Skill Demo 4: ADC and Interrupts

V1.7

Goals:

- Learn about the analog-to-digital converter built into our microcontrollers
- Learn about potentiometers, photoresistors, and temperature sensors
- Capture data and stream it to a USB host, graph the captured data
- Learn about Interrupts and Interrupt Service Routines.

Tools/supplies:

- Breadboard
- Wires
- Mega microcontroller development board
- Amplified microphone module
- 1x potentiometer
- 1x photoresistor
- 1x 1M resistor
- 1x 100k resistor

4a: Basic Analog-to-Digital Converters

Background

Potentiometer

A potentiometer, informally a 'pot', is (usually) a three-terminal resistor with a sliding or rotating contact that forms an adjustable voltage divider. If only two terminals are used, one end and the wiper, it acts as a variable resistor or rheostat. There is also a measuring instrument called a potentiometer, which is essentially a voltage divider used for measuring electric potential (voltage); the component is an implementation of the same principle, hence its name.

Photoresistor

A photoresistor is a light-sensitive resistor. Incident light will change the amount of current that can flow through the device. You already used one of these in SD3.

Sensing Analog Voltage

A potentiometer, photoresistor, or other sensor can be set up to output analog voltages. To sense an analog voltage like this with a microcontroller, we'll use an analog to digital converter, often called an ADC. The ATmega2560 (like most microcontrollers) has one built in, and we can use it with the analogRead() function within the Arduino libraries. The analog to digital converter

is specialized hardware within the microcontroller IC, and is only connected to particular pins on the board.

For more about the Arduino ADC, take a look at [this write-up].

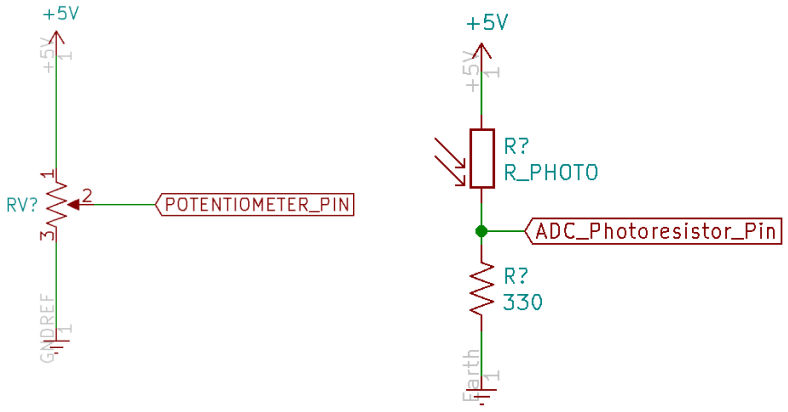
Procedure

Before you start, fill in the Response Page on the answer sheet to learn about the Arduino's ADC. (20/100 points)

Your goal is to write a program that reads a potentiometer's position and the voltage level from a simple light sensor, and relays that to the Arduino Serial console. Your program should output lines with the format:
`PotRaw: <raw ADC value>, Pot: <calculated pot voltage>, Light: <raw light ADC value>`

Connections

You will connect the potentiometer and photoresistor to any two of the analog input pins. These are noted with the numbering A0, A1, etc.



Reading the potentiometer

With your multimeter, determine which legs of the potentiometer perform what function. Turn the potentiometer to the middle of its range, then find the ends of the resistor element. Measure the resistance across two of the terminals. At least one pair of terminals will read 10K ohms. (Note: if this is a 'stereo' pot with more pins, two pairs may read 10K).

Next find the adjustable center terminal or "wiper". With your multimeter probe on one 10K end terminal, find a terminal that shows a resistance that is less than 10K. The resistance should change when you turn the potentiometer.

Plug the potentiometer into your breadboard and hook up the three terminals according to the above schematic. The center potentiometer pin should be connected to an analog input on your Arduino. Use analogRead() to read the ADC value at this pin, and convert it to the voltage on the potentiometer pin.

Reading the photoresistor

Connect the photoresistor and resistor to a second analog input as shown in the above schematic. This is similar to how we connected the photoresistor in SD2 and SD3, except now we can use the ADC hardware.

Use a 10K resistor for the photoresstor's pull-down resistor. (this value was selected through trial and error for the light level in our lab).

Convert the ADC values of the sensor readings to voltage and print them in a loop, once per second.

Testing your serial program

Now try the following:

- Turn the potentiometer. What are the min and max readings? Does this match what you read with a multimeter?
- Expose your light sensor to a bright light, room light levels, and darkness (covered with an object). What are the raw values?

4b: Interrupt-driven ADC sampling

The Task

Create a program that will detect audio tones of a particular frequency. Pick a musical note between C5 (523.25 Hz) and C6 (1046.5 Hz) to try to detect. Ensure that you don't trigger on sounds that are a musical semitone higher or lower than your chosen note. You can use a phone or laptop as a tone generator (sine waves please!), and you can hold your speaker close to the microphone.

If you hold your phone close to the mic, you don't have to play the tone very loud.

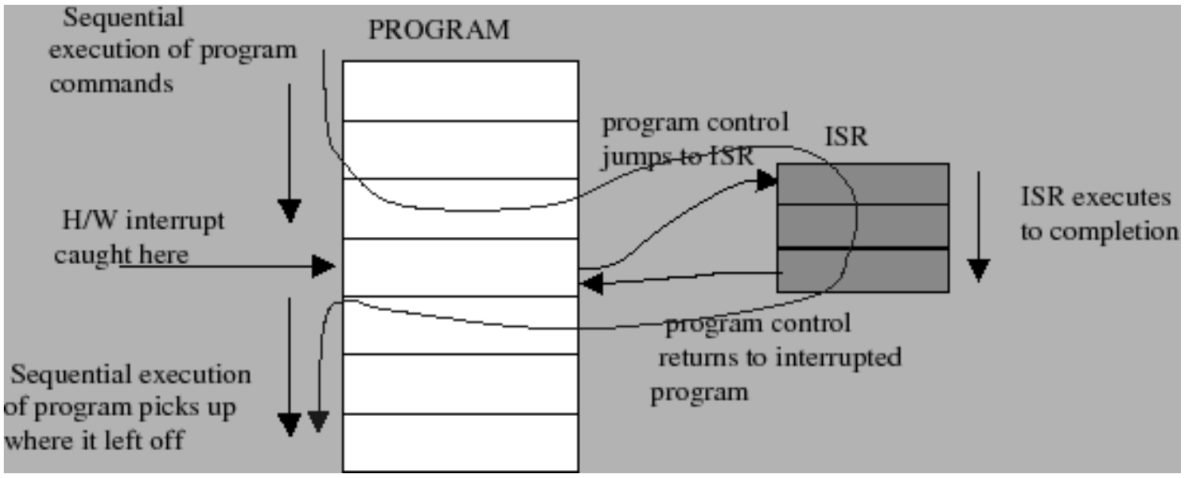
You may want to pick notes that are different from the people around you.

Background

Interrupt Driven ADC

As we have discussed we have no operating system running on the microcontroller. This means that if we want actions to take place with specific timing, or handle multiple threads of execution, it is up to us to support it.

One way to do this is with an interrupt. This is a signal inside the microcontroller that occurs when a predefined event occurs. An event could be a timer hitting a certain value, a GPIO input changing state, an Analog to Digital conversion finishing, or many other things.



Program execution

When the interrupt occurs, the processor stops whatever it is doing, saves the execution state, and jumps execution to a specified function. The function called in response to an interrupt is called an interrupt handler or interrupt service routine (ISR). When the processor finishes the function, it reloads the previously saved execution state, and then continues from where it was interrupted.

Timers under Arduino

In order to detect tones accurately, you'll need to record ADC samples at a consistent rate. Too fast or slow, and your detection won't be pitch accurate.

The timer/counter hardware in the ATmega can help with this: it can be configured to count upwards independent of the CPU, and can generate an interrupt every N counts. (See the ATMega 2560 datasheet, chapter 17, '16-bit Timer/Counter (Timer/Counter 1, 3, 4, and 5)' for more info on the low-level hardware)

One way to use this hardware is with the TimerOne library, which you can install with Arduino's library manager. It gives us an easy way to configure how often interrupts are generated by the Timer/Counter 1 hardware, and a way to attach a function of our own to be run when the interrupt occurs.

Take a look at the TimerOne docs and example to see how it works.

Connect a Microphone

In your kits, there is a "sound sensor module", basically an electret microphone with some adapter circuitry to make it easier to work with. Plug that into your Arduino so that the module has 5V power and GND, and connect the analog output of the module to any analog pin on your Aduino.

Start by capturing audio samples without an ISR to make sure the ADC is working. In a simple loop, capture 512 analog samples into a buffer as quickly as possible, then calculate the minimum and maximum value in that buffer.

You should be able to tune the average output voltage of the audio module using the little blue 100-turn potentiometer on board. Tune this so that you're getting around VCC/2 (2.5V) into the Arduino's ADC.

You should also be able to see the minimum and maximum values change when you make loud sounds right next to the mic.

You can also print the contents of your buffer over serial, and use the serial plotter to visualize it. You should be able to play a tone and see a sine wave.

Use some method to measure how quickly your sampling loop is running. Note that the CPU is spending all of its time checking if the ADC hardware is ready yet. **This is as fast as we can easily run the ADC on our Arduino.**

For example, you could:

- Use millis() or micros() to record the time when you start or stop sampling, and calculate the sample rate from that.
- Flip a GPIO pin every time you take a sample, and look at that GPIO pin with an oscilloscope or multimeter.
- Play a known-frequency tone at your mic, graph the contents of your buffer, and look at the period of the wave to back-calculate the sample rate.

Set up a Timer Interrupt

In a separate program, set up an interrupt handler to run at a consistent rate, which we can later use to start ADC samples. Do this however you want, but the TimerOne library mentioned above is recommended.

(technically, what we provide to the TimerOne library is a callback, and the actual ADC interrupt function is inside the TimerOne library. This is done so that the program doesn't crash if you forget to attach a callback function.)

Now, choose a sample rate for sampling your ADC pin. You need to sample at least 2x as fast as the tone you are trying to detect. You should also sample a little slower than the interrupt-less version we ran before. We want to make sure the ADC finishes taking one sample before we ask it to take the next one.

Set up a timer interrupt to turn on the pin 13 LED (the one on the Arduino) whenever it enters the handler, and turn off the LED when it exits the handler. You may want to temporarily add a tiny delay (delayMicroseconds(10) or similar) so that your LED stays on enough of the time to be visible.

Keep in mind that we don't want to leave the processor running code inside of an interrupt handler. While in an ISR, other computation won't be able to take place, including other time-critical ISR's. Don't use any blocking calls, Serial.print()'s, or long delay()'s.

Asynchronous ADC Sampling

The default Arduino analogRead() function starts an ADC sampling, and then busy-waits until it is finished, usually around 100 µs. This isn't suitable to use within an interrupt, since we don't want to 'block' in an ISR.

The ADC in the Mega (and most ADC's) can run independently of the main CPU. Our code can start the ADC sampling, and then check back later when it's finished.

Take a look at the AsyncAnalog library (install-able from the Arduino Library Manager). It's an alternative to analogRead() that breaks it into .start(), .ready(), and .value() methods. You can see how it is used in its example.

Combine your timer interrupt code with asynchronous ADC sampling to record samples with consistent timing, and store them into a buffer. When the buffer is full, set a variable so that your main loop knows to start processing. In the main loop, print the values from that buffer via serial. (You can pause sampling while you process.)

(hint: each time the timer tells you to take a new sample, the previous sample should be ready)

Notes on use of 'Volatile'

One thing you will have to deal with is synchronizing the data between your main loop (setup() and loop()) and the interrupts and callbacks. It's entirely possible for an interrupt to happen and modify your buffer's values while you're in the middle of reading them.

The C compiler used in the Arduino environment is great at compiling fast assembly code. However when multiple execution paths exist, the compiler might assume a variable won't change, and might make mistakes. We can fix this with the use of the 'volatile' keyword. This alerts the compiler that the variable could be changed by another execution path and forces the state of the variable to be saved to/read from memory every time it's accessed. In short, use the volatile keyword on any variables that are read or modified in multiple threads.

With 'volatile', you can use shared global variables to pass data between execution threads, but you might also consider using nointerrupts() and interrupts() to pause interrupts, keeping any of the ISR's from modifying your buffer while you're reading from it.

(The 'correct' answer is to use proper thread synchronization, but that's a longer discussion. 'Volatile' is a quick fix for embedded systems.)

Do some Frequency Analysis

Finally, you are capturing audio samples into a buffer with a known, consistent sample rate. Replace the code that prints the buffer with code that looks at your 512 analog samples and detects if there is much of your tone in there.

There are many ways to do this. This isn't a DSP class, so you will probably want to use an existing implementation.

One way is to use the Goertzel algorithm. It basically looks to see how well your samples line up with a sine wave of the desired frequency. The math for Goertzel is a bit of a pain, so feel free to look at this article. The algorithm is nicely implemented by the goertzel arduino library.

Finally you can also use a very simple approach that doesn't require a library. A tone without much noise can be detected by counting zero crossings of the signal.

```
int zeroCrossing(int data, int n, int thresh)
{
	int i, count;
	for(i=1; i<n-1; i++)	/* loop over data  */
		if((data[i-1] < thresh)&&(data[1] >= thresh))
			count++;
	return count;
}
```

This can produce a noisy result and must be converted to a frequency (count / buffer length in seconds).

Skill Demo 4: ADC and Interrupts

NAME_____ GTID_____
☐ CS3651A
☐ CS3651B

Before you start, fill in the Response Page on the answer sheet to learn about the Arduino's ADC. (20/100 points)

4a: Basic Analog-to-Digital Converters Checkoff

Implement the potentiometer and light sensing circuits with two ADC pins.

Potentiometer

Adjust the potentiometer to somewhere between min and max.

What is the raw value? _____ Calculated voltage _____

What is the actual voltage as measured with your multimeter? _____

Photoresistor

(Use a 10k for the pulldown resistor)

What is the raw value under a bright light? _____

What is the raw value with room illumination? _____

What is the raw value in the dark? _____

30/100 Sign-off Initials_____ Date_____

4b: Interrupt-driven ADC sampling Checkoff

Demo your Arduino successfully detecting a tone. It's sufficient to print out a number representing how much of the tone you detect, but you can turn on an LED to indicate that you're above a threshold too.

Be prepared with the following to make the checkoff easier to grade (or easier for us to help debug):

What note did you choose?

What frequency is that?

How fast was your Arduino able to run analogRead() when in a tight loop?

What rate are you sampling at with the timer hardware?

If you have trouble, turn on a GPIO pin (like 13 with the LED) when you are in the timer ISR, so we can check that with an oscilloscope or multimeter. It doesn't have to be visible to the naked eye, but is handy for debugging.

50/100 Sign-off initials:_____ Date:_____