Microcontroller Lab 2 (Class 11)

In this lab you're going to learn about digital-to-analog conversion and timer interrupts. Digital-to-analog conversion is essentially the process of converting a numeric value (usually binary) to an electrical signal. The device which accomplishes this is called a DAC. DACs are widely used in many applications, including the generation of audio signals in digital music players.

Revisions:
05 April 2013 - change 2MHz square wave to 500Hz in the timer interrupt section

Table of Contents

Reading

Required:

PWM
http://arduino.cc/en/Tutorial/PWM

Sampling
http://en.wikipedia.org/wiki/Nyquist_frequency

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

Note that these are preview chapters from a new edition of the Student Lab Book, so you need a password to get them (it's on the whiteboard, and will be announced in class). Please don't share these PDFs outside of class.

Recommended:
http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem

Lab Work

Using the PWM DAC

The Arduino has a simple built-in DAC (digital-to-analog converter) which uses the technique of pulse-width modulation (PWM). Please read the brief tutorial on PWM if you haven't.

▢ **Lab 2.1**

Build a pulse width modulation DAC
Wire an LED between Arduino pin 3 and GND.
Write a sketch which calls analogWrite() to pin 3 with various values from 0 to 255.
What happens to the brightness of the LED? Connect an oscilloscope to pin 3. You should see a train of pulses. What is the frequency?

Make a table in your notebook with 3 columns as follows:

PWM Value	**Pulse Width**	**Voltage**
0		
64		
128		
192		
255		

Set the PWM to each of the values suggested and measure the pulse width.
Often it is useful to get a programmable DC voltage from a PWM signal.
You can use a low-pass filter to convert the pulse train to a DC value.

▢ **Lab 2.2**

Design a *low-pass RC filter* with a cutoff frequency which you think is appropriate to get a DC value from the pulse train. Test it. If you still see pulses, the frequency is too high. Experiment with different R, C until the output is smooth (less than maybe 100mV of ripple) What component values did you end up with? What is the cutoff frequency?

Fill in the third column of the table with your filter connected (use a volt meter).

The filter you have designed is a common feature of systems which generate waveforms from digital data (like mp3 players).

Limitations of PWM -- PWM works well applications where speed is not important (such as dimming LEDs or controlling motor speed). It is not so good for generating waveforms.

▢ **Lab 2.3**

Measure the maximum frequency you can output with your filtered PWM DAC
With the oscilloscope connected to the output of your filter, enter and run the following sketch:

```
int d = 1;                        // half-cycle delay in ms
void setup() { }
void loop() {
     analogWrite( 3, 0);          // set PWM to minimum
     delay( d);
     analogWrite( 3, 255);        // set PWM to maximum
     delay( d);
}
```

This sketch will output a square wave (in principle) with a frequency of 1/2d where d is the delay for one half-cycle in ms.

If you set the delay very large, you should see a 5V square wave. Experiment to find the delay at which the amplitude drops by 3dB (1/sqrt(2)). How does this compare with the cutoff frequency of your filter?

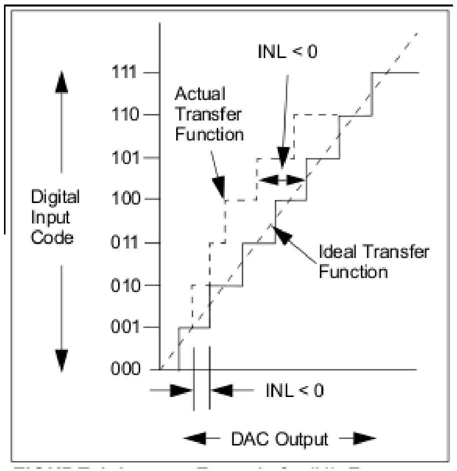
Would this circuit be good for generating sound? Why or why not?

Using the Serial DAC

To overcome the speed limitations of PWM, we can use an external DAC.

The DAC we will use for this lab is the Microchip MCP4921 (link to data sheet).
Please open the data sheet and *read now* at least pages 1-2 and 17-26.

This is an 8-pin chip which should be in the kit of accessory parts. The MCP4921 is a 12-bit serial converter with an SPI bus interface. It accepts a 12-bit binary number (decimal values 0-4095) and converts it to a voltage from 0-5V (with normal gain settings). The graph below illustrates the transfer function of a typical DAC.



To communicate with the DAC you must use a specific protocol called SPI. To quote page 23 of the data sheet:

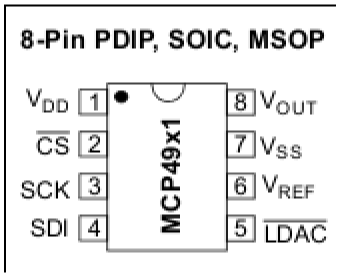
> "The write command is initiated by driving the CS* pin low, followed by clocking the four Configuration bits and the 12 data bits into the SDI pin on the rising edge of SCK. The CS* pin is then raised, causing the data to be latched into the DAC's input register."

Pin Connections

For a data sheet, this is actually quite a clear explanation! Let's translate it. First, looking at the pinout of the chip below, identify the pins you need:

SDI - pin 4	Arduino pin 11
SCK - pin 3	Arduino pin 13
CS* - pin 2	Arduino pin 10

You will need to connect those pins to digital pins on your Arduino so you can control them. I suggest using the connections above (for reasons we'll get to later).



You also need to connect some other pins. First, where is the ground pin?! Microchip has confused us by naming it VSS. This name has historical significance (based on the name of the source terminal of a FET). How would you figure out it is actually GND? A good clue is on page 3 of the data sheet under the **ELECTRICAL CHARACTERISTICS** table where it says: "Unless otherwise indicated, VDD=5V, VSS=0V". So, connect those pins too:

VDD - pin 1	+5V
VSS - pin 7	0V (GND)

Finally, what about the other pins? You can often figure out what to do with them by reading the

"PIN DESCRIPTIONS" section. See page 17 of the data sheet. For **LDAC** it says "This pin can be tied to low (VSS)..." so let's do that. **VREF** provides the full-scale voltage and as they suggest it may be tied to VDD (5V). So now we have:

LDAC* - pin 5	GND
VREF - pin 6	+5V

▢ Wire the DAC as given above to your Arduino and the breadboard.

Be sure you connect all the pins somewhere (pin 8 should go to a volt meter)
Make sure that a GND pin on the arduino is connected to breadboard ground.

Serial Protocol

Now we need to figure out what signals to put on the digital pins to send information to the DAC. See the figure below from page 25 of the datasheet. Be careful; there are figures for a few different parts on this page, and we want the one for the MCP4921.

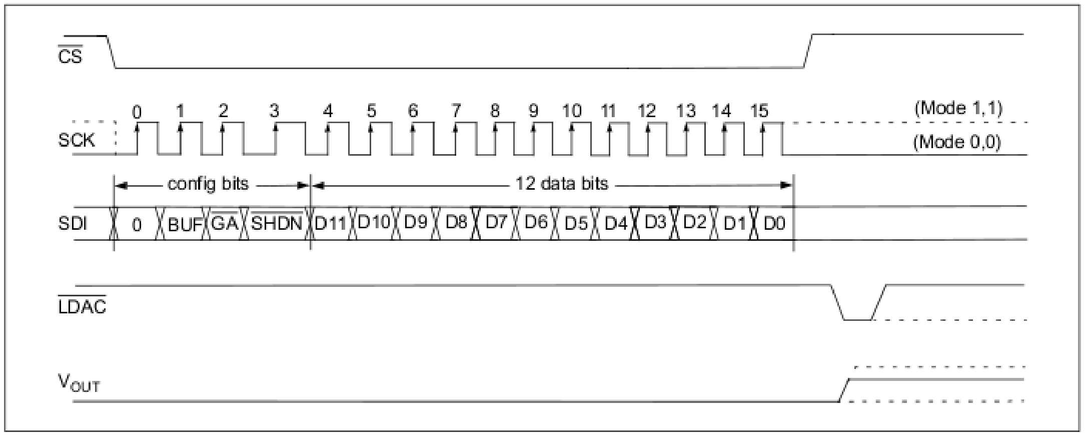
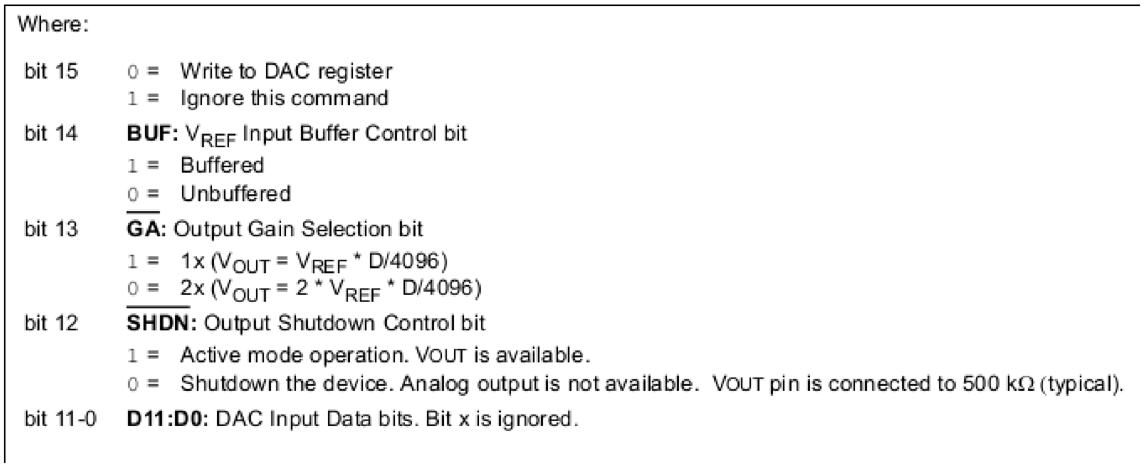


FIGURE 5-1: *Write Command for MCP4921 (12-bit DAC).*

This diagram tells us that we need the following sequence of operations to load a value into the DAC:

1. Initialize (in **setup()**) by setting nCS to HIGH and SCK to LOW.
   We don't have to worry about nLDAC as we have tied it to GND
2. Set nCS to LOW
3. For bits 0 to 15 (loop):
   a. Output data (0 or 1) on SDI
   b. Set SCK to HIGH
   c. Set SCK to LOW
4. Set nCS to HIGH

Note that in step 3 you need to provide 16 different bit values. The first four (BUF, GA* and SHDN*) are described on page 24 of the data sheet:



Where:

bit 15	0 = Write to DAC register 1 = Ignore this command
bit 14	**BUF:** V_{REF} Input Buffer Control bit 1 = Buffered 0 = Unbuffered
bit 13	**$\overline{\text{GA}}$:** Output Gain Selection bit 1 = 1x ($V_{OUT} = V_{REF} * D/4096$) 0 = 2x ($V_{OUT} = 2 * V_{REF} * D/4096$)
bit 12	**$\overline{\text{SHDN}}$:** Output Shutdown Control bit 1 = Active mode operation. VOUT is available. 0 = Shutdown the device. Analog output is not available. VOUT pin is connected to 500 kΩ (typical).
bit 11-0	**D11:D0:** DAC Input Data bits. Bit x is ignored.

Reasonable choices for these bits would be:

bit 15 = 0	
bit 14 = 1	(this is unimportant)
bit 13 = 1	(set output scale to 0-5V)
bit 12 = 1	(enable the output)

Note that the bits with bars over their name are “active low” meaning the named function is active when the bit is set to 0 or “low”. When typing we usually write **nSHDN** or **SHDN***.

So, to set the DAC you would need to send the bits 0, 1, 1, 1 given above followed by twelve bits representing the desired output voltage.

Now you need to write a sketch for the Arduino which produces the signals above. Let’s start with CS*. From your previous lab, you probably remember that you can do this like so:

```
int dac_nCS = 10;                // pin number for nCS

void setup() {                   // setup function runs once
  pinMode( dac_nCS, OUTPUT);     // define nCS as an output
}

void loop() {
  digitalWrite( dac_nCS, 0);     // set nCS to 0 (LOW)
  digitalWrite( dac_nCS, 1);     // set nCS to 1 (HIGH)
}
```

☐ Enter the sketch above and download it.

Connect your oscilloscope to the **nCS** pin.

You should see the CS* pin switching on and off rapidly. A pin value of ‘1’ is indicated by a high level on the diagram. Now you need to make the other signals switch on and off in the order given above. You will add the code to do that between the two digitalWrite() calls which turn nCS on and off.

▢ **Lab 2.4**

Write a sketch to set the DAC to 2.5V
Follow the diagram in Figure 5-1 above.
At first, output the fixed binary value 1000 0000 0000 to the DAC.
This should result in a voltage of 2.5V on the output (pin 8).

▢ **Lab 2.5**

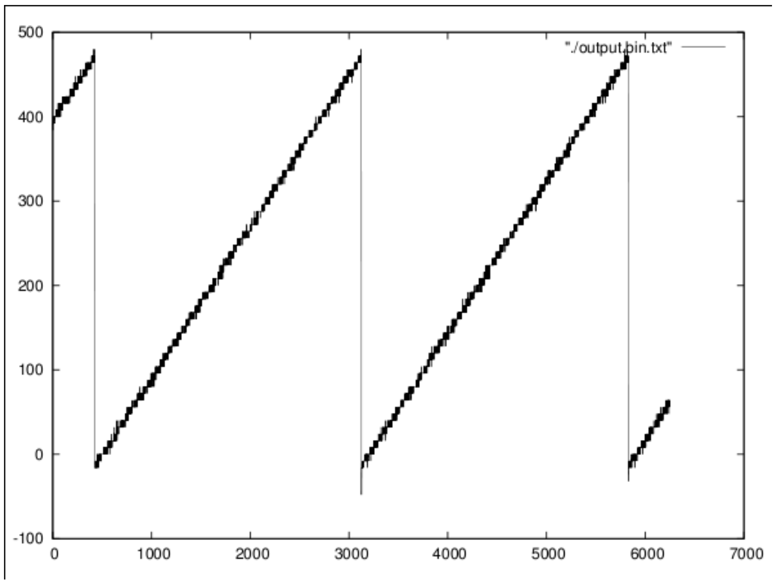
Write and debug a sketch to output a ramp from 0-5V on the DAC.

Use your oscilloscope as needed to debug it if it doesn’t work. You’ll need both channels. Use one to trigger on the falling edge of **nCS** and look at the other signals to be sure they look like the timing diagram.

Here are a few programming hints:

- use the following to test a binary bit in an integer value:
```
int n;          // integer value
int b;          // bit number i.e. 0-15
if( n & (1<<b)) // test if bit b is ‘1’ in n
     ...do something
```
- use a for() loop to output the 12 bits of data

The scope output should look like the plot below, with a peak-to-peak amplitude of 5V. Using my sketch, the frequency is about 10Hz, but yours may be different.



▢ **Lab 2.6**

Convert the main part of your sketch into a function
(Read about Arduino functions if you need review).
Use this function declaration as a prototype:

```
void setDac( word v)
```

The function argument v is declared as a **word** because it is specified to hold a 16-bit value, rather than an **int** whose size is not specified.

Test your function by writing a sketch which outputs various values to the DAC in succession, and viewing the output on the oscilloscope.

How long does it take to output a new value? (Hint: measure this by programming alternating large and small values and measuring on the oscilloscope).

Save this sketch to turn in (name it `SPI_DAC_bitbang`)

▢ **Lab 2.7**

Make a new version of the sketch using the Arduino SPI library (reading link).
This requires several changes to your code. First you must add:

```
#include <SPI.h>
```

at the top of the sketch to include the function declarations for the library.
Next, you must add:

```
SPI.begin();
SPI.setBitOrder( MSBFIRST);
```

to your **setup()** function. The first line initializes the library. The second tells the library to output the MSB (most-significant bit) first as shown in the timing diagram from the datasheet. Finally, you have to call the function:

```
SPI.transfer( data);
```

for each byte (8 bits) to be sent. Note that the DAC requires two bytes (16 bits) to be sent so you have to call the function twice for each update of the DAC. Also note that the library does not take care of the **nCS** signal, so you need to handle that in your own code.

Save this sketch to turn in (name it `SPI_DAC_library`)

If you have trouble getting the code using the library to work, don't forget that you need to take care of the BUF, GA* and SHDN* bits. Ask for help if you need it.

Finally, make a function called `setDac()` which works like the one you wrote before.
Output a ramp from 0-5V as you did in Lab 2.5.
Compare the speed. Is it faster?

The SPI library uses dedicated hardware in the microcontroller called the UART (Universal Asynchronous Receiver-Transmitter). You can read about it starting on page 199 of the ATMega328 data sheet. The UART is a bit complicated if you have to set it up yourself, but with the the Arduino SPI library it is relatively simple.

☐ **Lab 2.8**

Load your sketch `SPI_DAC_library`.
Modify your main loop to output a square wave like this:

```
void loop() {
      setDac( 0);
      setDac( 4095);
}
```

Trigger the oscilloscope on rising edge and look at the output. You should see a square wave with a frequency of about 30kHz. Adjust the oscilloscope so that 10-20 cycles fit on the screen. Notice how the wave "jumps" sometimes and the period seems to change? This is because periodically the AVR microcontroller is *interrupted* and goes off to do something else, causing a delay in your sketch. This is troublesome if you want to create periodic waveforms. There is a way around this...

Timer Interrupts

An *interrupt* is an event which causes the processor in a computer (such as the Arduino) to stop what it is doing and temporarily do something else. These can be a bit confusing, but you can think of an interrupt as a magic way to arrange for a function to be called when an external event occurs. One useful type of interrupt is a *timer interrupt*, which is an interrupt which occurs at a fixed time interval. Let's use one of these to clean up the jitter in our waveform generator.

We'll need a library called Timer1. Please read the introduction on the timer 1 page now. You'll need to *install* the library before you can use it. Read how to do it here.

- ▢ Install the library Timer1

  Create a copy of your sketch `SPI_DAC_library` and call it `SPI_DAC_interrupt`.

  Add the following code to the top of your sketch:

  ```
  #include <TimerOne.h>
  ```

  (you can use the menu **Skech->Import Library->Timer One**) to do this for you. This defines the functions used in the TimerOne library.
  Then add the following lines to your **setup()** function:

  ```
  Timer1.initialize( 250);     // timer interrupt every 250us
  Timer1.attachInterrupt( timerFunc);  // call every
  interrupt
  ```

The first line initializes the timer to trigger an interrupt every 250 microseconds. The second line specifies that the function `timerFunc` should be called when the interrupt occurs.

What do you put in `timerFunc`? Code to update the DAC output and calculate the next value. Here is an example for a square wave output:

```
volatile int dac = 0;
void timerFunc()
{
  setDac( dac);             // update the DAC output
  dac ^= 0xfff;             // XOR (toggle) all 12 bits
}
```

This code will alternate between 0 and 0xfff (all 1's) output to the DAC each time it is called.

- ▢ **Lab 2.9**

  Make a sketch which uses the timer interrupt to output a square wave
  at exactly 500Hz using the suggestions above

  Save this as `SPI_DAC_Interrupt`

▢ **Lab 2.10**

Make a new version which outputs a sine wave.

Use `Timer.initialize( 250)` to set a 250us interrupt period.

Use the sin() function in the Arduino library to calculate a new value for the output each time **timerFunc()** is called. Declare a **static float** variable called **a** and pass it to the **sin()** function, which takes an argument in radians and returns (of course) a value fro -1 to 1. Here is a bit of code to get you started (this code will need to go in your **timerFunc**:

```
const float pi2 = 3.1415927 * 2.0;
static float a;      // sin angle in radians
static float da = pi2/8;  // delta-a (increment value)
unsigned int d;      // DAC value 0-4095 (12 bits)

d = 2048.0 + 2047.0 * sin(a);
setDac( d);
a += da;
if( a > pi2) a = a - pi2;
```

Add the code above to your sketch in **timerFunc**. Connect your oscilloscope to the DAC output (pin 8). Draw the resulting waveform in your notebook with the axis labeled. What frequency and amplitude do you see?

Set two parameters at the top of the sketch:

```
float freq = 500.0;         // output frequency in Hz
float amplitude = 1.0;      // output amplitude pk-pk in V
```

▢ **Lab 2.11**

Modify your sketch to use the fwo parameters above to modify the output of your sketch.

Save this as `SPI_DAC_Sine` to turn in.

Congratulations! You have made a programmable sine wave generator. What are it's limitations? Here are some things we might like to improve:

- Clean up the stair-stepping in the output
- Speed it up so that frequencies above a few kHz can be generated

We’ll work on these next week. *Please leave the DAC connected as you’ll need it next week.*

Extra Credit

▢ **Lab 2.12**

Modify your sketch or make a new version which can output square, triangle and ramp waves with adjustable amplitude and frequency

▢ **Lab 2.13**

Add controls to allow a user to change the settings. Some ideas:

- switches to select sine, square, triangle outputs (with LEDs to indicate the mode)
- push-buttons to raise and lower frequency and amplitude
- digital display (from the voltmeter project) to display what is being set

You may run out of output pins. You could use a shift register or latch logic chip to drive LEDs or the digital display.