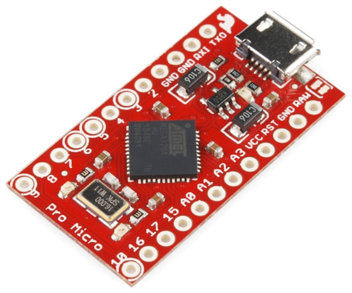


Figure Five – Output Options Schematic

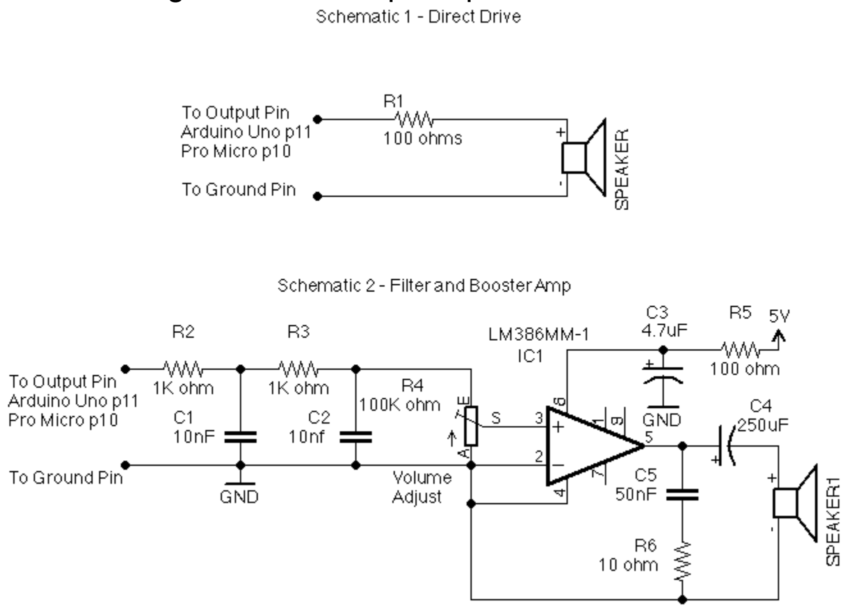


Figure Six – The Attack Waveform

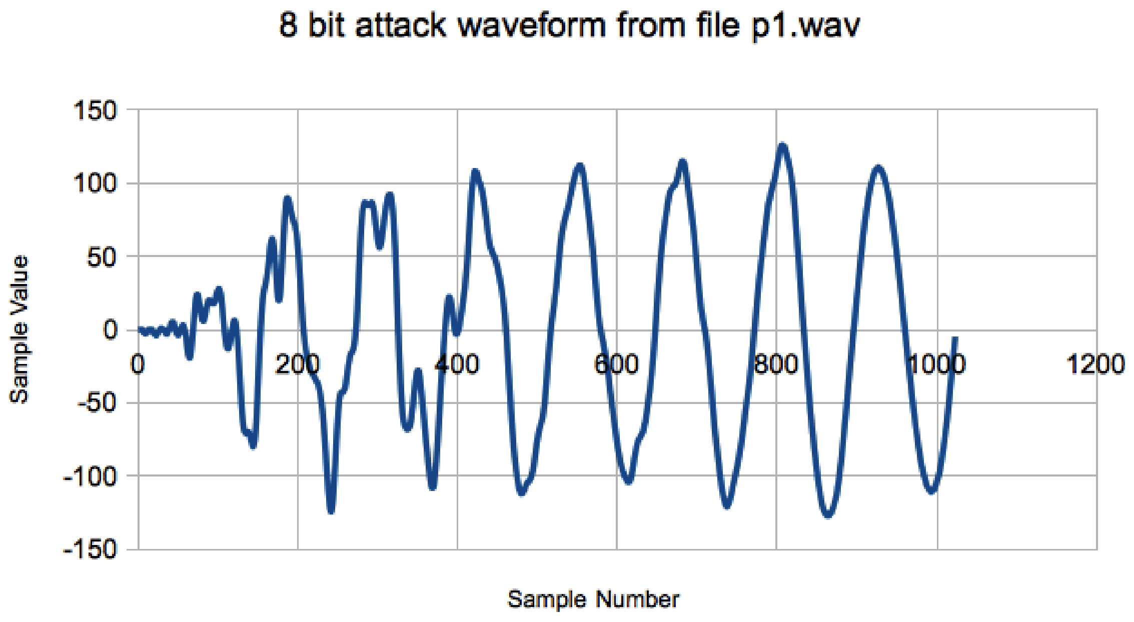


Figure Seven – The Decay Envelope

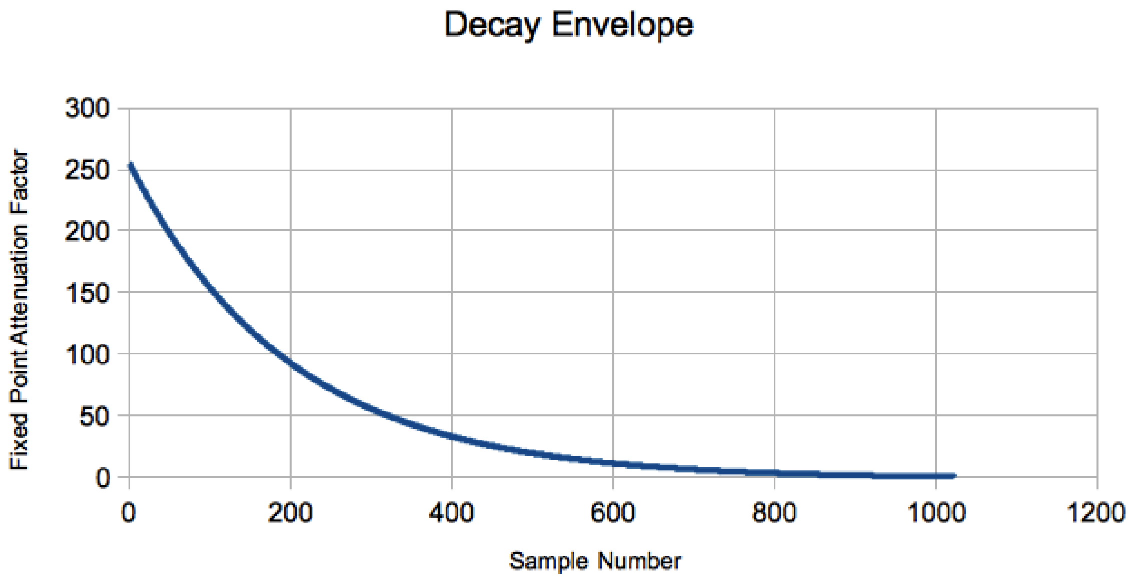
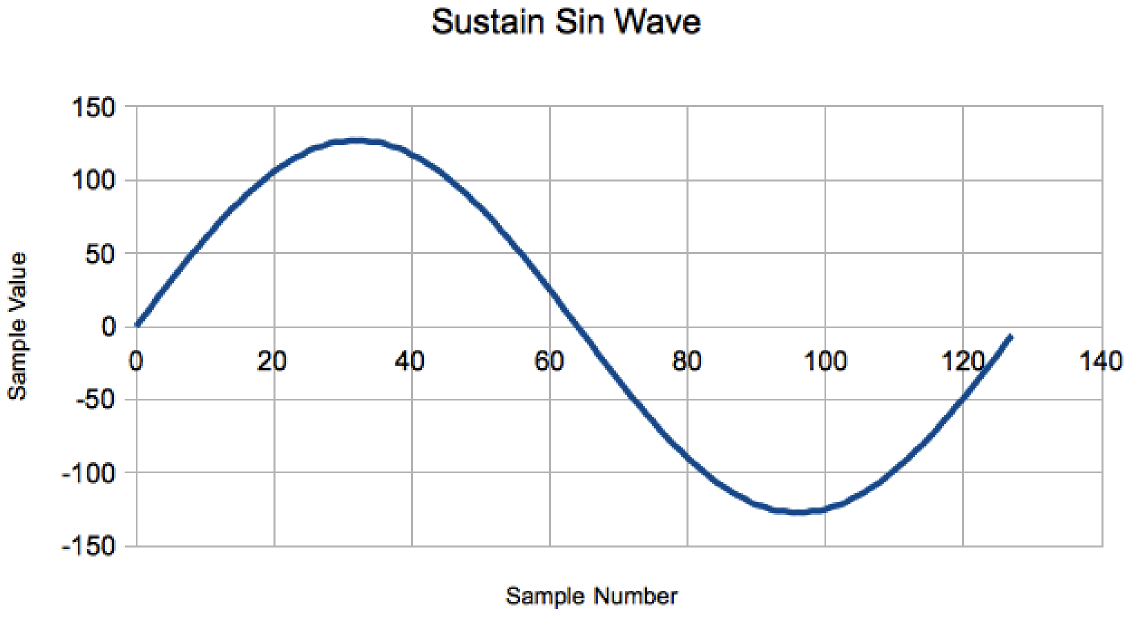


Figure Eight – The Sustain Waveform



Resources

The following links may be of interest to those seeking more information on the topics described in this article.

The official source of Arduino information is: http://arduino.cc.

The free and open source Arduino development tools for Windows, OSX and Linux are available for free at: http://arduino.cc/en/Main/Software.

Information about the SparkFun Pro Micro is available at: http://www.sparkfun.com/products/10998

All of the software described in this article is available from the Nuts and Volts website and associated with this magazine issue. The zip file called ddsmusicbox.zip contains the following:

1. The music box sketch for the Arduino Uno based on the ATMega328 processor (MusicBox.ino)
2. The music box sketch for SparkFun's Pro Micro Arduino compatible based on the ATMega32U4 processor (MusicBoxProMicro.ino).
3. A Java jar file called musicboxsupport.jar which contains the software tools (both in executable and source code forms) necessary to support the music box's operation and to add to or change the tunes in the music box.
4. An mp3 file of the music box playing Fur Elise.

A technical tutorial on digital signal synthesis is available from Analog Devices at: www.analog.com/static/imported-files/tutorials/MT-085.pdf.

The original idea for a DDS music box came from the article entitled Wavetable Melody Generator described at: http://elm-chan.org/works/mxb/report.html.

Author BIO

Craig has been interested in the production of lights/sounds/music with computers for a long time. He is the author of the book “Digital Audio with Java” published by Prentice-Hall. He lives in the mountains of Colorado and can be contacted at calhjh@gmail.com. When not messing around with electronics and/or computer projects, wood working or beer brewing, he does a solo musical act around Colorado Springs.