Introduction to Microcontrollers: Exploring Interfaces

Introduction

With the proliferation of modern microcontrollers, control of electromechanical systems has moved from dedicated circuitry and specialized hardware to more general implementations where a small computer – the microcontroller – is programmed to work with various sensors and actuators using high-level languages (e.g., C or BASIC). One of the primary benefits is that microcontrollers are very flexible, and that flexibility allows the designer to quickly change parameters and try them out without having to significantly redesign the system hardware (think prototyping!).

In this lab, you'll explore different ways to get information from a sensor into the microcontroller and perform some basic processing. Specifically, you'll determine whether or not light is hitting a photoresistor and use it to make a simple night light. You'll also use the oscilloscope to probe the circuit so you can learn more about its functionality.

Objectives

Upon completion of this lab, the student should be able to:

- Program an Arduino to perform digital input and output,
- Program an Arduino to perform basic control and logical functions,
- Demonstrate an understanding of an RC circuit,
- Use an oscilloscope to observe the dynamic response of an electronic circuit, and
- Use a comparator to add precision to a sensor reading.

Preparation

Basic laws of capacitors and RC circuits. Review RC circuits from your Physics classes. Read Carryer *et al.*, Sections 9.6 up to and including 9.9.2. Pay particular attention to anything related to RC circuits in the time domain. Read the rest of Section 9.9 if you want. Read 9.14 on real capacitors, if you're interested.

You must complete the pre-lab worksheet before you come to lab. The worksheet is an individual assignment.

Analog vs. Digital

To begin with, we'll recap digital and analog interfaces. There are four potential kinds of connections between a microcontroller and the outside world: digital in, digital out, analog in, and analog out. Physically, these interfaces are through wires, called *pins* on the microcontroller, and each pin is addressed through internal circuitry, which sets or reads *voltages* at the pins, based on whatever program has been put on the microcontroller by a programmer.

- *Digital output* is probably the easiest of the interfaces. Digital output can either be LOW, in which case the output pin is connected to ground (0V) via internal circuitry, or HIGH, in which case the pin is connected to whatever the supply voltage is (5V, in your system). In Arduino, digital output is accomplished by setting the pin as an output with `pinMode()` and then using `digitalWrite()` to set the pin HIGH or LOW. Note that if you pass enough current through a pin, the voltage might differ from 0 V or 5 V by a few tenths of a volt.
- Perhaps surprisingly, *digital input* is a little more complex than digital output. The reason is that there is no guarantee that the voltage applied externally to a pin will be exactly 0V or 5V. The circuitry must tolerate variances from defined HIGH and LOW voltages, leaving a gray area where the reading is ambiguous. In Arduino, reading a digital value is accomplished by setting the pin as an input with `pinMode()`[1] and reading it with verb—digitalRead()—.
- *Analog input* is generally accomplished by means of an *analog-to-digital converter*, or simply, an *ADC*. An ADC takes a continuous range of voltages and divides it up into a discrete set of numbers, where the size and range depends on the specifics of the ADC. The conversion formula has been covered elsewhere. Reading an analog pin is accomplished using `analogRead()`.
- *Analog output* is probably the most complicated of the interfaces. In fact, it's complicated enough at the hardware level that with the Arduino Uno, analog output isn't analog at all – it's digital! Essentially, internal timers switch the voltage back and forth between HIGH and LOW at a very high rate such that the *average* voltage approximates the desired analog voltage. We'll cover this method, called *pulse width modulation* (PWM) later in the term.

In lab

Lab activities will center around making a night light. Although not the most advanced system you can make, it allows us to explore a lot of the functionality of a microcontroller.

Night light with an ADC

Here, you'll use the analog-to-digital converter again to determine if it is dark (and the night light is needed) or light.

[1]When a program is started, all of the digital pins default to input pins, though it is still a good idea to declare them as such, just for clarity.

Procedure

You will use this circuit for several experiments today, so be sure to place it in a convenient spot on your breadboard.

1. Begin building circuits in Figure 1. When you insert the photoresistor into your breadboard, put it about 3-4 cm from the LED, but don't connect it to the resistor, R, just yet. Gently, bend it and the white LED so that they face each other – you'll want them close enough to get a good signal while still being able to pass a pencil or a finger between them without bumping either. Once you've arranged them, try not to bump either component so that the measurements you are about to take will consistent for the rest of the lab.

2. Using jumper wires (so you don't disturb the photoresistor) and a DMM, measure and record the resistance of the photoresistor when the LED is on and when it is off. This will correspond to "day" and "night" for your nightlight.

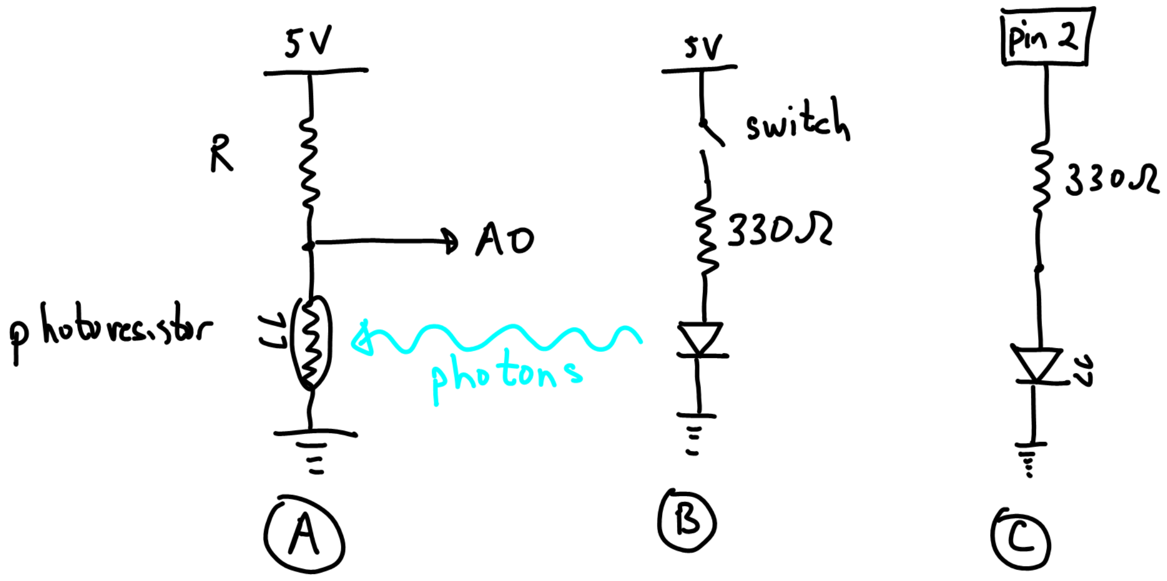

Figure 1: Circuits for the night light. Circuit A is the photosensor; B is a manually controlled LED to make it "day" or "night"; C is the night light, which you'll control from pin 2.

Resistance (day): ______________________

Resistance (night): ______________________

3. Choose a fixed resistor, R, with a value that is near the geometric mean of the two values you measured before and insert into the circuit. It can be shown that doing so will give the largest voltage swing at the junction.

4. Write a program that uses the ADC to read the voltage at the junction every 500 ms. Print

the state ("light" or "dark") to the Serial Monitor and turn the "night light" on or off appropriately. That is, you'll have to find a useful threshold and write the logic to turn the light on and off.

Demonstrate your working system to an instructor.

Basic RC circuit

In the above example, the light intensity is clearly an *analog* value. Given such a problem, it's quite common to use the ADC to measure voltages, but it's not the only way. In this next experiment, you'll use an RC-circuit to measure changes in resistance. Such a technique is invaluable to measure changes in *capacitance* if you happened to have a capacitance-based sensor.

First, you'll analyze a basic RC-circuit, then you'll put the theory to use to measure light levels.

Procedure

1. You've been provided a nominal 1000Ω resistor and a $0.1\mu F$ capacitor. Use a DMM to measure the actual resistance and capacitance of the components and calculate the rise time for the circuit in Figure 2. Also calculate the time-constant.

   Calculated rise-time (10 → 90%): ____________________________

   Calculated time-constant: ____________________________

2. Without disturbing your existing circuits, construct the circuit in Figure 2. The circuit will be driven by a digital output pin. We'll use the oscilloscope to simultaneously measure the input voltage on CH1 and the output voltage between the resistor and capacitor on CH2.

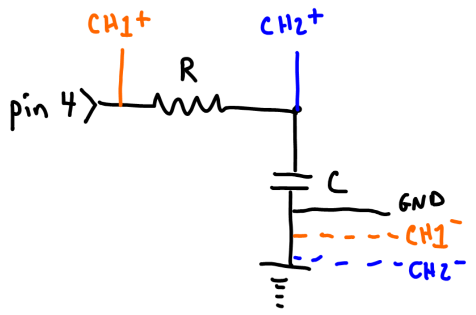

Figure 2: RC circuit with connections to the Analog Discovery labelled. GND here refers to the black wire on the Analog Discovery (denoted by an arrow on the device). Connect the GND from the Arduino as normal.

3. Write a program that sets pin 4 as an OUTPUT and toggles the state between HIGH and LOW every 1 ms. Use `delay()` to perform the timing. Check that you won't exceed the current limits of an output pin.
4. In the Waveforms software, run the oscilloscope and set the trigger to Normal. Adjust the settings to capture a rising signal on CH1 with a threshold of 2V.
5. Explore the functionality of the Waveforms software by adjusting the timescale and offsets. Note how you can adjust the various levels by grabbing the appropriate triangle and dragging it. One thing you will probably want to do is go to Options → Display and change the offsets for both Voltage and Time to "Divisions." This will prevent the draggable triangles from disappearing when you drastically change the scales.
6. Adjust the oscilloscope settings so that you have a good view of the voltage on CH2. Eyeball the rise-time. Does it look reasonable, given your calculations? If not, check your connections and re-calculate the rise-time from the pre-lab. Small variations are normal; being off by an order of magnitude indicates something is wrong.
7. In the oscilloscope program, find the "Measure" option and explore the built-in measurements provided in the software. Dig around until you find "Rise Time" and add the measurement (for the channel connected to the output) to your list of measurements. How does the measured rise-time compare to your calculations? Give possible explanations for any discrepancies.

   Measured rise-time: ____________________________

8. Set the trigger to capture on the falling edge and calculate the fall-time. Is it different from the rise-time? Explain.
9. Change the trigger to CH2. Grab the trigger level triangle and drag it up and down a little. What happens?

The RC timer

When you measured the rise and fall time of the RC-circuit in Section , the circuit had two essential connections: an input (the Arduino pin and CH1) and an output (CH2). The circuit in Figure 3, however, uses only one connection, but the math is essentially the same as the previous circuit. In this circuit, the Arduino pin will both charge the capacitor *and* measure its decay. To charge the capacitor, you need to set the pin to be an output and turn it HIGH. But you can't just turn it LOW to time the decay – it would immediately source all the current and drain the capacitor. Instead, you must switch it to be an INPUT pin, which will allow you to monitor the voltage without sinking any current. The 100Ω resistor is simply acting as a current limiter during the charging phase – during the decay, the Arduino pin is set as an input and already has a very large resistance, so the small extra resistance is irrelevant.

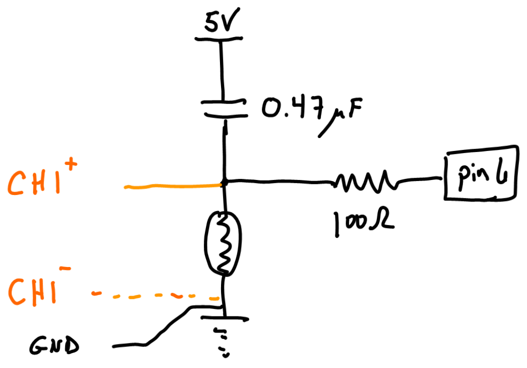

Figure 3: An RC-timer with a photoresistor as the resistive element.

Procedure

1. Leaving the photoresistor in place, remove all of the other components from Circuit A in Figure 1.
2. Replace it with the circuit in Figure 3.
3. Add a *function* that reads the decay time of the RC circuit, as follows:
   (a) Set pin 7 as an OUTPUT.
   (b) Set the pin to HIGH.
   (c) Wait 2ms for the capacitor to charge. Use `delay()`.
   (d) Set the pin as an INPUT.
   (e) Record the start time using `micros()`.
   (f) Wait for the pin to go LOW.
   (g) Record the end time using `micros()`.
   (h) Return the difference between the two times – i.e., the decay time.
4. Use CH1 on the oscilloscope to measure the voltage across the photoresistor, as shown.
5. Starting with the LED toggling program, edit it so that it calls the new function in the program's `loop()` and writes the decay time to the Serial Monitor every 1 s. For now, just continue to use a `delay()` function to slow the loop down.
6. Run the program and toggle the light on and off. Note the decay time and compare it to the fall time read by the oscilloscope. Record both values in Table 1. Why aren't they the same? Do they show the same trends?

Demonstrate your working system to an instructor.

Condition	Fall time (measured from oscilloscope)	Decay time (measured by the Arduino)
Light		
Dark		

Table 1: RC timer dynamics

Precision timing with a comparator

Though easy to implement, the RC-timer above suffers from one potential pitfall, namely that you are not guaranteed that the pin on the ATmega328 will switch from `HIGH` to `LOW` at a consistent voltage. That is, the datasheet guarantees that any voltage below 1 V will register as `LOW` and anything above 3V as `HIGH`. In between, the pin state is undefined, and while it may switch at a consistent value for this lab, there's no guarantee that it will continue to do so, especially when more components are added to the system. For this system, it's hardly critical, but in others, it is necessary to define a precise threshold for the RC-timer.

A straightforward solution to the dilemma is to use a *comparator* to set a precise threshold where the pin will change, as shown in Figure 4. A comparator does just as its name suggests: it compares two voltages and outputs a `HIGH` or `LOW` depending on which is higher: if the input on the *non-inverting* input (labelled with a '`+`') is higher than the *inverting* input (with a '`-`'), it outputs a `HIGH`; otherwise it outputs a `LOW`. Thus, one can establish the threshold voltage for the RC decay using a voltage divider and use the comparator to precisely time the decay. The added precision comes at a cost, however. Not only do you need an extra component, but you must also use two pins on the Arduino since you can't send current "backwards" through the comparator, you'd need one pin to read the compartor and a second pin to charge the capacitor.

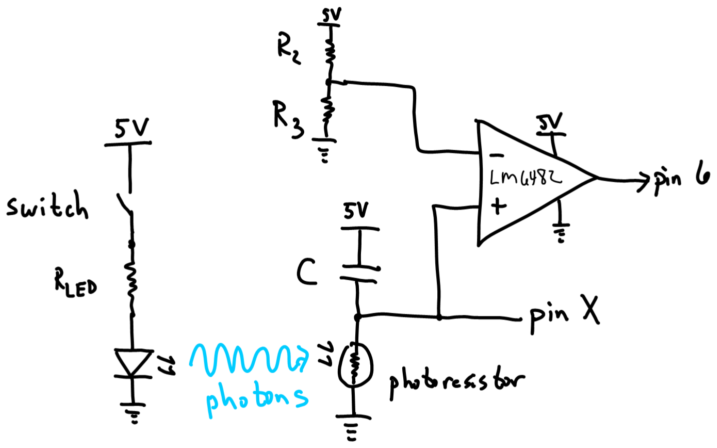

Figure 4: Circuit for precision timing using an external comparator. *Do not build this circuit* – there is an easier way!

The circuit can be simplified greatly by using the built-in comparator on the ATmega328. By default, pins 6 and 7 on the Arduino (`PIND6` and `PIND7` on the ATmega328) are the non-inverting (`AIN0`) and inverting (`AIN1`) terminals, respectively, of an internal comparator. Unlike a dedicated op-amp, the comparator does not have an output pin, but instead controls an internal value that can be read with a low-level commands:

```
int comparatorOutput = (ACSR >> ACO) & 0x01;
```

A little arcane, but `comparatorOutput` in the line above will be `1` when the voltage on pin 6 is greater than pin 7 and `0` otherwise. The nice part is that you can freely command the pins to be inputs or outputs without worrying about the comparator – it runs in the background regardless.

1. Build the circuit in Figure 5. The resistors in the voltage divider on pin 7, the inverting terminal of the comparator, should be sized to create as close to 1.84V as reasonable. (Why did I choose 1.84V?) A good rule-of-thumb is to make the series resistance somewhere between $10k\Omega - 50k\Omega$.

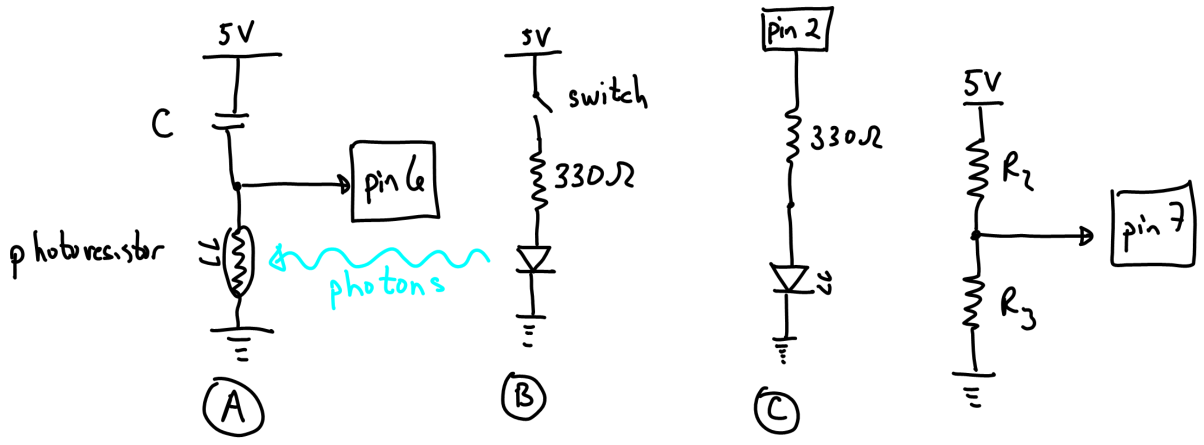

Figure 5: Circuit for precision timing using the ATmega328's internal comparator. It is much easier than the external comparator above.

2. Edit your code from the previous section to read the output of the internal comparator, instead of the pin directly.
3. Run your code again and use your oscilloscope to confirm that the decay time is the same as before.

Demonstrate your working system to an instructor.

Pre-lab worksheet

To be turned in at the start of lab. Completed individually, but you are welcome to talk to others about the material.

Answers with incomplete or incorrect units will be marked as wrong!

1. Consider the circuit in Figure 6. Calculate the rise-time, defined in this instance as the amount of time to go from 10% $\rightarrow$ 90% of the final value, for a step input (i.e., the input voltage jumps from 0V to 5V at some time, t_0).

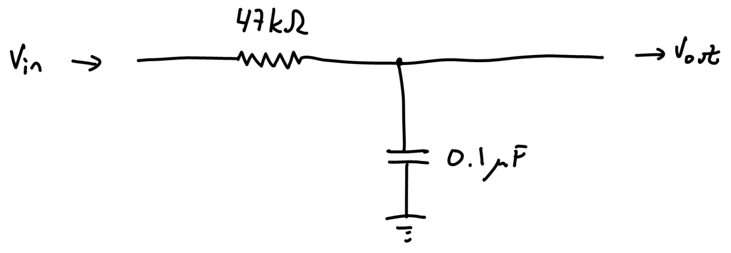

Figure 6: An RC circuit.

2. Using a circuit with the same topology, find a resistance, R, that gives a rise-time of 1ms if $C = 22\mu F$, a common capacitor size. What is the closest *common* resistor you can find?

3. In the oscilloscope image in Figure 7, what trigger settings are being used (source, condition, and level)?

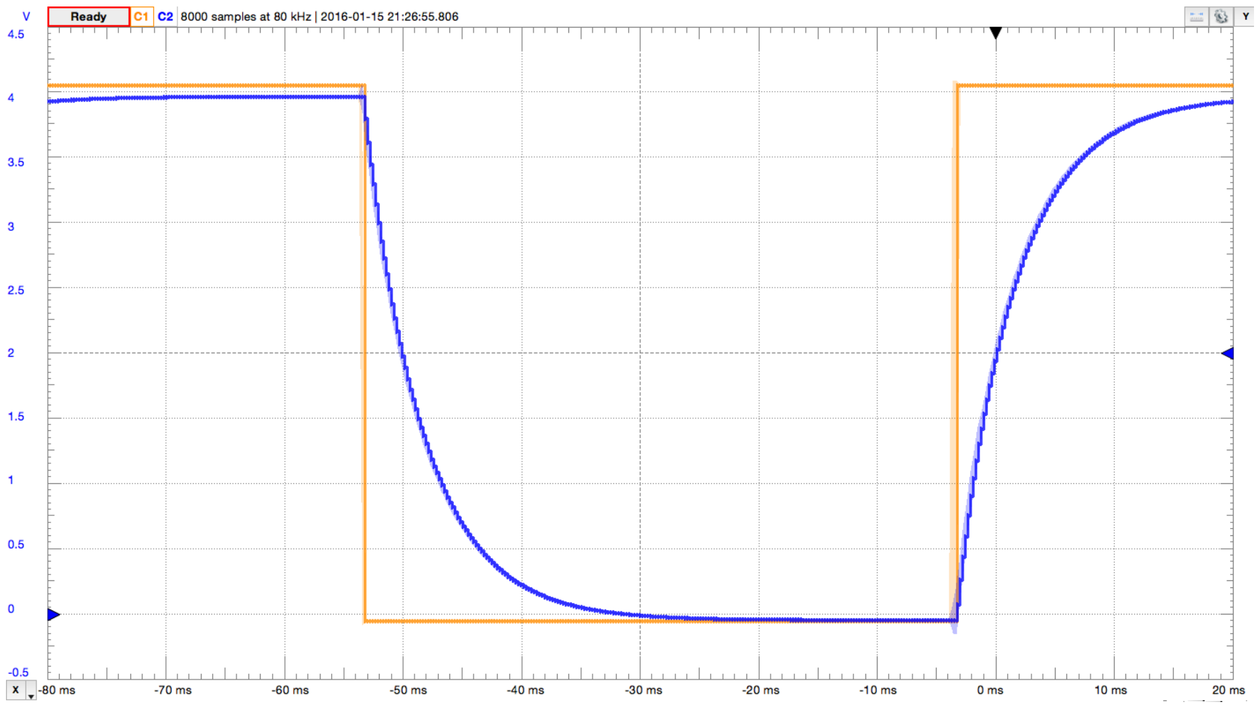

Figure 7: A oscilloscope capture from an RC circuit.