Physics 364, Fall 2012, Lab #9 Part 2

(DACs and ADCs with the Arduino)

Lab for Wednesday and Friday, November 7th and 9th

Up until now, all of the signals we have dealt with have been strictly binary or strictly analog. The buttons being pressed are either on or off and the emitter diodes have been on or off. In this part of the lab we are now going to use Arduinos to explore two very important topics in their own right – Analog to Digital Converters(ADCs), and Digital to Analog Converters(DACs).

Part 2, Digital To Analog Converters

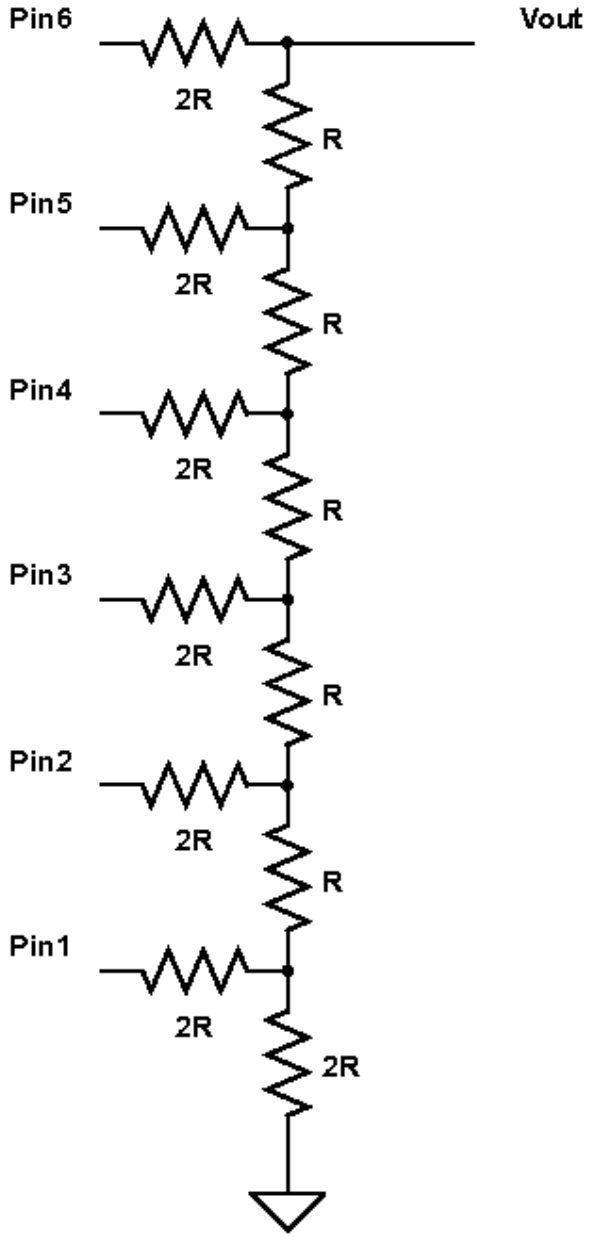
DACs are circuits the Arduino can talk to digitally, and instruct to output a range of voltages. DACs are unable to take arbitrary values, because the Arduino cannot itself produce an arbitrary signal. It is limited by the number of pins available for parallel connections, and the speed of interpretation for serial connections(which would use a single wire). For this lab, we are going to start by making a 6-bit DAC. For a 5 volt full range DAC, we split it into 2^6 segments. Each bit is set to either 0 or 1. So the total voltage output is a function of each bit:

$$V_{out} =(b_6 2^5 + b_5 2^4 + b_4 2^3 + b_3 2^2 + b_2 2^1 + b_1 2^0)\frac{5V}{64}$$

Here we refer to b_1 as the least significant bit(LSB), and b_6 as the most significant bit(MSB). The digitization error on the signal(or the maximum resolution of this DAC) is half the LSB. In this case you can get to within LSB/2 = 5V/128 = 0.0391 V, and the discrete values the DAC can take occur every 0.0781 V. So counting upwards the DAC can take values:

V_{out} = 0, 0.0781, 0.156, ..., $5\frac{63}{64}$V

So how do you go about making one of these circuits? Although you could just sum a bunch of signals from voltage dividers, this would take too many different resistor values. A nice way around this is a 1K/2K resistor ladder:



Hookup digital pins (say 2–7) to the ladder as shown in the diagram, and then have it drive an input to the scope. Write an Arduino program which will perform a signal ramp. Over the course of about 0.1 second, ramp the signal from 0 volts to 5 volts, then drop to zero before repeating. This shape is referred to as a saw wave. You should notice that you have trouble making out the changes of the LSBs, while the MSBs are quite obvious, particularly the largest one.

Measure the output voltage of the DAC while it is driving a resistor, and an LED. What do you think is going on? There is a fairly straight forward fix for this - work it out and implement it to make your DAC behave properly. At this point you will have used an external supply for the first time around your Arduino – keep in mind

that using a voltage above 5 volts or below 0 volts may damage it, so be careful to keep them apart. Remember that you also need a common ground reference, so you should connect the Arduino ground to the supply ground.

Next we are going to make a simple function generator out of the Arduino. In the previous section we made a saw wave. Write code to produce:

Exercise 1: A triangle wave(ie, come back down in a linear ramp as well).

Exercise 2: A square wave, where you store the desired period and amplitude in two variables in your code letting you change them easily.

Exercise 3: A sine wave. The software already has a sin() function($\sin(3.14159) = 0$, etc). Remember that the wave goes from 0 to 5 Volts, so you need to shift and scale the values you instruct the DAC to take. Make sure you can produce the frequency you want and amplitude you want. You may find it necessary to turn a floating point number(ie, one that contains decimals) to an integer. This is most easily done with a "cast" to the new variable type. IE, for float f, you produce int i by:

int i = int(f);

This will drop all of the decimals rather than round(ie, 5.6 becomes 5), so it's not perfect.

Now we are going to examine the frequency space behaviour of the signals we are producing, and what this will mean for ADCs and other circuits which might interact with them. Let's start by looking at your sine wave on the scope. It currently has a bunch of ugly step functions in it. We can get rid of these by introducing a filter. What sort of filter is necessary – will the undesirable features be higher frequency or lower frequency than our sine wave?

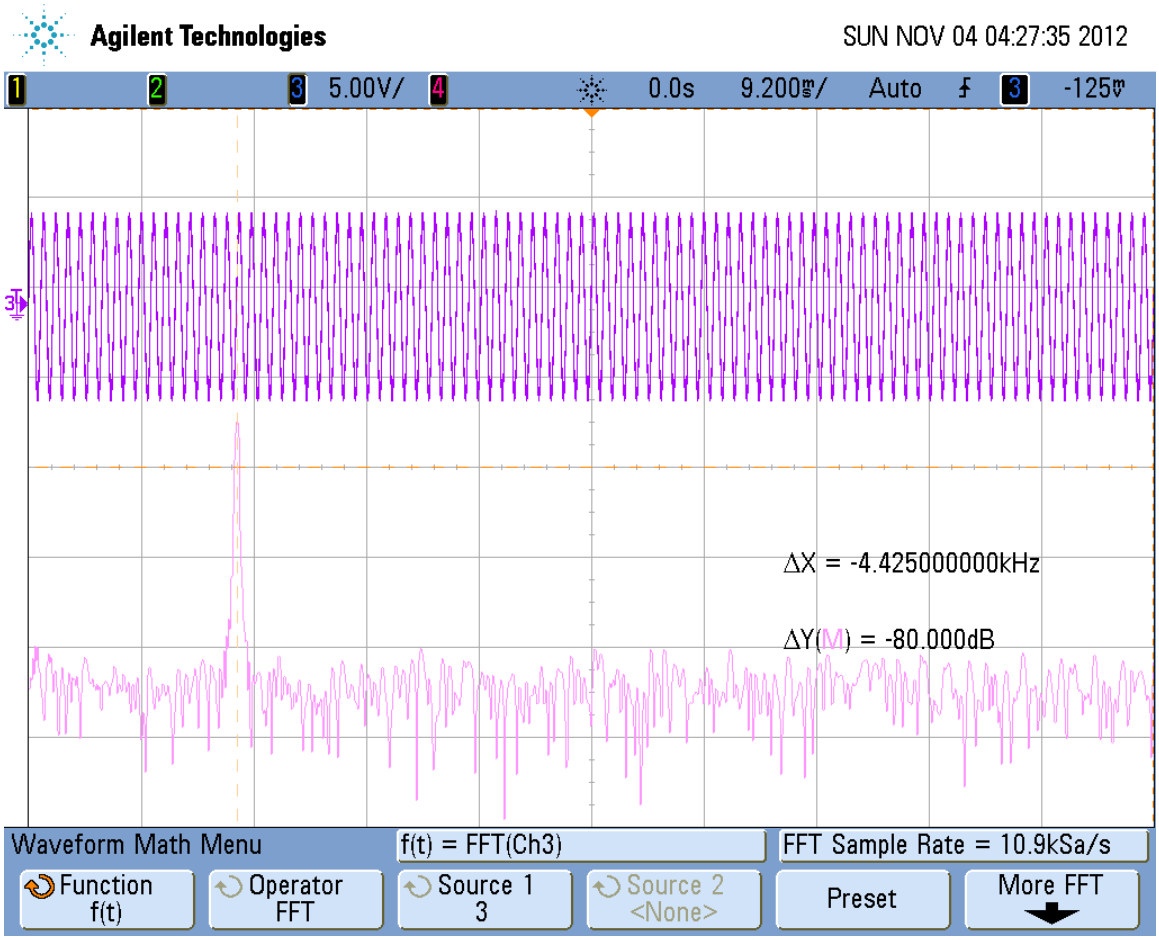
Exercise 4: Add the appropriate filter to remove the step features from your DAC output, and compare the images before and after the filter on the scope to confirm they are gone or reduced.

Exercise 5: Look at the Fast Fourier Transform of the before and after waves(Under the math menu in the scope). For a sine wave you expect most of the frequency content to be in one sharp spike. What is the filter removing?

At this point we want to get a better feel for some faster signals. The Arduino can operate reasonably quickly for something it's price, but it does take it a while to perform the 2^{10} operations needed to produce this waveform each period(careful coding can drastically increase the speed, but we won't ask you to do it. Can you think of how to make this code run fast?). The function generator gives us some

faster signals which won't take so long to draw on the scope, so switch to that now.

Exercise 6: Generate a sine wave with the Agilent 3351A function generator at 1 KHz. What is the Nyquist frequency for this signal? Display the FFT on the scope. Confirm that the peak is in the correct position. The FFT on the Agilint is setup to only display up to the Nyquist frequency for the number of samples it's collecting, so that any true signal is being displayed correctly. However, that doesn't mean that higher frequency signals aren't being aliased ONTO the FFT. Let's start by looking at the easiest case of aliasing. We are going to use the scope as our ADC for this part of the lab. This is all a scope really is doing – taking an analog signal and converting it to a digital one and displaying it on screen for you. Keep in mind that although the scope is a very good ADC, it still has all the limitations inherent in measurement.



The screen above shows the FFT screen with the samples per second shown. To see the number of samples per second, look at the FFT sample rate(you may need to hit the "More FFT" button). This is the number of samples/second(or kila–samples per second). How many samples do we need at minimum to reconstruct our wave correctly? Use the measure menu to find the X@MAX value for the FFT(ie, use MATH as the source) so you can track the frequency peak.

Try increasing the frequency until the Nyquist frequency, and then past it. What do you observe happening to the frequency and amplitude as you move over this threshold?

Alternatively, you can switch the horizontal display to fine mode, and decrease the KSa/second to that number of samples. As you decrease the number of samples to the minimum required number, what happens to the amplitude and primary frequency of the wave? Now decrease the number of sample further, and observe the results.

This is the Aliasing phenomenon discussed in the readings, which is dangerous not only because you lose information, but because you gain WRONG information.

Exercise 7: For a sine wave this is not so bad though – just sample faster than your primary frequency. But what about for other signals? Zoom to a nice viewing position for a square wave at 1 KHz. We know that the Fourier transform of a square wave should be a series of distinct peaks. With ample sampling you should see a series of large peaks. Decrease the sampling rate until you see aliasing in the 3rd peak but not the first two. Is it possible to measure the signal without aliasing? This measurement should lead you to conclude that you should always filter input into an ADC. Why does this help?

Part 3, Analog to Digital Conversion

Using a comparator we can use our DAC to make an ADC - an analog to digital converter. The first way we will do this is a ramp–compare ADC. Setup a 311 comparator IC, and use a DG403 analog switch IC to make a Sample and Hold circuit as in lab7 part 4(`http://positron.hep.upenn.edu/wja/P364_2012/lab07_part4.pdf`), and then use the sample and hold circuit as the input for one leg of the comparator, and the output of your DAC for the other. Now we are somewhat stuck because we need a way to display our result. We are going to use the computer itself to display our result. To open a serial connection over the USB cable to the computer add the following line to your setup code:

```
Serial.begin(9600);
```

This sets up a 9600 baud connection to the computer. You can write your result from the ADC calculation with the line:

```
Serial.print("\t output = ");
Serial.println(outputValue);
```

Recall that your outputValue from the ADC is probably measured in units of the DAC voltage step(often referred to as 'channels'). Before you write it to the serial interface you should convert it back to volts. Hint: variables labelled 'float' instead of 'int' are able to contain non–integer values.

Exercise 8: If you now output a saw tooth on the DAC the voltage at which the comparator switches is the analog voltage value, so you just need to watch for the switch and note the voltage value at which it happens. Remember the order of events that need to occur here: You first need to fix your sample with the sample and hold circuit, and then ramp. You can think of this as taking the value, and progressively checking each DAC value in turn to see if you've got the right one yet.

This strategy is referred to as 'polling' the value, because your Arduino needs to constantly ask the digital pin what value it has while a ramp is occurring. Although we will not discuss the topic further, a second type of behaviour exists called an 'interrupt', where the act of a pin going high will cause the microprocessor to immediately execute a piece of code, regardless of what it is doing at the time. This behaviour is extremely important to many modern systems – for instance pressing the power key on most modern computers doesn't cut power, but rather sends a hardware interrupt asking things to shut down gracefully but immediately.

Another approach to ADCs is successive approximation. Instead of checking every value, perform a binary search. Start by checking if it is greater than 2.5 V, or less. If less, check 1.25 V, if greater 3.75 V, and repeat to whatever bit depth you need. In pseudocode:

```
estimate = 0
for bits i = 0...N(MSB to LSB)
  set bit i of estimate to 1
  if estimate too low, keep the bit
  if estimate too high, set bit back to 0
repeat
```

Exercise 9: Implement a successive approximation ADC - how much faster is this method on average?

A third class of ADCs is available – Flash ADCs. These work by comparing the signal to fixed voltages, and then decoding each comparators output in parallel. In terms of number of pin updates(clock cycles), how much faster is this than the other two methods we have discussed? As a practical matter these tend to be expensive, and are impractical as the number of pins increases, so a hybrid system of successive approximation and flash is used.

Exercise 10: How fast does your successive approximation system sample? You can remove the serial writes and monitor the rate by looking at the hold pin. By changing the sampling rate of your code you can increase this frequency and make a better ADC. How fast can you go? Remember that the delay() command only works with milliseconds, while delayMicroseconds() allows you to access microsecond precision. What is the limiting factor of your ADC? Lookup the clock rate of the Arduino on the internet – this is the rate at which calculations are updated in the Atmega chip. Is this your limiting step? A good scope can cost over 100 000$. Compare the rate you get with your ADC to the maximum you can see on the scope on the shelf(which is in the 10 000$ range).