Digital generation of sine waves

It’s really pretty easy to make an integer output from a microcontroller into an analog sinewave output.

**[FROM WIKIPEDIA:](https://en.wikipedia.org/wiki/Sine_wave)**

**[The sine wave or sinusoid is a mathematical curve that describes a smooth repetitive oscillation. It is named after the function sine, of which it is the graph. It occurs often in pure and applied mathematics, as well as physics, engineering, signal processing and many other fields.](https://en.wikipedia.org/wiki/Sine_wave)**

If you imagine a wave traveling along, the height, or amplitude of the wave can be mathematically determined by assigning an angle to its travel rather than a distance. Let’s say one wave is 10 cm long. If we assume this ***one wavelength is equivalent to 360 degrees:***

|  |  |  |
| --- | --- | --- |
| **Length along the wave, cm** | **Angle equivalent degrees(radians)** | **sine** |
| 10 | 360 (2p) | 0 |
| 7.5 | 270 | -1 |
| 5.0 | 180 (p) | 0 |
| 2.5 | 90 (p/2) | 1 |
| 0 | 0 | 0 |

 2 is just 6.28 (2\*3.14). We take the SINE of the angle equivalent to make a sine curve. We could just as well take the COSINE of the angle to make a cosine wave/curve. Sine waves and Cosine waves look just the same, just offset each other in angle. You can use degrees, but things will be a whole lot simpler computationally if we use radians instead of degrees.

Wavelength is usually notated as . The number of wavelengths produced per second is the signal’s frequency (cycles per second, or Hertz, usually called notated as f). The angular rotation produced per second is the angular frequency (radians/second, or degrees/second) – usually notated as . The amount of time it takes for the signal to rotate once is the signal’s period, usually t, or .

First, you have to decide on the resolution of wave, in terms of how many samples of it you are going to generate per wavelength, and how fine a resolution on the intensity of the output signal

I am going to present an example that splits up the sine wave into 32 chunks during one rotation. That means we will generate one digital integer for every 1/32 of a wavelength

2/32=0.19635 radians

360 degrees/32 = 11.25 degrees

Below is a quick excel sheet that shows the result:



It’s not really a sine wave – just an approximation of one.

Let’s imagine that we are going to use an 8-bit port of our hypothetical microcontroller to simulate this sine wave.

The port uses 8-lines … each representing one bit of an 8-bit byte we put out on the port. There is a Most Significant Bit (MSb) and a Least Significant Bit (LSb).

We want to put out a byte on the port that represents one of the 32 chunks of the sine wave, waits some time, then puts out the next byte representing the next of the 32 chunks, in sequence, until all 32 chunks have appeared – whereupon we start all over again at the very first chunk. The time between the bytes put on the port is 1/32 of the sinewave period we have chosen (we’ll use a timer on the microcontroller to time how long each byte stays up on the port).

Notice the sine values in the table above vary between -1.0 and +1.0, and they are floaters, not integers. If you simply multiply the sine values by the number of bit values in our port, we can fix that. An 8-bit byte can represent any number from 0-255 using its constituent bits.

First, add one to all the numbers in the sine column above. That will take care of the negative numbers and give you a list of 32 numbers going from a minimum of zero to a maximum of 2. Now multiply all those numbers by 127.5 .

Those operations should give you something like this:



Now this has bumped up the value to a maximum of 255, and a minimum of 0. However, we’re using floating point numbers.

We can chop off the right side of the decimal point simply by applying the INT() function, but that would be lightly less accurate than using a round() function to make integers out of our sine wave data.

Once we have integer values from 0 to 255, we can turn them to binary and assign each bit to a pin on the microcontroller port.

To extend this data (rounding it up or down to make integers out of the floats) we extend our calculations to the next level:



The column “rounded value” is the set of decimal numbers we can use to represent a sine wave as a constant stream of integers out of our microcontroller port.

I highlight the value “255” as it is the maximum number. If you go over to its binary representation, all the bits are 1.

When we write an 8-bit integer to a port, each pin comprising the port takes on the value of one bit of the integer. The rounded value of “128” for instance, is comprised of seven “1’s”. The most significant bit (MSB) of the port when representing that integer will be “off” (0). In the “binary value” column the least significant bit (LSB) is the digit all the way to the right, while the MSB is all the way to the left.

 For some reason Microsoft Excel drops the zeros on the left side. 128 should be 01111111, not 1111111. The rounded value of 37 is shown as 100101, but is actually 00100101 when showing the MSB’s.

**NOW, HOW TO IMPLEMENT IT IN HARDWARE:**

We are going to hook up each pin of the microcontroller port to a transistor. We will arrange it so that each pin turns on its own transistor and permits just a known amount of current through the transistor. That current is proportional to the currents through the other transistors turned on by the other pins of the port.

Consider this schematic:



Here we have our port pin connected to the gate of an N-channel MOSFET , Q1 (the resistor, R1 is just there to ensure we don’t exceed the current capabilities of our microcontroller pin). If the pin is zero, nothing happens. When the pin goes to one (which will be something like two, three, or even five volts on that pin), it turns on the MOSFET – causing current to flow through it to the resistor R2 and, hence, through the blue led, D1. If the voltage V1 is high enough and the resistance, R2 is low enough, there will be enough current to make the LED light up. The values shown for the LED (20mA and 3.3 V are typical operating conditions for many blue led’s, not min or max values).

What to make R2?

To make the LED just turn on/off as the port pin goes on/off, it’s just R=V/I. In this case 3.3V/0.02A=165 Ohms. R2=165 Ohms

Actually, there is a little voltage drop across the MOSFET when it turns on, so if V+ = 3.3V, the voltage at R2 will be a few tenths of a volt less – which will drop the resistance absolutely required for R2. However, it is safer to neglect the voltage drop across the MOSFET (at this time). Don’t worry about it, yet.

This circuit just turns the LED on and off. When it’s on, it puts out about 104 millicandella’s, when it’s off, it puts out nothing.

However, if we add another MOSFET and current-limiting resistor, we can make two port pins control the brightness of the same LED over four levels! That’s 2-pins operating at 2-states (22, or 4).

Here is a circuit using 2 MOSFET’s:



Port Pin A controls one MOSFET, and Port Pin B controls another. To operate the LED at 104 mcd, the current through R2, combined with the current flowing through R4, must add up to 20mA at 3.3 V. A simple way to do that is to ensure that both resistors contribute 10mA to the LED, which means R2 and R4 are both about 330 Ohms. We now have four port configurations that can provide 3 brightness levels on the LED:



To take it one step further, let’s divide one of these “current channels” even further to provide even more control.

Let’s take channel “B” which consists of Q2 and R4, and split it into two channels – both contributing 10mA/2 = 5 mA each. To contribute 5 mA each to the LED current, R4 and our new resistor, R6, both have to be 660 Ohms:



And here is the state table:



It’s nice, but redundant: there are multiple port configurations that produce the same current through the LED. We can eliminate those redundant configurations by sacrificing a little LED brightness.

If we made R6 1.32 KOhms instead of 660 Ohms:



Our table of LED currents becomes:



Now we have eight different levels of brightness out of three pins, with no redundant levels, but we lost 2.5 mA of LED current in the process. We can get that 2.5 mA back by duplicating the last channel (add a channel “D” that duplicates the components of channel “C”).

Or, we can add a “D” channel with a current limiting resistance of 2.64 KOhms and buy an additional 1.25 mA (making us only 1.25 mA short of our 20 mA maximum for the LED). We can continue to add channels and more brightness controls by adding more MOSFETS and doubling their current-limiting resistance on subsequent channels.

I’ve controlled just regular old LED’s with 14 bits in just this way (that’s controlling the brightness to one part in 16,384 – assuming brightness is linear with current [it’s not]).

Remember our 8-bit string of values representing a sine wave? Well, if we extend this “one-pin/one-bit” principle further, we can use 8 MOSFET/resistor channels to provide all 256 levels of current control that will permit our LED to duplicate our sine wave using blue light.

Here the extended circuit:



**NEXT: DRIVING THE ILLUMINATOR (LED) WITH THE SINE WAVE**