Physics 364, Fall 2014, Lab #21 **Name:** _____

(Arduino 2: digital ↔ analog conversion)

Wednesday, November 12 (section 401); Thursday, November 13 (section 402)

Course materials and schedule are at positron.hep.upenn.edu/p364

Physics 364 Arduino Lab 2 Vithayathil/Kroll

Adapted from a lab originally written by Simon Hastings and Bill Ashmanskas

Last revised: 2014-11-12

Introduction

Up until now, all of the signals we have dealt with have been strictly analog voltages or strictly two-state voltages (digital or binary). In this part of the lab we are now going to use Arduinos to explore Digital-to-Analog Converters (DACs) and Analog-to-Digital Converters (ADCs).

DACs are circuits that convert a binary number expressed using two-state digital voltages to an analog voltage. The analog voltage resolution is determined by the number of bits used to divide the digital voltage level. ADCs convert an analog voltage to a binary number represented by two-state digital voltages.

In this lab, you will construct a 6-bit DAC. Each bit is set to either 0 or 1 (LOW=0 V or HIGH=5 V). The voltage output is a function of each bit:

$$V_{OUT} = (bit_6 \cdot 2^5 + bit_5 \cdot 2^4 + bit_4 \cdot 2^3 + bit_3 \cdot 2^2 + bit_2 \cdot 2^1 + bit_1 \cdot 2^0) \cdot \frac{5\,V}{64}$$

Here 5 V is the output voltage of the HIGH output of the Arduino board, bit1 is the least significant bit (LSB) and bit6 is the most significant bit(MSB). The analog voltage V_{OUT} is quantized: what is the smallest difference between analog voltage levels when the maximum voltage is 5 V and there are six bits of precision?

Once you build a DAC, you can use it to compare the output voltage of the ramp created by the DAC with an analog signal. The binary bits of the DAC voltage that match the analog voltage is the digital representation of the analog voltage.

Exercise 1: DAC

A nice way to build a DAC is with a R-2R resistor ladder. A schematic of such a ladder is shown in Fig. 1. Before proceeding, you should understand how this resistor ladder works. Think about it from the point of view of superposition: for example, what value would the output voltage have if the input voltage is 5 V at pin 6 and the remaining pins are at ground? Then consider this same case, but with pin 6 replaced with one of the other pins.

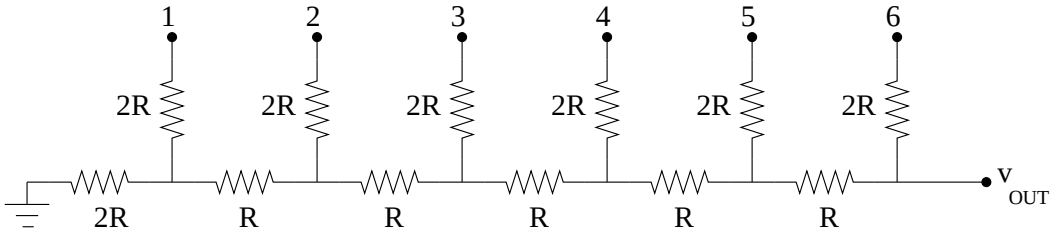


Figure 1: A six-input R-2R resistor ladder for digital-to-analog conversion.

Assemble this ladder on your breadboard; use a resistor value R = 1 k. For ground, use the Arduino board ground. Connect the Arduino digital pins (say 2-7) to the ladder, and connect the output of the R-2R ladder directly into the oscilloscope. Write an Arduino sketch that will perform a signal ramp, that is, you want to have the digital pins cycle from 0 to 63 in binary so that the output voltage cycles from the lowest value to the highest value. (This sketch should be very similar to the sketch that you wrote in the first Arduino to make the six LED lights count from 0 to 63.) You should use a delay of at least 1 ms for the steps in your ramp. Observe the output on the oscilloscope. This shape is referred to as a saw wave. Discrete steps should be noticeable on the DAC output. In addition you should observe narrow spikes, which are $\sim 20\,\mu$s wide, whenever significant bits change state. These spikes arise because the ramp is generated by changing the six bits in sequence and not simultaneously.[1] Summarize your observations on the next page.

[1] If you want to try updating all of the bits simultaneously, to eliminate the spikes, look up `PORTD` at http://www.arduino.cc/en/Reference/PortManipulation .

Now measure the output voltage of the DAC while it is driving a load consisting of a 1 k resistor (that is, connect a 1 k resistor from the output to ground and look at the voltage across this 1 k resistor). You should observe that the output impedance of the DAC is comparable to your 1 k load.

To reduce the output impedance of the R-2R ladder, you may use a voltage buffer (a.k.a. "follower") constructed with a 741 opamp. The pinout for the 741 is shown in Fig. 2; you probably remember from the analog labs how to wire up an opamp as a follower, but if you don't, feel free to ask us to remind you. Use ± 15 V to power the opamp and **make sure that the power-supply ground and the Arduino-board ground are connected** on the breadboard.

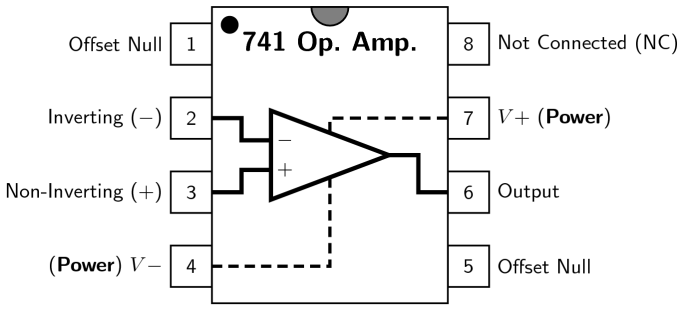


Figure 2: 741 opamp pinout. Use an opamp "follower" (a.k.a. voltage buffer) circuit to buffer the output of the R-$2R$ ladder shown in Fig. 1.

Check that the voltage buffer is doing its job by placing the 1 k load at the output of the voltage buffer and then looking at the ramp signal across this 1 k output load.

You should explain why this buffer makes the output impedance small. You should also explain why the output voltage of the R-2R resistor ladder is determined by the binary number fed to the DAC inputs.

The narrow spikes of the output of the DAC should be filtered with a low-pass RC filter placed on the output of your opamp buffer. You can try a time constant of $100\,\mu$s for your low-pass filter.

Please draw your updated schematic diagram, including resistor ladder, opamp voltage buffer ("follower"), and lowpass filter, including relevant component values.

Exercise 2: A Function Generator

Now modify your Arduino sketch to produce a triangle wave (the voltage ramps up linearly with time and then ramps down linearly with time). This triangle wave will be offset from zero, of course. **Please summarize below how you accomplished this task.**

Optional: If you have time, try to produce a sine wave. The sin() function is available in the Arduino (consult the Arduino reference web page). Remember that the wave goes from 0 volts to 5 volts, so you need to shift and scale the values you instruct the DAC to make. Make sure you can produce the frequency and amplitude you want (as with the triangle wave, this sine wave will be offset from zero as well). You may find it necessary to turn a floating point number (*i.e.*, a number that contains decimals) to an integer. This conversion is most easily done with a "cast" to the new variable type, *i.e.*, for a floating variable f, you produce an integer value i by: int i = int(f); this will drop all digits after the decimal point, it e.g., 5.6 becomes 5. **If you do this, please either show it off in person to one of us or describe briefly in the space below what you did.**

Exercise 3: Analog to Digital converter

To convert an input analog signal to a digital voltage expressed as binary data, the voltage of the analog signal should be compared with a reference signal set by the DAC. To do this we will use a comparator chip: the LM311, which we used in Lab 11. The LM311 data sheet is at https://www.fairchildsemi.com/datasheets/LM/LM311.pdf . The configuration we will use (Fig. 3) closely resembles the configuration we used in Lab 11.

A comparator compares two input signals, S_+ and S_-, and outputs one of two states, a HIGH or a LOW, depending on whether $S_+ - S_-$ is greater than zero or less than zero. It is possible to accomplish the same result with an opamp used without feedback, however, a comparator is designed to have a faster response. (Basically if the difference between the inputs is anything but zero, the output will saturate.)

We now know enough to understand the comparator's **open-collector output**, which appeared somewhat mysterious before we had studied transistor circuits. If $S_+ < S_-$, the comparator turns ON (saturates) the output transistor, which connects V_{out} (pin 7) to ground (pin 1) with low resistance. If $S_+ > S_-$, the comparator turns OFF (cuts off) the transistor, which puts pin 7 into a disconnected (so-called "high-impedance") state. When the transistor is ON, pin 7 is driven down to equal whatever voltage is supplied by pin 1; when the transistor is OFF, pin 7 is pulled up through the external 1 kΩ resistor to +5 V. This "open-collector" output is convenient, because it allows us to make the two output states be +5 V and 0 V (for example) instead of ±15 V as they were in Lab 11.[2]

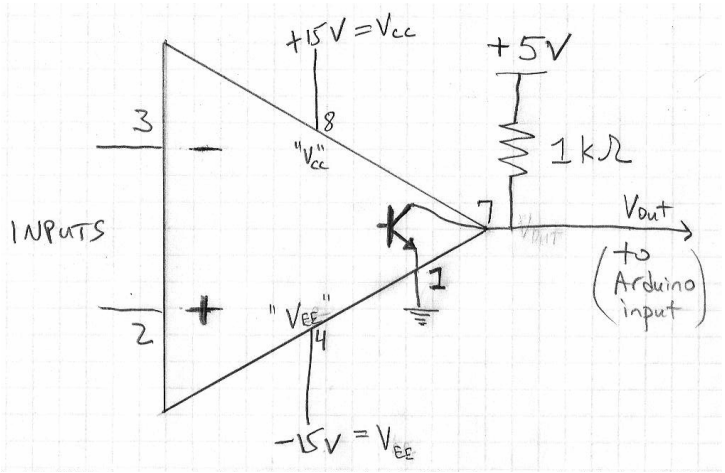


Figure 3: Schematic for using a '311 comparator as part of an ADC. The inputs are the analog signal and the ramping voltage from the DAC you built earlier in this lab.

[2]Another common use of an open-collector output, which we do not exploit here, is the ability to make a "wired OR." If you wire together the open-collector outputs of several comparators, with a single pull-up resistor to +5 V, the output will be HIGH only if ALL of the comparators are in the HIGH state; if one or more comparators are in the LOW state, the shared output is driven LOW. This configuration (which is confusingly called a "wired OR") implements the logical AND function.

Build the comparator circuit as shown in Fig. 3. Check that you have the comparator circuit assembled correctly by feeding two signals into the input and seeing under what conditions the output is HIGH and LOW. Make sure that the HIGH output is between 4 and 5 volts and the LOW output is $\sim 0\,\mathrm{V}$. (Remember, the input signals on the Arduino should be between 0 and 5 volts.)

Now connect one of the Arduino digital inputs to the output of the comparator. For an analog input signal use a 2 k resistor and a 1 k potentiometer between the power supply $+15\,\mathrm{V}$ and ground. Make sure that the analog input signal to the comparator is between 0 and 5 volts.

Write a sketch to ramp up the DAC and then read the output of the comparator at each ramp setting to determine your analog input voltage (you will have to scale the bit value to a voltage). Display the voltage on the laptop. Check if the voltage displayed corresponds to the analog input voltage.

"Report"

Please include on this page, and the next blank page if needed, both a diagram of the circuit you have built and a very brief synopsis of how your Arduino program ("sketch") works.

Exercise 4: ADC with Sample and Hold (Optional)

An analog signal only conveys interesting information if it varies with time. In order to digitize an analog signal, we need a way to sample repetitively the analog signal and hold the voltage while the ADC determines its digital value. This is achieved with a sample and hold (S/H) circuit. You will use an analog FET switch (DG403) and a FET opamp (TL084) buffer for your sample and hold circuit. The schematic is shown in Fig. 4.

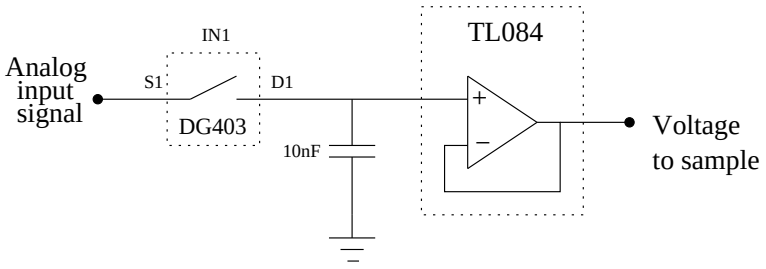


Figure 4: Sample and hold for ADC circuit.

FETs are field effect transistors. They can be used in a similar way as the BJTs you have already encountered. Current between the source and drain of the FET (equivalent to the collector and emitter of a bipolar transistor) is varied by a voltage applied to the gate of the FET (equivalent to the base of a transistor). Because the source to drain current is varied by a voltage, and not by a bias current, the gate current of a FET is very small. The input bias current for the FET opamp is $\sim 50\,\mathrm{pA}$—about three orders of magnitude smaller than the input bias current of 741 opamp, which has BJTs on the input stage. Since these bias currents are small, the voltage applied to a sample and hold capacitor (see Fig. 4) decays quite slowly. The analog FET switch works similar to a push button switch; when the gate signal IN_1 is HIGH, the switch is closed and transmits the analog signal. When the gate signal is LOW, the switch is open.

Build the sample and hold circuit shown in Fig. 4 and connect it to your comparator input. The pin assignments are specified in Table 1 and Table 2. The data sheets for the DG403 and TL084 are provided on the course website.[3]

Connect the DC analog signal from the potentiometer to the S/H circuit. Sample the signal by holding IN_1 HIGH and then HOLD it by holding IN_1 LOW. Measure the output of the S/H and establish that the droop in voltage with time is small. In addition to the FET input current, there are leakage currents in the polystyrene capacitor, and between the pins on the S/H and the ground, that add to the droop rate.

Use a ramp signal from the function generator as the input to the S/H. One of the digital

[3] http://www.intersil.com/content/dam/Intersil/documents/dg40/dg401-03.pdf
http://www.ti.com/lit/ds/symlink/tl084.pdf

Pin	Assignment
1	input analog signal (S1)
16	switch output (D1)
15	gate (IN_1) from Arduino
10	gate (IN_2) connect to ground
11	+15 V (V+)
12	+5 V (V_L) from Arduino
13	GND
14	-15 V (V-)

Table 1: Pin assignments for the analog switch DG403. As always, pin numbers start from the lower-left corner and increase in the counterclockwise direction.

Pin	Assignment
3	non-inverting input
2	inverting input
1	output
4	+15 V
11	-15 V

Table 2: Pin assignments for the FET opamp (TL084). As always, pin numbers start from the lower-left corner and increase in the counterclockwise direction.

pins of the Arduino has to be connected to the IN_1 of the analog switch. Setting IN_1 to HIGH samples the input. Write a sketch to repetitively sample and hold (with a uniform delay) the function generator ramp signal and display the value on the laptop. Do successive values displayed differ by a constant as expected for a ramp? You may cut and paste the output values into a spread sheet program such as Excel and plot the signal versus time (or use Matlab) — or not, since all of this is optional.

Don't take apart your circuit yet, as Exercise 5 builds on this circuit.

Exercise 5: Successive Approximation (optional)

The ADC using the S/H in Exercise 4 needs on the order of 2^6 steps to linearly ramp up the DAC and then locate a match with the analog input signal. Instead of this linear ramp, we can use a binary search: *i.e.*, set the high bit of the DAC and compare with the analog signal; if the analog signal is larger, set the next higher bit too, etc. Modify your sketch to implement a successive approximation ADC.

With an n-bit ADC, a binary search takes only n steps to converge on the binary representation of the analog voltage, in comparison with the 2^n steps needed for a linear search. As n increases, a binary search presents an immense time advantage compared to a linear search. ADCs using a binary search are called successive approximation ADCs. They are widely available commercially. Flash ADCs (FADCs) have an even faster conversion time. FADCs use several comparators in parallel to obtain all the bit values in one step. The A/D converter in the oscilloscope uses an FADC.

Optional 6: reaction timer (same as Lab20 optional6)

Using one push-button input (with connection to +5 V and pull-down resistor to ground) and one LED (with current-limiting series resistor), make a **reaction timer** that measures how quickly your finger responds to a visual stimulus from the Arduino.

The Arduino should first turn on the LED, then wait for you to press a button. Once you press the button, the Arduino should turn off the LED, then delay a length of time $1000 \text{ ms} < \Delta t < 5000 \text{ ms}$, then turn on the LED. The Arduino then measures how long it takes before you respond by pressing the button. As soon as you press the button, the Arduino turns the LED off, then reports (via `Serial.print()` to the serial console) the number of milliseconds that it took you to react. After another brief delay, the Arduino turns the LED back on and returns to the initial state.

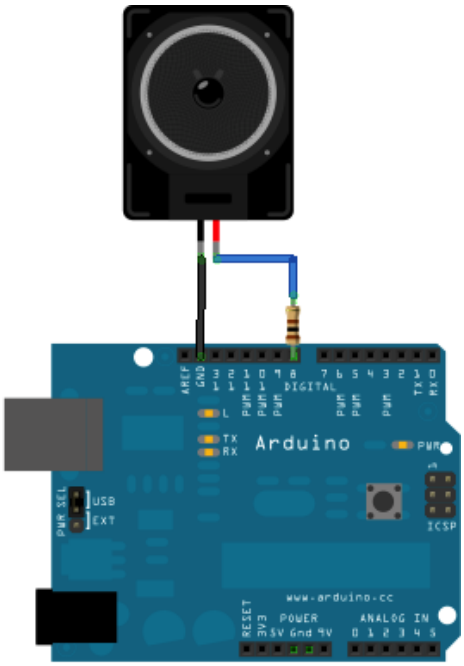
If you do this exercise, you don't need to write anything down, but please call us over so that you can show off your results!

Optional 7: musical tones (same as Lab20 optional7)

First make a sketch whose `loop()` function first outputs `HIGH` on pin 8, then waits 1.136 ms by calling `delayMicroseconds(1136)`, then outputs `LOW` on pin 8, then once again calls `delayMicroseconds(1136)`. Check with your oscilloscope that the result is a 440 Hz square wave on pin 8.

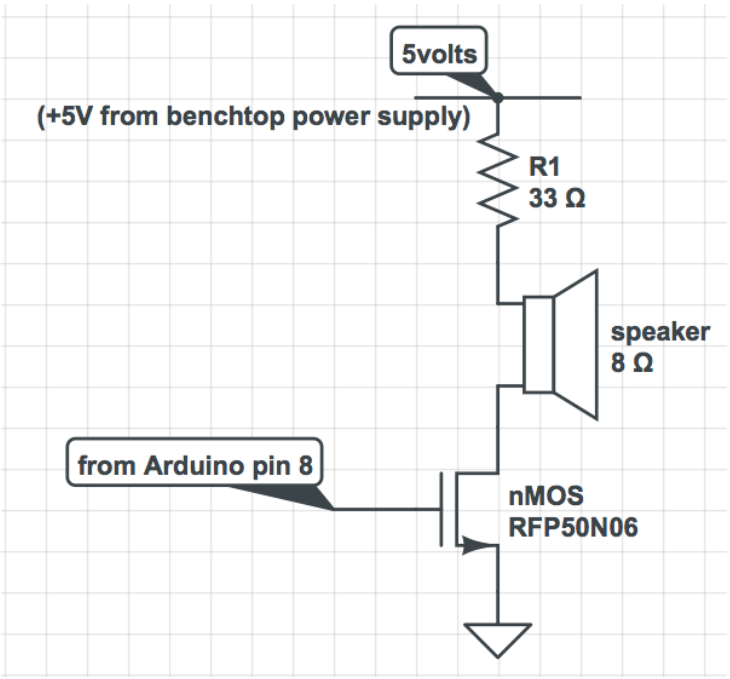
Now notice that the same result can be achieved much more easily by calling the built-in function `tone(pin, frequency)` where in this case `pin` = 8 and `frequency` = 440. You can also call `tone(pin, frequency, duration)` where `duration` is in milliseconds.

Next, connect a 100 Ω resistor and a speaker in series from pin 8 to ground, as shown below. You should hear (quietly) the **A** note above middle C.



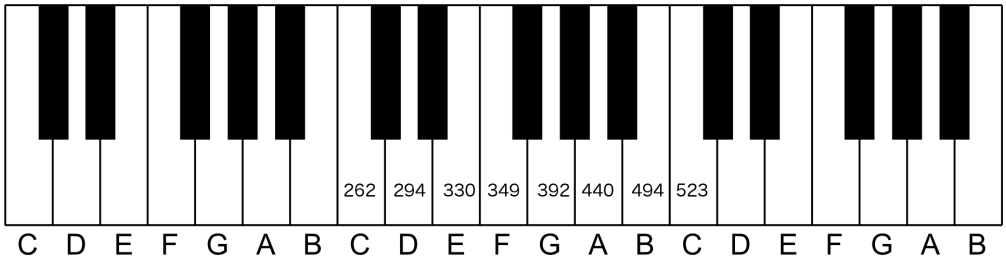
(Continued on next page.)

If you really want to make enough noise to annoy your neighbors, you can try using a MOSFET switch to boost the power supplied to the speaker! Here is a circuit that I (Bill) tried out in the lab the other day:

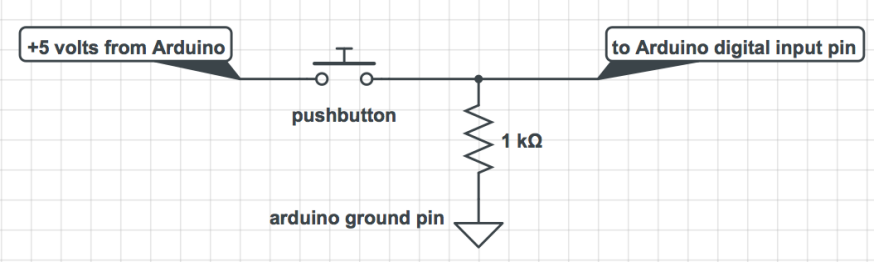


Whether you go with the quiet or the noisy version, here are two more challenges you can add to this exercise:

First, try playing a major scale, starting from middle C (262 Hz). The notes C, D, E, F, G, A, B, C that you need to play have frequencies 262, 294, 330, 349, 392, 440, 494, 523 Hz, as indicated on the keyboard below. Notice that the frequency ratio between octaves is 2, between adjacent keys ("half step," e.g. E to F) is $2^{1/12} \approx 1.06$, and between two keys having one key in between ("whole step," e.g. C to D) is $2^{1/6} \approx 1.12$.

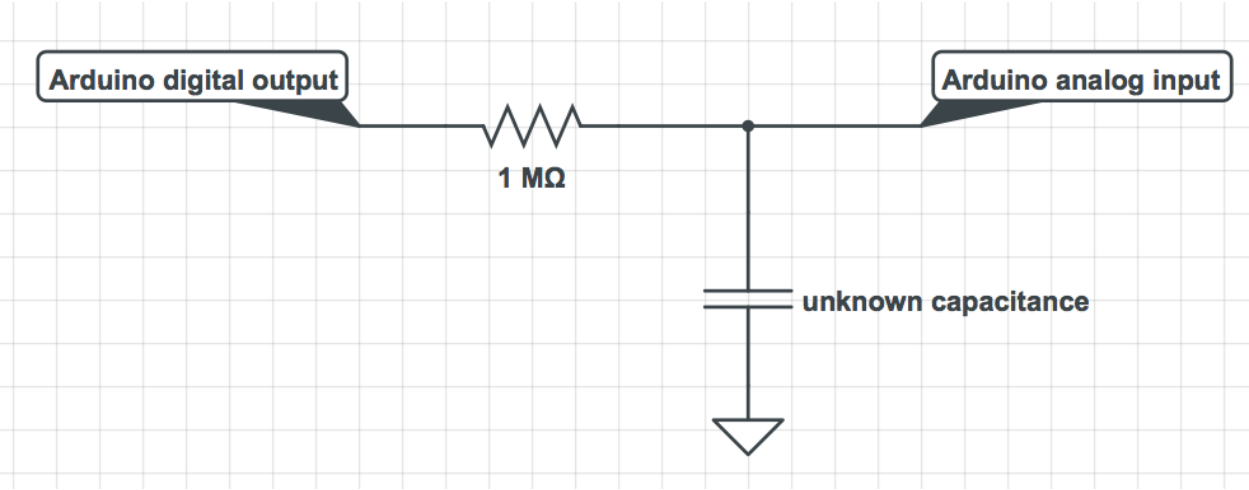


Second, if you're feeling ambitious, try making a simple 8-button musical keyboard, using 8 copies of this circuit to drive 8 Arduino digital input pins:



Optional 8: capacitance meter (new!)

Here is a chance to apply the Arduino's own built-in analog-to-digital converter, in order to build something potentially useful — a *capacitance meter*. The idea is this: first, the Arduino drives a digital output to 0 V, to discharge the capacitor. Then it monitors an analog input until the voltage reads $\approx$ 0 V, so that it knows the capacitor is fully discharged. Then it records the present time, using e.g. `int time0 = millis();` , drives the digital output to +5 V, and monitors the analog input until V_{in} exceeds (5 V)$(1 - e^{-1}) \approx 3.1606$ V. Then it records the stop time, using e.g. `int time1 = millis();` . The unknown capacitance, determined from the known resistance R and known decay time $\Delta t = RC$, can then be displayed on the serial monitor, in nanofarads. Once the meter is verified using a known capacitor, try it using several "unknown" capacitors.

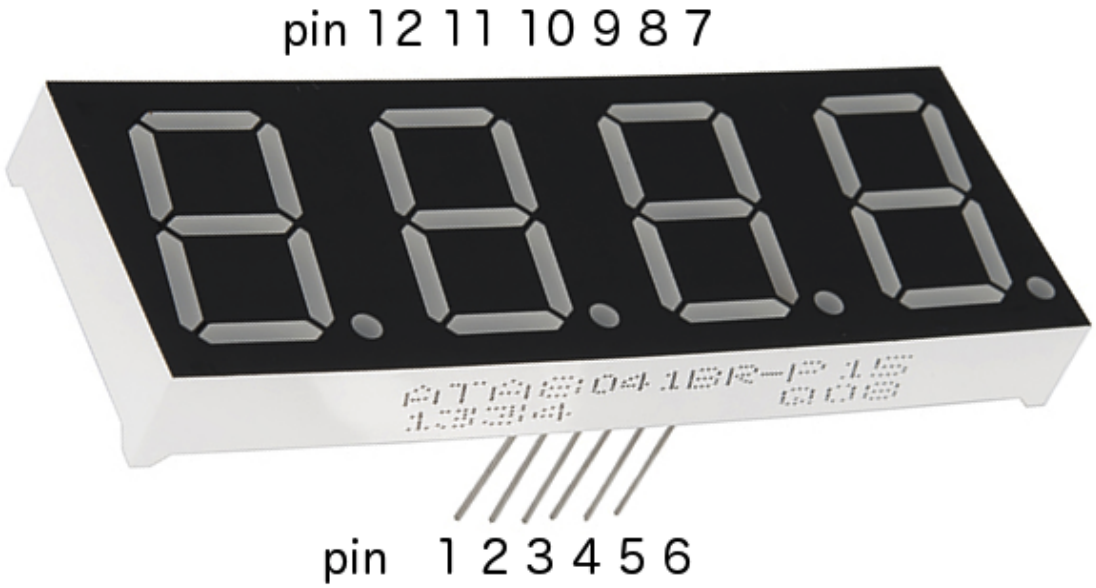


First, learn to use the Arduino's `analogRead()` function, from this example:
http://arduino.cc/en/Tutorial/AnalogReadSerial

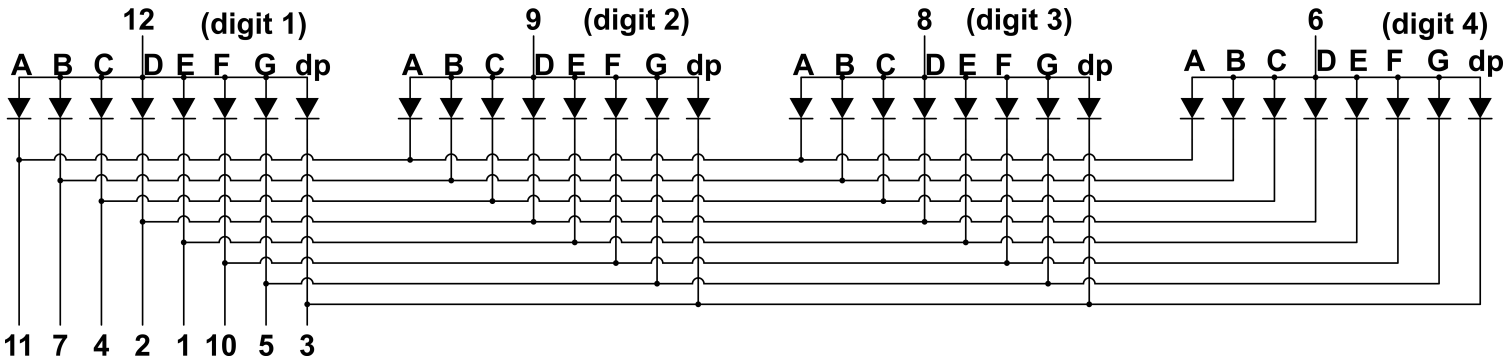
Then wire up the above circuit and code your sketch. Once you get it to work, please be sure to show it off to us! When you're done, save your sketch, because you may have a chance to build upon it later.

Optional 9: 7-segment LED display (new!)

Seven-segment LED displays are widely used as a simple way to display digits 0-9 and sometimes also letters A-F. We will use 7-segment displays during the FPGA labs at the end of the course. This device offers you a chance to get the Aruino to display numbers in human-readable form without the help of the PC serial-port monitor. The display consists of 4 digits, each of which is composed of 7 separate LED "segments" (i.e. line segments), plus a decimal point. The display is conveniently breadboard-compatible. The pins, as usual, are numbered from lower-left, increasing in the counterclockwise direction.



One would think that $8 \times 4 = 32$ separate pins (plus at least one common power or ground pin) would be needed to control 4 digits, each of which consists of 7 segments plus DP. But the manufacturers of these devices use a clever trick, shown below, to reduce the total number of wires needed to just $8 + 4 = 12$. Essentially, the anodes (+ sides) of the 8 LEDs for each digit all share a common pin, while the cathodes (− sides) of each corresponding segment (e.g. the bottom segment) for all 4 digits share a common pin. So pin 12 connects to the "+" sides of all eight LEDs for digit 1, while pin 11 connects to the "−" sides of the top segments for all four digits. So pins 12,9,8,6 select which digit is illuminated, while the other pins select which segments of the selected digit are illuminated.



The right figure below indicates the naming scheme (A–G) for the 7 segments that compose each digit. The decimal point is called "dp." The four digits are numbered 1–4 from left to right. To make the lower-left segment (segment E) of the left digit (digit 1) light up, you need to connect pin 1 ("E cathodes") to ground, and pin 12 ("digit 1 anodes") **through a** 1 kΩ **series resistor** to +5 V. **Don't forget the series resistor,** as the maximum allowed current per LED segment is 20 mA, while the voltage drop across the LED (when illuminated) is about 1.9 V.

To make digit 2 display the numeral "7" you would connect pins 11,7,4 (segments A,B,C) to ground and pin 9 (digit 2) **through a** 1 kΩ **series resistor** to +5 V.

pin	function
1	segment E cathodes
2	segment D cathodes
3	decimal point cathodes
4	segment C cathodes
5	segment G cathodes
6	digit 4 anodes
7	segment B cathodes
8	digit 3 anodes
9	digit 2 anodes
10	segment F cathodes
11	segment A cathodes
12	digit 1 anodes

https://www.sparkfun.com/products/retired/12098

Let's start by teaching the Arduino to count from 0 to 9 using only digit 1. Connect pin 12 **via a** 1 kΩ **series resistor** to the Arduino's +5 V pin. Then connect pins 11,7,4,2,1,10,5 to seven of the Arduino's available digital output pins. To light a given segment, set that output `LOW`; to dim the segment, set that output `HIGH`. To light the correct LED pattern for each digit 0–9, figure out which segments need to light up, and drive the corresponding outputs `LOW`, and the other outputs `HIGH`. The next page will be blank, in case you need space to think.

Next, try driving the anode pins for digits 1,2,3,4 (pins 12,9,8,6) from four additional Arduino digital outputs (rather than the +5 V pin), **using a separate** 1 kΩ **series resistor** for each of the four anode pins. Try counting with digit 1, then counting with digit 2, then counting with digit 3, then digit 4.

Next, try counting from 0000 to 9999, displaying the 1000's place on digit 1, the 100's place on digit 2, the 10's place on digit 3, and the 1's place on digit 4. You can use the division operator, `/`, and the modulus (remainder) operator, `%`, to extract the digits:

```
counter = counter + 1;
int digit1 = (counter/1000) % 10;
int digit2 = (counter/100) % 10;
int digit3 = (counter/10) % 10;
int digit4 = counter % 10;
```

To display a four-digit number, first display `digit1` on digit 1, for a few hundred milliseconds, then display `digit2` on digit 2, for a few hundred milliseconds, . . ., then display `digit4` on `digit4`. During each digit's display, you drive only that one digit's anode `HIGH` (driving the other anodes `LOW`), and you drive the corresponding cathodes `LOW` to light the desired pattern of segments on that digit.

It may look silly at first to display a four-digit number by lighting one digit at a time for a few hundred ms. But now change the time for which each digit is displayed to just a few tens of ms instead of a few hundreds. Your eye will be fooled into thinking that all four digits are illuminated at once, and you will see a four-digit number!

If you manage to get all of this working (!), try modifying either your capacitance meter or your reaction timer to display its output as a four-digit number on the LED display rather than on the PC's serial monitor.

If you achieve the monumental task of making this four-digit display work, you should **definitely** show it off to us!

Finally, there is some small possibility that the Arduino Uno board's power supply may not be capable of supplying the necessary current to drive this LED display. We doubt that this is true, as Jose succeeded yesterday at displaying one digit with the Arduino. Nevertheless, if you do run into this problem, there is an easy work-around using a MOSFET switch to power each anode pin from the external +5 V benchtop supply **with a** 1 kΩ **resistor in series with each anode pin**. It is the same trick that we used to make the speaker louder in optional exercise 7.