Microcontroller Lab 3 (Class 12)

In this lab we will improve your signal generator in two important ways. First, you will add an active filter to the output to smooth out the stair-stepping. Second, you will explore a couple of different ways to speed up the output by pre-computing output values. Finally, we'll learn some better ways to handle buttons and switches and develop a reasonable user interface to control the generator.

Reading

- AOE Chapter 3 on Field-Effect Transistors:
  - Beginning through 3.04
  - 3.14 "MOSFET logic and power switches"
- Excerpts from "Jones on Stepping Motors":
  - Introduction
  - Unipolar Motors
  - Low-Level Drivers
  - Mid-Level Control

Reference

- IRF520 Datasheet
- EAD Stepping Motor Datasheet

Signal Generator Part 2

Active Filter

We will continue this week for a bit with the serial DAC signal generator (hopefully you still have it wired up!)

To remove the stair-stepping in your output, you need a filter. This is a very typical filter application, to remove the effect of the discrete-time sampling introduced by the digital-to-analog conversion. Such a filter is called an *reconstruction* filter. You should have already read AOE chapter 5, and also the following:

> http://edf.bu.edu/PY371/Secret/active_filter_note_sept10.pdf
> http://edf.bu.edu/PY371/Secret/lab_op4_july12.pdf (9L1 only)

(Look on the whiteboard or ask someone if you need the username and password.)

☐ **Lab 3.1 - Active Filter**

Please do the 9L1 lab linked above. Build the circuit in a breadboard area where you can leave it assembled.

☐ **Lab 3.2 - Filtering the DAC output**

Configure your 9L1 active filter as a *Butterworth Filter* and connect the input to the output of your DAC. Connect two oscilloscope channels to the filter input and output. Try generating sine waves of various frequencies and compare the filter input and output. Sketch a couple of examples in your lab book. Does the filter improve the performance of your signal generator?

Speeding Things Up

EXTRA CREDIT SECTION!
If you are running short of time, please skip to the stepping motor project now.

☐ Load up your sine wave generator.

What is the maximum frequency you can generate (without changing `Timer.initialize( 250)`?

Try changing the 250 to a smaller value. What happens? What is the maximum frequency you can achieve?

Let's try to understand what is happening. A common debugging technique for microcontrollers is to use a spare output pin connected to an oscilloscope to trace code execution and measure times.

☐ **Lab 3.3**

Pick an unused digital pin. Connect your oscilloscope to it.
Set it as an output and set it to HIGH at the beginning of `timerFunc` and to LOW at the end of `timerFunc`. Run your sketch.

How long does `timerFunc` take to execute? Move the digitalWrite() calls around to find out what part of the function takes so long. What line in the code is the slowest, and how long does it take to execute?

Look-up Table

A common technique to speed up waveform generation is to pre-compute the values to be output. For example, to generate a sine wave with up to points per cycle we could use code like this:

```
#define MAX_SINE 512
word sine_table[MAX_SINE];
#define pi2 (2*3.14159265)

// compute sine table for npt points per cycle
// scale to 0..4095
void compute_sine( int npt) {
  if( npt > MAX_SINE)
    return;
  float a = 0;
  float da = pi2/npt;

  for( int i=0; i<npt; i++) {
    sine_table[i] = 2048.0 + 2047.0 * sin(a);
    a += da;
  }
}
```

This function will calculate the values of sin() for the specified number of points and store them in an array. Then, your interrupt routine (`timerFunc`) could just retrieve the next value from the waveform and output it, which would be much faster than calling sin() each time. The only issue here is that you have to have enough memory for the table. Remember that your Arduino only has 2048 bytes of RAM, and each value in a “word” variable takes two bytes.

You can save another factor of four by observing that all four quadrants of the sine function have the same shape, so you can just store values from 0..pi/2.

Storing Data in Program Memory (Flash)

Finally, there is yet another trick to avoid using a lot of RAM. You can pre-compute the sine values and store them in the *flash* memory, where the program is stored. You can read about it here.

Here is a pre-computed table of sine values for the domain 0..pi/2 in 128 steps, scaled from 0 to 32767.

```
// table of sines from 0-pi/2 in 128 steps
prog_int16_t sine[] PROGMEM = {
  0, 402, 804, 1206, 1607, 2009, 2410, 2811, 3211, 3611,
  4011, 4409, 4807, 5205, 5601, 5997, 6392, 6786, 7179, 7571,
  7961, 8351, 8739, 9126, 9511, 9895, 10278, 10659, 11038, 11416,
  11792, 12166, 12539, 12909, 13278, 13645, 14009, 14372, 14732, 15090,
  15446, 15799, 16150, 16499, 16845, 17189, 17530, 17868, 18204, 18537,
  18867, 19194, 19519, 19840, 20159, 20474, 20787, 21096, 21402, 21705,
  22004, 22301, 22594, 22883, 23169, 23452, 23731, 24006, 24278, 24546,
  24811, 25072, 25329, 25582, 25831, 26077, 26318, 26556, 26789, 27019,
  27244, 27466, 27683, 27896, 28105, 28309, 28510, 28706, 28897, 29085,
  29268, 29446, 29621, 29790, 29955, 30116, 30272, 30424, 30571, 30713,
  30851, 30984, 31113, 31236, 31356, 31470, 31580, 31684, 31785, 31880,
  31970, 32056, 32137, 32213, 32284, 32350, 32412, 32468, 32520, 32567,
  32609, 32646, 32678, 32705, 32727, 32744, 32757, 32764, 32767
};
```

We can use this to make a function which calculates the sine of an angle, with domain 0-511 and range -32767 to 32767:

```
// calculate sine in domain 0..511 returning range +/-32767
int lookup_sine( int a) {
  int rv;
  if( a <= 128)          // if in quadrant I
      return pgm_read_word_near( sine+a);
  else if( a <= 256)    // quadrant II : mirror about Y axis
      return pgm_read_word_near( sine+(256-a));
  // quadrants III or IV, mirror about X axis
  else return( -lookup_sine( a-256));
}
```

There are several interesting things going on here.

First, to store data in flash, you have to use both a special type name (**prog_int16_t**) for the variable plus the qualifier **PROGMEM** to tell the compiler to put the array in the flash. The **sizeof()** function returns the size of something *in bytes*. So if you want the compiler to calculate how many elements there are in an array, use the idiom **sizeof(array)/sizeof(array[0]).** Finally, to access a variable in flash you must use a special function call, which must be matched to the data type you wish to read, and just to be confusing the function names don't match the names of the data types. Read the documentation here or here to learn more. The reason you have to use (**squares+i)** instead of the more obvious **squares[i]** is a bit subtle. The **pgm_read...** functions take a *pointer* as an argument, which is essentially the address in memory where the item to be read is stored. Google around if you want to learn more about pointers... here is one useful tutorial.

- [ ] **Lab 3.4**

  Modify your sine wave generator from Lab 2 to use one of the techniques above. Instrument the timer interrupt with a test pin as Lab 3.3 and see how much faster it runs.

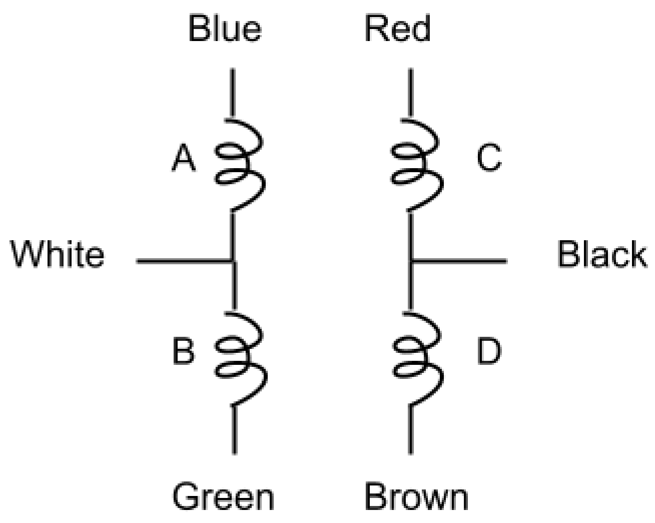
Power Electronics - Stepping Motor

CAUTION! The resistors get hot enough to burn you when the motor has run for a while. This is ok... that's why we use 10W resistors.

There are several types of motors commonly used in robotics and other applications where precise control is needed. One of the most common is the *stepping motor*, which can provide exact, repeatable positioning. Today you are going to build an interface circuit to drive a stepping motor from your Arduino. You need the following parts, which should be available in the lab:

- (1) Eastern Air Devices stepping motor
- (4) IRF520 N-Channel power MOSFET transistors
- (4) 1N4001 diodes
- (2) 20 ohm 10W power resistors
- (1) power supply capable of delivering 12V at ~ 1A

Eastern Air Devices motor windings diagrammed below. Possibly yours will have different colors. Each winding should be about 6 ohms; verify the wiring with your ohm-meter.



The basic challenge in driving a stepping motor is to energize the motor windings in the correct sequence, which causes the motor to rotate in discrete steps. The motor we are using is a precision one, with 1.8 degree full steps and 0.9 degree half steps. From Jones we read that you energize the windings in the following typical sequence for full-step operation:

ABCD
1010
1001
0101
0110

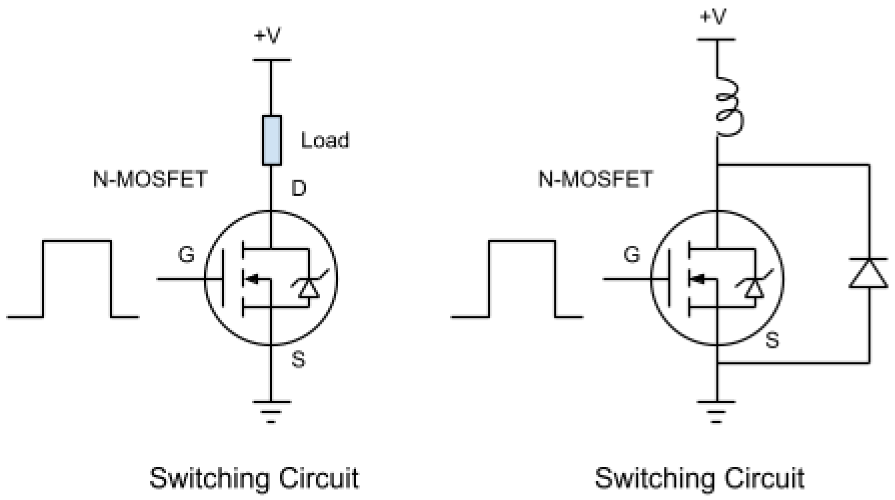
A '1' represents an energized winding, with the bit positions representing the four windings A, B, C, D as shown in the diagram above. For "full step" operation with two windings energized at a time, the motor rotates 1.8 degrees per step. The following trajectory will half-step the motor, resulting in 0.9 degrees per step but slightly reduced torque:

1010
1000
1001
0001

0101
0100
0110
0010

The motor windings require much more current (about 1 Amp) than the Arduino can provide directly from it's digital outputs, so we use a power MOSFET as a switch. The principle is the same as for the small MOSFETs you used to make logic gates in an earlier lab. We are using an N-Channel enhancement mode MOSFET, which behaves like an NPN bipolar transistor, with the source connected like the emitter, and the gate connected like the base. An enhancement mode MOSFET is "normally off" and requires a positive V_{GS} (gate-source voltage) to turn it on. For the IRF520 MOSFET, a V_{GS} of at least 5V is required for turn on. This voltage can be provided directly by the Arduino digital outputs.

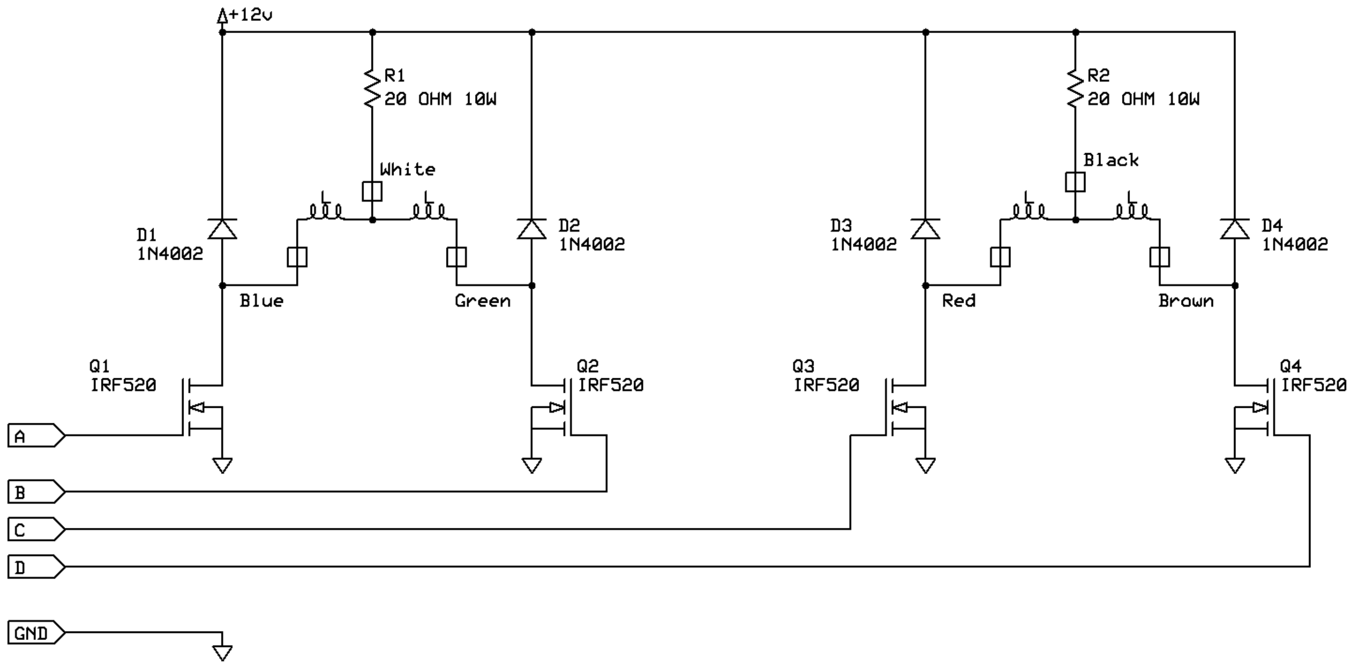
Typical MOSFET power switching circuits are shown in the figure below. For a resistive load (i.e. a high current lamp) the circuit is very simple; just connect the load between the drain terminal and an appropriate positive supply. For an inductive load such as a motor winding, a diode is required to dissipate the energy stored in the inductor, which would otherwise cause a potentially large voltage spike when the switch is opened which could damage the MOSFET. The IRF520 has a hefty diode built-in so we don't need a diode there.



Switching Circuit
Resistive Load

Switching Circuit
Inductive Load

Below is the full schematic for the drive circuit for the stepping motor. The boxes with color names represent the connections to the motor, while the flags with signal names A/B/C/D represent connections to Arduino digital outputs.



Notice that there are now diodes in a new position in the circuit. The reason for this is slightly subtle. Each pair of windings acts like an auto-transformer. Briefly, this means that the magnetic field produced when one half of the double winding is energized produces a potential in the other half which could exceed the voltage rating on the MOSFET. The diode prevents the other end of the winding from going above the 12V supply.

▢ **Lab 3.5**

Construct the circuit shown above on your breadboard. Since relatively large currents and voltages are involved which could damage the Arduino, be extra careful to wire everything correctly. Check the pinout of the MOSFET carefully with the datasheet. Wire the signals A/B/C/D to digital outputs on your Arduino.

Write a sketch which will turn the A/B/C/D outputs on in sequence as shown in the table above. Add a delay of 10 ms after you update all four outputs.

What happens to the motor?

How many steps does it take to complete one rotation?

▢ **Lab 3.6**

Write a function in your sketch with the prototype:

```
void run( int steps, int speed)
```

where **steps** is the number of steps to turn (negative numbers for reverse direction) and

speed is the speed, expressed in steps per second (i.e. 200 is one revolution per second). What is the fastest speed the motor will run at reliably? How does this compare with what is in the datasheet?

- ▢ **Lab 3.7**

  Modify the sketch to accelerate the motor gradually (over, say, 25 steps) instead of starting at full speed. Can you run reliably at higher speed now?

- ▢ **Lab 3.8**

  Modify the sketch to support half-step mode on the motor, and repeat the speed tests. How do the results differ?