Alyssa J. Pasquale, Ph.D.
College of DuPage

Microcontrollers Lab Manual

Spring 2025 Edition

this lab manual belongs to

Table of Contents

A note on content

Topics in this lab manual will be discussed with a focus on practical application. It will be assumed that you have read the relevant textbook sections for a detailed discussion of each individual topic before reading the lab content. The textbook sections related to each lab are listed on the website page for each lab. The textbook is available online at `https://doctor-pasquale.com/wp-content/uploads/2021/02/The-Yellow-Book.pdf`.

A note on flowcharts

Flowcharts have been provided in this lab manual to help you to visualize the flow of software code. Please keep the following notes in mind as you use the flowcharts to help you solve each lab circuit.

- They may not include every step.
- They may not include variable types or attributes.
- They may not include all subroutines or interrupt service requests.
- They may not be the best way to solve the circuit.

License and attribution information

Pre-Lab 1

Carefully read the entirety of Lab 1, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. What are two advantages of using an Arduino Uno rather than just using an ATmega328P directly?

2. What are two advantages of using C code rather than Arduino IDE functions?

3. In what port can each of the following pins be found, and in what bit are they located?
   (a) D12

   (b) A0

   (c) D9

   (d) D5

4. What precaution must be taken with pins D0 and D1, and why is this so?

5. Why are bitwise operators used when manipulating I/O pins?

6. Specifically, which registers require the use of bitwise operators?

7. What are the results of the following bitwise operations? (Show all of your work.)
   (a) `0b10110101 & 0b00111100`

   (b) `0b10110101 | 0b00111100`

   (c) `0b10110101 ^ 0b00111100`

8. Find the final value of each variable after the compound operation has been executed.

```
unsigned char a = 102;
a /= 6;
```

```
unsigned char n = 15;
n++;
```

```
unsigned char j = 50;
j %= 7;
```

```
unsigned char x = 255;
x ^= 0x0F;
```

```
unsigned char num = 22;
num &= 3;
```

Lab 1: Introduction to the Arduino Uno and AVR ATmega328P

In this introductory lab, the basic operations of the Arduino Uno will be explored by designing several LED circuits. Each circuit will build off of provided C code to create circuits of increasing complexity. In the process, C functions, digital output pins on the Arduino, and bitwise operators will be explored. The importance of writing well-documented code will be emphasized. **C Concepts:** `_delay_us()` and `_delay_ms()` functions, `char` datatype, bitwise operators (bitwise AND, bitwise OR, bitwise XOR), compound operators **AVR Concepts:** I/O port registers (`DDRx`, `PORTx`, `PINx`)

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

1.1 Arduino Uno

The Arduino Uno is a microcontroller breakout board with 20 bidirectional input/output pins. At the heart of the Arduino Uno is the ATmega328P microcontroller. This microcontroller is manufactured by Microchip (formerly Atmel), and is part of the AVR family of microcontrollers. Devices that are part of the AVR family share a common machine language and can be configured and programmed in a very similar manner.

The Arduino Uno board can be powered with an external power supply, batteries, or by a computer via the USB port. When plugged into the computer, software programs can be downloaded onto the Arduino board using the Arduino IDE (integrated development environment, which is the compiler used in this class to write and upload code).

To explain the difference between the Arduino and the ATmega328P, Figure 1.1 shows a diagram showing that the Arduino Uno is a package that provides easy access to the microcontroller pins. It includes a power jack and power regulator to protect the microcontroller and all other components. A reset button connects directly to the reset pin on the ATmega328P to trigger a system reset when pressed. The USB interface allows seamless communication between your computer and the device, and the bootloader on the ATmega328P allows the microcontroller to understand the instructions it receives when code is uploaded to the board. The debug LED and ICSP headers are other useful features that the Arduino package provides.

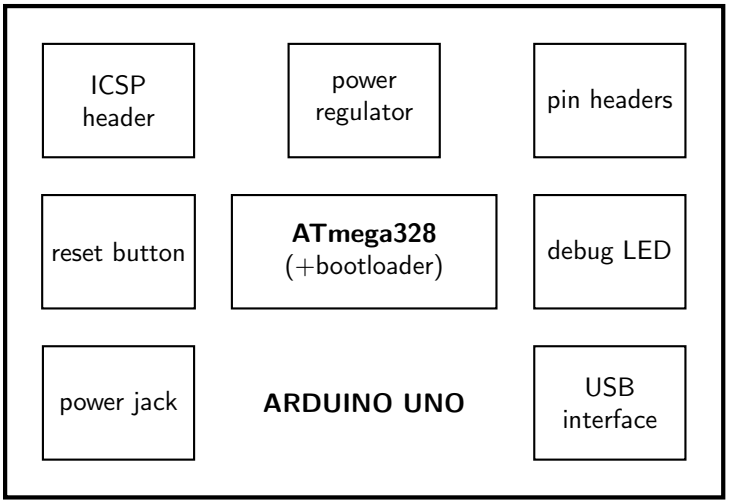


Figure 1.1: Arduino Uno breakout board features.

Why bother using an Arduino Uno, which can cost about $30, when an ATmega328P microcontroller can be purchased for less than $5? There are some benefits to using the Arduino platform, as well as drawbacks. The benefits of using the Arduino include the power regulator that can help prevent overvoltages applied to the chip. (The power regulator can only protect up to 20 V, however.) The USB interface allows for seamless communication between the Arduino IDE software and the microcontroller. Pin headers that are soldered onto the platform make it straightforward to connect to each of the pins on the microcontroller without needing to refer to a pinout diagram. Drawbacks of using the Arduino Uno will be explored in greater detail in Lab 10, but notably include a dramatic increase in power consumption.

Because the ATmega328P microcontroller on the Arduino will be programmed using the C language, it is necessary to understand the architecture of the device in order to properly configure it. There are "simple" functions provided by the Arduino IDE, but these come with a massive increase in processing time and memory consumption, they are very inflexible, and they do not facilitate a thorough understanding of the microcontroller. Use of C programming also allows the code written for the ATmega328P to be more portable, which is important if the code is ever to be used on a different microcontroller in the future.

1.1.1 Uploading code to the Arduino Uno

Code can be uploaded to the Arduino Uno using a USB cable and the Arduino IDE software. The following steps should be followed. (These steps are also available online at `https://github.com/DoctorPCOD/DoctorPCOD/tree/main/ENGIN-2223`.)

1. Write the software code using the Arduino IDE.
2. Use CTRL-T to get the Arduino IDE to clean up the code formatting.
3. Connect the Arduino to the computer using the USB cable.
4. Select the correct board type in the Arduino IDE by selecting Tools > Board > Arduino Uno.
5. Select the correct port in the Arduino IDE by selecting Tools > Port and selecting the port that contains the label Arduino Uno.
6. (Optional) Verify the code is free from errors by using CTRL-R.
7. Before uploading code, ensure that nothing is connected to pins `D0` and `D1`.
8. Upload code to the Arduino Uno by using CTRL-U.

1.2 ATmega328P I/O ports

In order to control the pins of the Arduino, the I/O ports on the microcontroller must be understood. The digital pins on the Arduino Uno are numbered from D0–D13. There are in addition digital pins with optional analog input functionality that are numbered A0–A5. Each of these pins resides in a different port on the ATmega328P. Each port contains 8 bits, and is therefore associated with at most 8 pins. The three ports are Port B, Port C and Port D. The pins in each port are described in Table 1.1. Bits labeled – are reserved.

bit:	7	6	5	4	3	2	1	0
Port B:	–	–	D13	D12	D11	D10	D9	D8
Port C:	–	–	A5	A4	A3	A2	A1	A0
Port D:	D7	D6	D5	D4	D3	D2	D1	D0

Table 1.1: Pins in ports B, C, and D.

1.2.1 I/O pin data direction

Each pin has the capability to be either an output pin (data is written to the pin from the microcontroller and sent from the pin to control devices such as LEDs, motors, or other output devices) or an input pin (data is received by the pin from devices such as pushbuttons or sensors which can be read by the microcontroller to determine the state of the input device). In order to control this direction, there is a register on each port known as the Data Direction Register. To change the data direction on Port b requires use of the `DDRB` register. To change direction on Port c requires use of the `DDRC` register. Port d directionality is controlled by `DDRD`. Setting (writing a value of 1) a bit causes it to be an output. Clearing (writing a value of 0) a bit causes it to be an input. By default, all pins are cleared (i.e. are input pins) at the start of any program.

1.2.2 Setting and clearing pin output values

For each output pin, it is important to be able to selectively set or clear the value of the pin. For example, to turn on an LED, connected via a current-limiting resistor to ground, the output pin connected to the LED will be configured to give it a logic HIGH value. To turn off the LED, the output pin will be configured to give it a logic LOW level. Changing the state of the pins is controlled using the Port Data Register. These are named `PORTB`, `PORTC` and `PORTD`.

1.2.3 Reading pin data values

To read data from an input pin, the value stored on that pin must be read and stored in a variable. The registers that store data from pins are known as the Port Data Input Registers, and they are named `PINB`, `PINC` and `PIND`. These registers will be used in Lab 2.

1.2.4 Pins D0 and D1

There is a precaution that must be taken any time pins D0 or D1 are utilized on the Arduino. The Arduino uses these pins for serial communication via the USB connection to the computer. Therefore, any time code is uploaded from the Arduino IDE to your board, these pins will need to be completely empty, with nothing plugged in to them. If there are wires or other devices connected to pins D0 or D1 when an upload occurs, there will be problems with the file transfer.

1.3 Program flow in Arduino IDE

All programs in the Arduino IDE are structured by using two major functions, `setup()` and `loop()`. The first of these functions, `setup()`, runs once when the program starts. It is used to do tasks that only need to occur once, such as configuring I/O pins and other peripheral functions.

The second function, `loop()` will run over and over again until the Arduino is either reset or the power is turned off. This can be depicted as a flowchart in Figure 1.2.

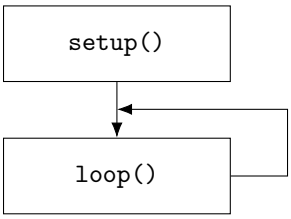


Figure 1.2: Flowchart of the two major functions used in the Arduino IDE.

1.4 Datatype: `char`

There are different datatypes used in C. For the purposes of this lab, 8-bit numbers will be sufficient for all circuit variables. The datatype for this purpose is known as `char`. The `char` datatype represents a signed

8-bit number, which means it is capable of representing values between −128 and 127. If unsigned numbers are needed, the datatype `unsigned char` can be used, which represents values between 0 and 255. Refer to Appendix E for information about datatypes and examples of their usage.

To create a variable that can be used in C code, it can declared using any one of three different number systems: decimal, binary, and hexadecimal. No prefix is required when assigning a decimal value to a variable or a register. However, a prefix of `0b` is required for assigning a binary value to a variable or register, and a prefix of `0x` is required for hexadecimal values. This is demonstrated below.

```
char a = -62;           // a decimal value requires no prefix
char a = 0b11000010;    // a binary value requires a prefix of 0b
char a = 0xC2;          // a hexadecimal value requires a prefix of 0x
```

1.5 Bitwise operators

Bitwise operators are convenient to use when manipulating binary variables. In fact, they are necessary to use to manipulate I/O pin data using `PORTx` registers. When multiple pins are in use (either as input or output pins), it is vital to ensure that changing the state of one pin doesn't interfere with any other pin in that register. All bitwise operators are defined in Appendix E.

1.5.1 Bitwise AND: `&`

The single ampersand symbol `&` represents the bitwise AND function. In a bitwise function, each bit is compared individually with the other corresponding bit and the logical AND is taken between them to find the final value. The bitwise AND function is used when one or more specific bits need to be cleared with all other bits kept at their previous value. Following are the Boolean algebra identities for operations with 0 and 1 to analyze how the bitwise AND works.

```
X & 0 = 0
X & 1 = X
```

To clear a bit (`X`), perform the function `X & 0`, as any number AND 0 is 0. To leave `X` unchanged, perform the function `X & 1`, as any number AND 1 is itself.

1.5.2 Bitwise OR: |

The single pipe symbol | (shift-backslash on a keyboard) represents the bitwise OR function. In a bitwise function, each bit is compared individually with the other corresponding bit and the logical OR is taken between them to find the final value. The bitwise OR function is used when one or more specific bits need to be set, with all other bits kept at their previous value. Following are the Boolean algebra identities for operations with 0 and 1 to analyze how the bitwise OR works.

```
X | 0 = X
X | 1 = 1
```

To set a bit (`X`), perform the function `X | 1`, as any number OR 1 is 1. To leave `X` unchanged, perform the function `X | 0`, as any number OR 0 is itself.

1.5.3 Bitwise XOR: ^

The single caret symbol ^ (shift-6 on a keyboard) represents the bitwise XOR function. In a bitwise function, each bit is compared individually with the other corresponding bit and the logical XOR is taken between them to find the final value. The bitwise XOR function is used when one or more specific bits need to be toggled, with all other bits kept at their previous value. Following are the Boolean algebra identities for operations with 0 and 1 to analyze how the bitwise XOR works.

```
X ^ 0 = X
X ^ 1 = X'
```

To toggle a bit (`X`), perform the function `X ^ 1`, as any number XOR 1 will cause its value to toggle. To leave `X` unchanged, perform the function `X ^ 0`, as any number XOR 0 is itself.

1.6 Compound operators

Compound operators (also known as combined assignment operators) are frequently used when a variable will be modified and then stored back into the same variable. For example, suppose you wish to bitwise AND an 8-bit unsigned variable declared as `a` with the value `0x0F`, and then store the result back to `a`. The first option is to do this without a compound operator, as shown below.

```
1 a = a & 0x0F;
```

The second option is to use a compound operator to perform the same task.

```
1 a &= 0x0F;
```

Compound operators are also frequently used to increment (`a++` will increment the variable `a`) or decrement (`a--` will decrement the variable `a`). None of them are truly necessary to use, but make code much more readable. It is strongly recommended that you get into the habit of using compound operators in your code. All of the compound operators are listed in Appendix F.

1.7 AVR delay functions

The AVR family of microcontrollers have two functions for generating delays. They are called `_delay_us()` and `_delay_ms()`. These functions are useful when a certain amount of time needs to be spent waiting between actions. The function `_delay_us()` creates a delay in microseconds equal to the number passed through the argument of the function. The longest delay possible using this command is 4,294,967.295 μs (4.29 s). The function `_delay_ms()` creates a delay in miliseconds equal to the number passed through the argument of the function. The longest delay possible using this command is 4,294,967.295 ms (about 72 min).

Circuit I: Blinking an LED

This circuit will blink an LED on and off at regular intervals.

Download the file `lab1_circuit1.ino`. Connect an LED and current-limiting resistor as shown in Figure 1.3.

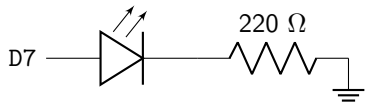
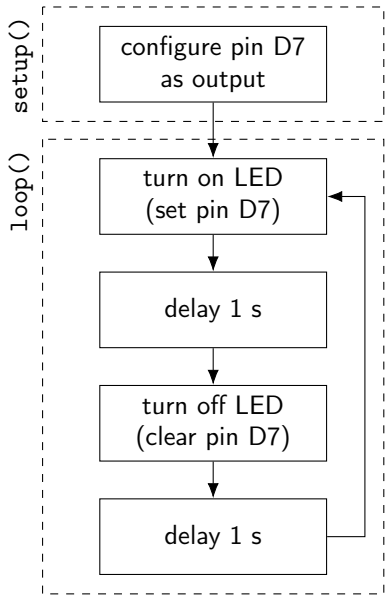


Figure 1.3: Schematic diagram for circuit I.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.



Program Memory: _____ bytes

Data Memory: _____ bytes

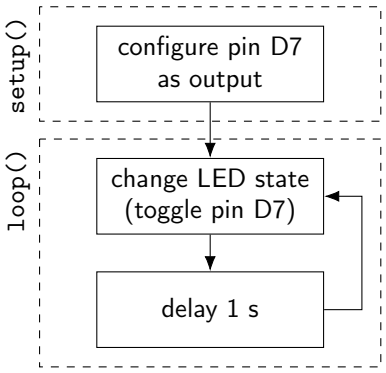
Instructor Stamp: _____

Circuit II: Toggling an LED

This circuit will blink an LED on and off at regular intervals.

Change the code from Circuit I to blink the LED by toggling pin D7 using the bitwise XOR function instead of using bitwise AND and OR to clear and set the pin.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.



Program Memory: _____ bytes

Data Memory: _____ bytes

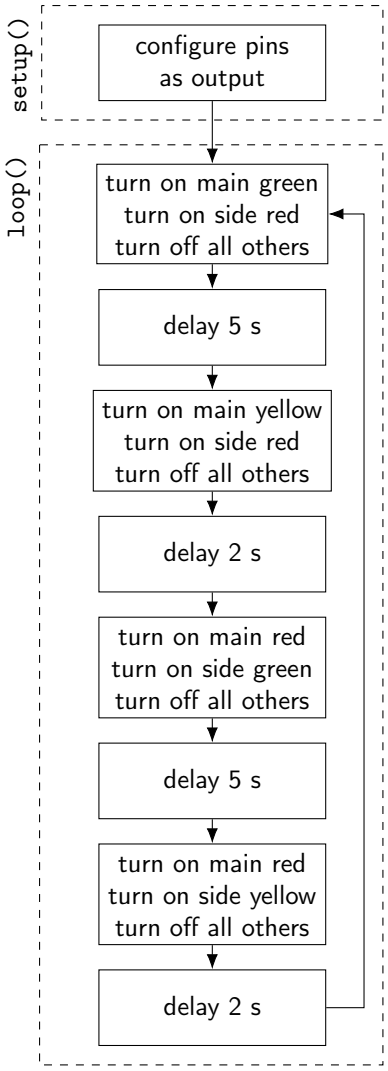
Instructor Stamp: _____

Circuit III: Traffic light circuit

> This circuit will emulate a traffic light at an intersection between a main street and a side street. The green light on each street will be on for 5 seconds, and the yellow light will be on for 2 seconds. When one traffic light is either green or yellow, the other traffic light must be red.

To demonstrate the efficiency of software over hardware, we will revisit the traffic light finite state machine circuit from Digital Systems Lab 13. Amend the software from the previous circuit(s) to light LEDs in accordance with a traffic intersection.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 1 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Functionally, what was the difference between Circuit I and Circuit II? Would you be able to tell the difference between them just by looking at the finished circuit?
- What are the advantage(s) of using pin toggling in Circuit II over the method used in Circuit I?
- Explain the benefits of using a microcontroller over using a purely hardware approach (as you did in Digital Systems) to system design.

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab?

Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 2

Carefully read the entirety of Lab 2, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. To read the state of a digital input pin you use the following code to isolate bit 6. Find and explain the error.

```
1 PIND &= 0x40;
2 PIND >> 6;
```

2. When, if ever, do pins A0–A5 require setup using the `DDRC` register? Justify your answer.

3. When using full-precision mode on the ADC...
   (a) ...what is the most efficient `ADLAR` value to use?

   (b) ...what register is used to read the ADC value?

   (c) ...what kind of datatype should be used to store the ADC result? Justify your answer.

4. When using 8-bit mode on the ADC...
   (a) ...what is the most efficient `ADLAR` value to use?

   (b) ...what register is used to read the ADC value?

   (c) ...what kind of datatype should be used to store the ADC result? Justify your answer.

5. What is the difference between a pull-up resistor and a pull-down resistor? When are these types of resistors used?

6. Explain the difference between the following two pieces of code: `if (x=5) { // do something }`, and `if (x==5) { // do something }`

7. Explain the difference between the following two pieces of code: `unsigned char f = a & b`, and `unsigned char f = a && b`

8. Find the final value of `x` after each loop has been executed.

```
int x = 0;
for (unsigned char j = 0; j > 6; j++) {
  x += 50;
}
```

```
int j = 0;
for (int x = 0; x < 10; x++) {
  j--;
}
```

Lab 2: Digital and Analog Input Devices

Input devices (both digital and analog), and their ability to control the operations of a system, will be explored. Many values (for example, from sensors) tend to take on analog values, which can be accommodated with the analog to digital converter (ADC) on a microcontroller. In this lab, input devices in general will be used, and the principles of operation of the ADC will also be explored. **C Concepts:** bitshift operators (bitshift left, bitshift right), comparison operators, Boolean operators, `if`/`else` statement, `for` loop, `int` datatype **AVR Concepts:** I/O register `PINx`, ADC registers `ADCSRA`, `ADCSRB`, `ADMUX`, `ADC`, `ADCH`, `ADCL`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

2.1 Input devices

In this lab, input devices will be used. Input devices allow the circuit designer to obtain data from the environment external to the microcontroller. Input devices can be lumped into two categories: digital input devices and analog input devices. Table 2.1 contains some examples of each type of input device.

Digital	**Analog**
Pushbutton	Potentiometer
Toggle Switch	Microphone
Keypad	Thermistor

Table 2.1: Various types of digital and analog input devices.

Some sensors or input devices communicate digitally using serial communication protocols. In fact, Arduinos themselves can be connected together and communicate serially, one as an input, one as an output. Serial devices will be utilized in future labs.

2.2 Digital inputs

The value of a digital input can be found by reading the respective pin values from the `PINx` register. This raises the question of how a single bit from a register can be isolated in order to read it individually. The steps for reading bits from a register follow.

- Mask the bit by bitwise ANDing with a constant that has all bits set to 0 except for the bit of interest.
- Save this result as a new variable and don't overwrite the `PINx` register!
- Bitshift right n times, where n is the bit number in the register.

The first step is known as masking. It is vitally important to mask data coming in from `PINx` registers because of the following two reasons.

1. It is possible that several pins are in use, but only one is of interest.
2. Unused input pins are subject to noise, and will occasionally read HIGH values.

Once data has been masked, it must be shifted using the bitshift right (>>) operator. Unless the pin of interest is in the least significant bit in the `PINx` register, the value of any variable assigned to that `PINx` register will be a power of two, but not necessarily one, when the pin is HIGH.

For example, to read only pin D11, while ignoring all of the other pins in Port B, the following code should be executed.

```
unsigned char a = PINB & 0x08;  // mask all other pins
a >>= 3;  // shift the bit of interest to the LSB
```

2.3 External pull-up and pull-down resistors

When closing a pushbutton switch (for example), a direct connection between power and ground is created. Without the presence of any resistance, this will lead to a short circuit and could damage components. It is important to use pull-up or pull-down resistors in series with any push buttons or DIP switches that are used as input devices. Throughout this course, 10 kΩ resistors will be used as pull-up or pull-downs. (Note that pull-up or pull-down resistors do not need to be used with toggle switches that have three pins and two passive positions.)

By using a pull-up resistor, a switch is set as active LOW (i.e. the switch is 1 when not pressed, and 0 when pressed), and by using a pull-down resistor, a switch is set as active HIGH (i.e. the switch is 0 when not pressed, and 1 when pressed). Schematics of pushbuttons and DIP switches wired with external pull-up or pull-down resistors are included in Appendix B of this lab manual.

2.4 Internal pull-up resistors

The ATmega328P has an internal pull-up resistor that can be individually activated on each of the I/O pins. When a pin has been configured as an input pin, writing a 1 to the corresponding bit in the correct `PORTx` register will enable the internal pull-up.

The circuit diagram shown in Figure 2.1 depicts a pushbutton connected to an input pin on the ATmega328P used with the internal pull-up. The solid lines on the schematic indicate the hardware that is external to the microcontroller, and the dashed lines depict components internal to the microcontroller.

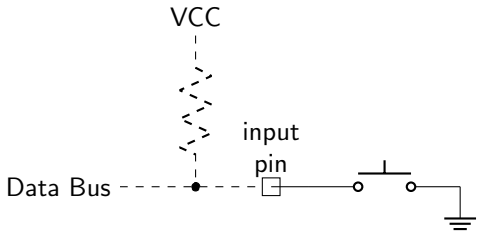


Figure 2.1: Schematic diagram of a pushbutton used with the internal pull-up resistor.

It is interesting to note that the ATmega328P does not have actual, physical resistors internally, so it is not possible to give an exact value of the resistance of the pull-ups. Instead of a resistor being included in the hardware, the resistance of the internal wiring on a transistor is exploited and used to save space. Therefore, the exact value of the internal pull-up will depend on external parameters such as voltage and temperature, but is nominally between 30 kΩ and 40 kΩ.

Using the internal pull-ups can save a lot of hardware, especially if multiple input devices are used. However, it is important to note that the pull-up architecture leads to active LOW signals on a pushbutton. This will need to be compensated for in software.

2.5 Analog inputs

In the digital world, 0 V corresponds to 0, and 5 V corresponds to 1. In the analog world, a continuum of values from 0–5 V are possible. These values do not directly correspond to digital (binary) values. They therefore need to be converted to a binary number using an analog to digital converter (ADC). The ATmega328P has a single 10-bit ADC. The input signal to the ADC can be selected from one of six I/O pins,

all of which are found on Port C (Arduino pins A0–A5). Some packages of the ATmega328P microcontroller (TQFP and QFN/MLF) contain eight I/O pins that can be used as an input source to the 10-bit ADC. The pin that is selected as the source for the ADC is selected in an ADC configuration register, described below.

2.5.1 Potentiometers as analog voltage sources

In this lab, a potentiometer will be used as an analog input device. A potentiometer, or "pot" for short, is a resistor that has three pins. The circuit symbol for a potentiometer is shown in Figure 2.2.

Figure 2.2: Circuit symbol for a potentiometer.

Two fixed leads are connected to a resistive element arranged in a semi-circular configuration. A moveable lead called the wiper can then be positioned at any point along the resistive element by turning the dial on the potentiometer. This is depicted in Figure 2.3.

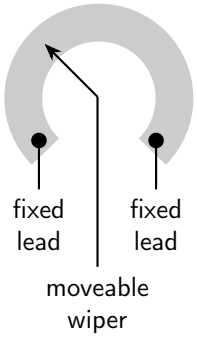


Figure 2.3: A schematic of a potentiometer. The gray semicircular area corresponds to the resistive element. Two leads (one at each end of the resistive element) are fixed. The third lead is a moveable wiper.

While the total resistance between the fixed leads is constant (R_{POT}), the resistance between the wiper and either of the two fixed leads (R_1 and R_2) changes as the wiper's position changes. The sum of R_1 and R_2 is equal to the total resistance of the potentiometer, defined by Equation 2.1.

$$R_1 + R_2 = R_{POT} \tag{2.1}$$

In this manner, a potentiometer can be used as a voltage divider. If the two fixed leads are connected to VCC and GND, then the center pin will act as an analog voltage, as shown in Figure 2.4. As the dial on the pot is turned from one extreme to the other (moving the position of the wiper), the values of R_1 and R_2 change in turn so that the voltage drop V_1 increases from 0 V to 5 V, while the voltage drop V_2 decreases from 5 V to 0 V. The relationship $V_1 + V_2 = 5$ V will always be satisfied.

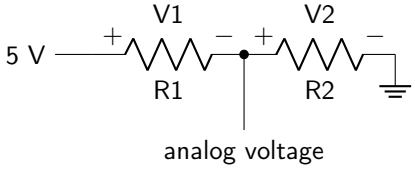


Figure 2.4: Schematic diagram of a potentiometer viewed as a voltage divider.

2.6 Datatypes: `int`

Because the ADC has 10 bits of resolution, a `char` datatype is no longer sufficient to store this data. `int` is a 16-bit signed number capable of taking on values between −32,768 and 32,767. If only unsigned numbers will be used, the related datatype `unsigned int` can be used. It is capable of representing values between 0 and 65,535. Refer to Appendix E for examples.

2.7 ATmega328P peripheral configuration registers

Peripheral features of the microcontroller (such as the analog to digital converter, timer/counters, and external interrupts) are highly configureable. Each peripheral therefore has one or more registers associated with them. Each bit in the register controls a configuration aspect of that peripheral feature. For example: one bit may be dedicated to enabling or disabling the feature, one bit may be dedicated to turning on or off interrupts associated with the feature, one or a collection of bits may be dedicated to controlling the exact mode of operation of the feature, and so on.

Some of the bits in a configuration register may be reserved (typically depicted by a – symbol in a bit location). According to the ATmega328P datasheet: "[f]or compatibility with future devices, reserved bits should be written to zero if accessed." Keep this in mind when assigning values to peripheral configuration registers.

2.8 ATmega328P analog to digital conversion

Because analog inputs require the use of the analog to digital converter (ADC), they are more complicated to work with than digital inputs. Several registers need to be initialized and properly set up before the conversion can take place, with many options available depending on how the ADC is to be used. Information about each of these registers can be accessed in Appendix A.

- `ADCSRA` – **ADC Control and Status Register A**: This register stores information about how the ADC is to be used. It is used in conjunction with `ADCSRB`.
- `ADCSRB` – **ADC Control and Status Register B**: A single register is not large enough to contain all information about how to run the ADC, therefore this register stores additional information about how the ADC is to be used.
- `ADMUX` – **ADC Multiplexer Selection Register**: This register contains information about the reference voltage to be used by the ADC, which pin is to be used for data input, and includes `ADLAR`, the data alignment configuration bit.

2.8.1 ADC data alignment and data registers

Because the analog to digital converter on the ATmega328P has 10 bits of precision, and most all registers are 8 bits wide, the data is therefore saved between two registers. Because the ADC precision is not a power of two ($10 = 2^{3.322\cdots}$), the data can be stored in different ways in the two ADC data registers, which are called `ADCH` (ADC High Byte Data Register) and `ADCL` (ADC Low Byte Data Register). The way that the data is stored within these two registers depends on the alignment, which is configured using the `ADLAR` bit in the `ADMUX` register. How to configure `ADLAR` depends on the desired precision level of the ADC. For full details of how the data is stored between these two registers, refer to Appendix A.

In **full-precision mode**, all 10 bits of the ADC result will be used. This is most easily accomplished when `ADLAR` has a value of zero. When `ADLAR` is zero, the 10-bit value of the ADC result can be accessed by reading the variable `ADC` (which will give the correctly formatted 10 bit conversion result).

```
1 unsigned int adcResult = ADC;   // int required for 10 bits of precision
```

Sometimes, 10 bits of precision are not needed. **8-bit precision mode** can be used instead. In this case, only the most significant byte of data is stored, ignoring the least significant two bits. When `ADLAR` has a

value of one, the most significant byte of data is saved in ADCH. Therefore, when 8-bit precision mode is used, the result can be read directly from ADCH. This is accomplished using the following code.

```
unsigned char adcResult = ADCH; // char used for 8 bits of precision
```

2.9 Comparison and Boolean operators

Comparison operators are used to compare two variables or values. This is useful when decisions need to be made based on how one variable compares to another. Comparison operators are listed in Appendix F.

Boolean operators are used to compare the results of two or more comparison operations. The Boolean operations consist of AND, OR, and NOT. Boolean operators are listed in Appendix F. It is important to note that Boolean operators use TWO symbols (with the exception of NOT), whereas bitwise operators only use one. Do not confuse the two types of operators, as they are radically different!

2.10 Control flow: conditional

A microcontroller is capable of executing specific segments of code if a certain condition is met. This is known as conditional flow. The conditional flow functions used in C are `if` statements and `if`/`else` statements.

In a flowchart, conditional statements are depicted with a diamond-shaped block. This is depicted in Figure 2.5.

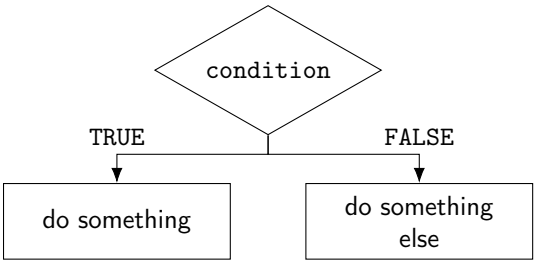


Figure 2.5: Conditional statements in a flowchart are depicted by a diamond.

2.11 Control flow: iterative

A microcontroller is capable of executing specific segments of code a certain number of times (or infinitely). This is known as iterative flow. The conditional flow functions used in C are `for` loops, `while` loops, and `do`/`while` loops.

2.11.1 `for` loop

A `for` loop is a type of iterative control flow. It is used when a piece of code will be repeated a well-defined number of times. This is slightly different from `while` and `do`/`while` loops, the purpose of which is to wait for a certain condition to be satisfied. (`while` and `do`/`while` loops will be covered in Lab 9.) In a `for` loop, a variable is initialized and checked against a condition. When that condition is satisfied, the code inside the loop executes. After the code has completed, an afterthought is executed. Then the condition is re-checked. This process continues until the condition is not satisfied, at which point the code leaves the `for` loop.

```
for (initilization;condition;afterthought) {
  // this code will execute if the condition is satisfied
  // after the code is complete, the afterthought will execute
  // then, the condition will be checked again
}
```

To write a `for` loop that executes continuously (an infinite `for` loop), the following code can be used.

```
// infinite for loop
for ( ; ; ) {
  // this will be repeated infinitely
}
```

Circuit Ia: Pushbutton input with external pull-down resistor

When a pushbutton is pressed, an LED will light up. Otherwise, the LED will remain off.

Download the file `lab2_circuit1.ino`. Connect a pushbutton with a pull-down resistor to pin D8 and hook up an LED and current limiting resistor to pin D13, as shown in Figure 2.6.

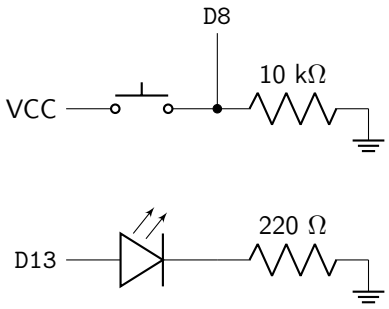
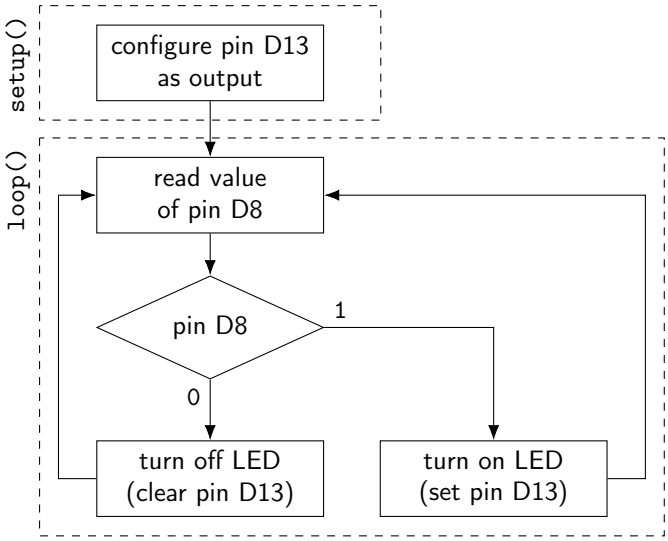


Figure 2.6: Schematic diagram for Circuit Ia.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit Ib: Pushbutton input with internal pull-up resistor

When a pushbutton is pressed, an LED will light up. Otherwise, the LED will remain off.

Accomplish the same functionality as in Circuit Ia, but instead of using an external pull-down resistor, activate the internal pull-up resistor. (This means that there will be no external 10 kΩ resistor connected to the pushbutton.)

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

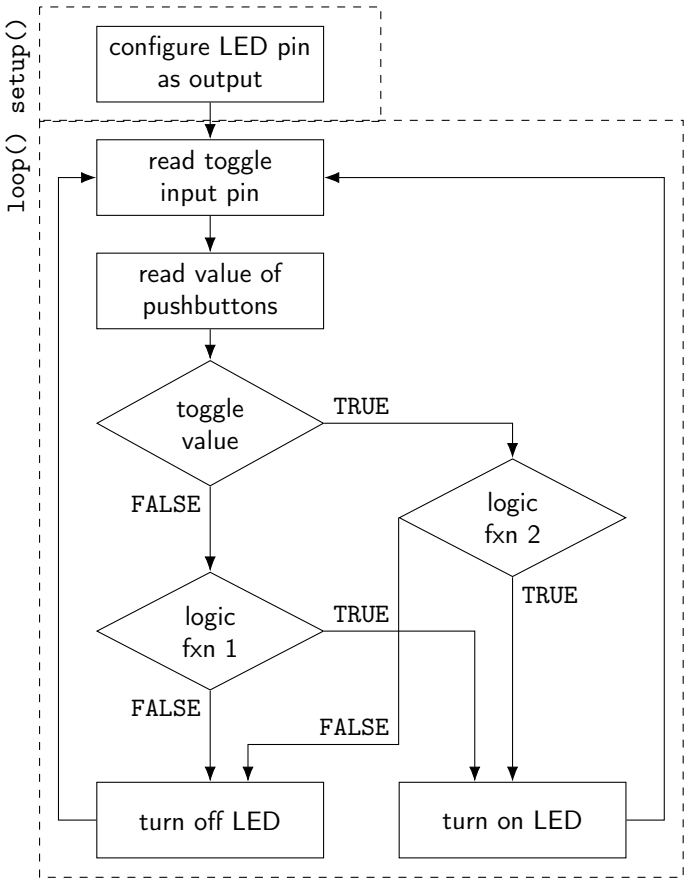
Circuit II: Programmable logic gate

This circuit will be a programmable logic gate. A toggle switch will select the logic function. Two pushbuttons will act as the inputs of the logic function. The output of the logic function will be shown using an LED. When the output is LOW, the LED will remain off. When the output is HIGH, the LED will light up.

Create a programmable logic gate. First decide which two logic functions (AND, OR, NAND, NOR, XOR, XNOR) that will be used and record them below.

Use a toggle switch to switch between the two functions. Two push buttons will be used as the two-bit input. Use an LED to display the output of the logic function. When a button is pushed, its input is a logic HIGH. When a button is not pushed, its input is a logic LOW. All input switches and buttons require the use of either a pull-down or a pull-up resistor! It is up to you to decide if you wish to use an external pull-up/down resistor or the internal pull-up resistor.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

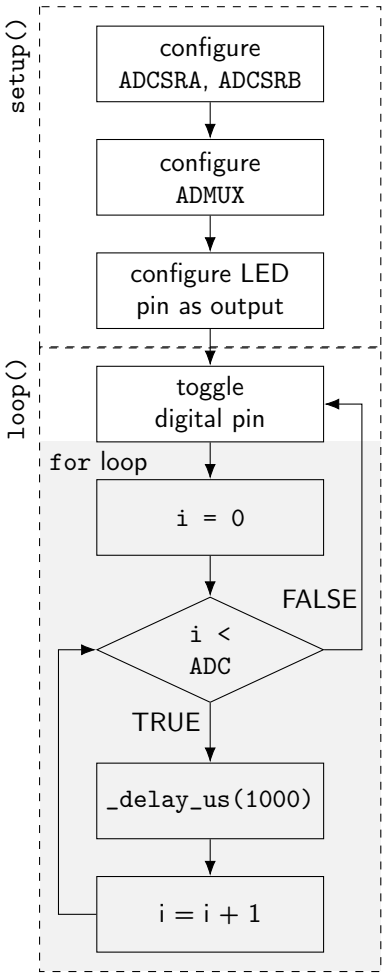
Instructor Stamp: _____

Circuit III: Potentiometer (analog) input

> This circuit will blink an LED on and off at regular intervals. The length of the time intervals will be dictated by the value of the microcontroller ADC. As the potentiometer rotates from one extreme to the other, the time interval will go from 0 ms ON / 0 ms OFF to 1023 ms ON / 1023 ms OFF.

Hook up a potentiometer to an ADC pin and an LED to the a digital pin. Use software to create a delay between LED blinks that is equal to the 10-bit (full-precision) value of the ADC conversion. Use iterative control flow to accomplish this, as `_delay_us()` must have a constant argument. The `for` loop you will use to change the length of the LED blink delay is shown with a shaded background in the flowchart.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 2 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- In Circuit Ib, the use of an internal pull-up resistor generated active LOW logic (as compared to the active HIGH logic in Circuit Ia that used an external pull-down resistor).
  - How **did you** change your code to deal with the active LOW logic?
  - What are two other ways you **could have** changed your code to deal with the active LOW logic?
- In Circuit II, what were the two logic functions that you chose?
- In Circuit II, what were the equations used in the conditional statements to execute each of these logic functions? Copy/paste the relevant C code into this report (i.e. don't just say "we did an AND operation").

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 3

Carefully read the entirety of Lab 3, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. In Circuit 2, you will need to use a character string to represent an integer base ten value coming from the ADC. How many entries will you need in this character string? (I.E. how many entities need to be in the `char` array?) Justify your answer.

2. Identify the problem with the following function.

```
1 void calcSum (int x, int y) {
2   int z = x + y;
3   return z;
4 }
```

3. Identify the problem with the following function. (Don't worry too much about what the function "does", but how it does it.)

```
1 unsigned char writeSPI (unsigned char x) {
2   PORTB |= 0x40;
3   SPDR = x;
4   // wait for data to transmit
5   return SPDR;
6   PORTB &= 0xBF;
7 }
```

Lab 3: Displays

This lab introduces different types of displays, which are crucial in embedded systems when data needs to be visualized. This lab will build off of the knowledge of LEDs and 7-segment displays from Digital Systems to the use of more complicated display systems such as LCD screens. **C Concepts:** arrays, `#include` directive, ASCII code, `int` to `char` conversion, writing functions

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

3.1 Segmented displays

Segmented displays are chips with several LEDs (or LCDs) configured in such a way so as to be able to display alphanumeric characters. The most common is the 7-segment display. 7-segment displays are capable of displaying numerals from 0–9, as well as A–F to display hexadecimal characters (A, b, C, d, E, F). The 7-segment display is not capable of representing all of the characters in the English alphabet. For this reason, 14- and 16-segment displays were created. These displays are capable of representing all alphanumeric characters, with the 16-segment display having the advantage of being the most legible. Figure 3.1 shows a diagram of each of these displays.

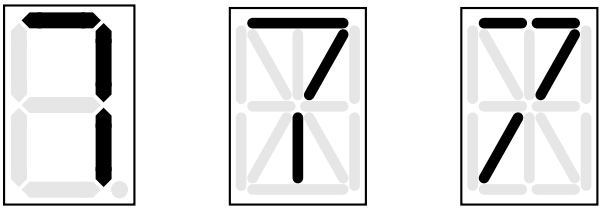


Figure 3.1: Three segmented displays showing the numeral seven: 7-segment (left), 14-segment (center), and 16-segment (right).

Using segmented displays with the ATmega328P microcontroller can be very convenient, given a sufficient number of available I/O pins. There is almost always a tradeoff between ease of use and number of pins that a device uses. To reduce the number of pins used by the display, a display decoder (such as the 7447 BCD to 7-segment decoder) can be used, or information can be sent serially. Serial control of the 7-segment display will be used in Lab 9.

In this lab, 7-segment displays will be used without the decimal point. All seven segments can be changed in parallel by using seven I/O pins to send the control signals. Port D (pins D0 – D7) is the most convenient to use, as a single byte of information sent to the `PORTD` register will simultaneously set or clear each segment of a display as desired.

3.1.1 Wiring a segmented display

A common mistake that students make is in improperly connecting current-limiting resistors to each of the segments in a segmented display. Because 5 V are used on the Arduino, it is important to include current-limiting resistors to ensure that the amount of current running through each of the segments does not burn out the LED in that segment. For the most part, 220 Ω resistors will be sufficient current-limiting resistors in this class. However, when working on a personal project with different displays or a different supply voltage, it is possible to determine the value of the current-limiting resistors that should be included, using the relationship defined by Equation 3.1, where V_{supply} is the value of the maximum voltage supplied to each

segment, V_F is the forward voltage drop of the LED (which can be determined by reading the datasheet for the display), and I_F is the maximum continuous forward current allowed by each segment (also determined by reading the datasheet for the display).

$$R = \frac{V_{supply} - V_F}{I_F} \tag{3.1}$$

For example: the NTE3078 and NTE3079 displays have a V_F value of 1.85 V, and I_F value of 40 mA (0.04 A). When using a supply voltage of 5 V, it can be calculated that the minimum sized current-limiting resistor to use would be 79 Ω. Any resistor size larger than this will protect the segments. Too large of a resistor value will cause each of the segments to become dim.

Because LEDs are non-linear devices, it is important to be careful in how each current-limiting resistor is connected to each segment. It is not appropriate to just connect a current-limiting resistor to each common cathode or common anode pin, because depending on how many segments are on at once the amount of current drawn through the resistor will be different. This can cause problems such as unequal brightness for each numeral that is displayed, and possibly overloading the current when many segments are lit at once. Therefore, **each individual segment needs to have a resistor connected between it and the I/O pin on the ATmega328P**. Note that the resistors are **not** directly connected to either ground or power (which is a common mistake made by students). The circuit diagram in Figure 3.2 shows the correct way to wire each of the resistors. A video with more details about using current-limiting resistors is available at `https://youtu.be/EN3FPsV-pFg`

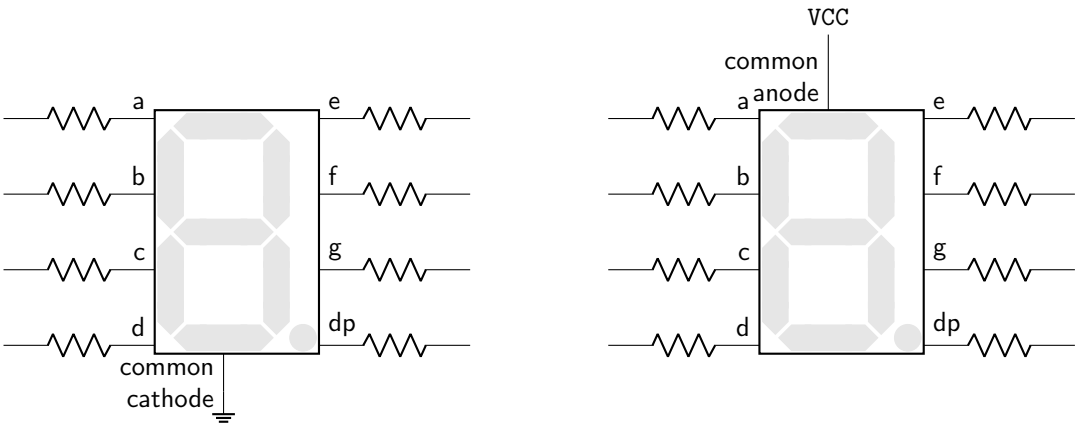


Figure 3.2: Correct connection of current-limiting resistors in a common cathode (left) and common anode (right) 7-segment display.

3.1.2 Common-cathode 4-digit multiplexed 7-segment display

In this lab, in addition to using a single 7-segment display, the ATA8401 display will be used. This display has four multiplexed common-cathode 7-segment displays. Setting one of the cathodes LOW allows that digit on the display to be written by sending HIGH signals to the corresponding segment pins. All of the displays have been mounted onto a PCB with current-limiting resistors connected through each segment and convenient pin headers. The pinout diagram of this PCB is provided in Appendix B.

To write the same digit to all four displays, all four cathodes can be written LOW simultaneously, with the correct segment values sent to each segment pin. To write different numbers to each of the displays, start by writing one of the displays by writing all cathodes HIGH except for the one you wish to write to, which is written LOW. Then write the numeral with a short timed delay afterward. A delay of between 1–10 ms may be required depending on the program. Try a few values until the screen doesn't flicker. A flowchart of this process flow is shown in Figure 3.3.

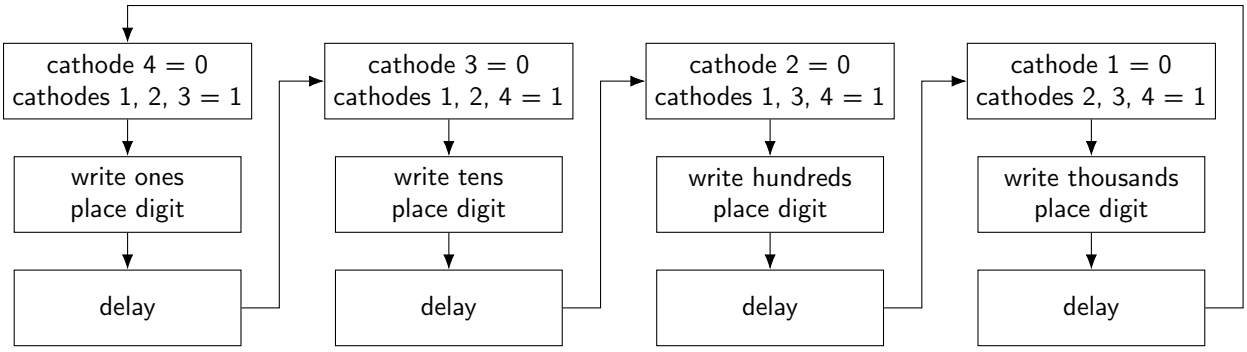


Figure 3.3: Process flowchart for writing data to the four digit multiplexed display.

It is important to note, when using this display, that any code delay for any reason (not just from using a `_delay_ms` or `_delay_us` function) will lead to lag in the multiplexed display, which could manifest as different amounts of brightness on each digit, flickering, or the inability to read some of the digits. Because the ability to automate timed functions in interrupts will not be covered until Lab 6, for now it is important to be cognizant of this fact when using multiplexed displays.

3.2 Liquid crystal display (LCD) screen

In addition to segmented displays, a liquid crystal display (LCD) screen will be used in this lab. LCDs have the ability to display ASCII characters at a relatively low cost. They require additional programming to make them work. Most LCDs make use of the Hitachi HD44780 LCD driver. The one that will be used in this class has 2 lines, 16 columns (16×2 display) with each character represented by 5×8 individual dots. The driver has ROM containing encoding information for ASCII characters. The driver also has built-in RAM, enabling the storage of (volatile) data on the device. RAM will not be used in this or future labs.

To simplify the coding of the LCD screen, the `hd44780` library will be used. It does not come included with the Arduino IDE, therefore the files will need to be downloaded and put into the same folder as the Arduino `.ino` file that references it. The library is included in the file by using the `#include` directive in the Arduino file before the `void setup()` function, as follows.

```
#include "hd44780.h" // note the double quotes
void setup() {
  // setup functions go here
}
void loop() {
  // loop functions go here
}
```

The library contains a file called `hd44780_settings.h` that is where information about how the LCD will be used will be typed in. This will include the mode of operation (4-bit or 8-bit), the clock frequency of the microcontroller, and the location of the pins (port and bit number) used for each control signal.

The LCD screen will be used in 4-bit mode, as this requires four fewer I/O pins than 8-bit operation and allows for the same functionality (at the cost of each command taking twice as long to execute). Because 4-bit mode in write-only operation will be used, 6 I/O pins are required for connections with the LCD screen. Pin 4 on the LCD is the RS (Register Select) pin, pin 6 is the E (Enable) pin, and pins 11-14 are DB4–DB7 (Data Bit 4 – 7).

The commands used on the LCD screen follow.

- `lcd_init()` – Initializes the LCD screen, this must occur before any other LCD commands are sent
- `lcd_clrscr()` – Clears the LCD screen (it is useful to do this at the beginning of `void loop()`)

- `lcd_putc(char displayChar)` – Writes the character `displayChar` on the LCD screen
- `lcd_puts(char displayString)` – Writes the character string `displayString` on the LCD screen
- `lcd_goto(unsigned char goSpot)` – Moves to a new location. See memory locations, below.

A pinout diagram of the LCD screen is included in Appendix B.

3.3 Binary codes

ASCII is a 7-bit character code where every number from 0–127 represents different control characters, numbers, uppercase letters, lowercase letters, or symbols. It is used primarily in two distinct situations. The first situation is when non-numeric information needs to be stored or displayed. The second is when numeric variables need to be displayed on a device that only understands ASCII.

Individual ASCII characters can be saved as variables using the `char` datatype. (The convention is to use `char` for ASCII instead of `unsigned char`, even though the concept of positive and negative ASCII is meaningless.) This can be accomplished using the direct ASCII encoding or by wrapping the desired character in single quotation marks. The following two examples will save the character q into the variable `asciiVar`.

```
1 char asciiVar = 'q'; // note the use of single quotes
2 char asciiVar = 113;
```

It is important to note that the `char` datatype is not limited solely to dealing with ASCII characters. In fact, this datatype has already been used many times to store 8-bit integer numbers. However, the ability to store ASCII characters is an additional benefit to this datatype. Do not confuse "character" and `char`. "Character" refers to an ASCII encoding of an alphanumeric symbol. `char` is a datatype used to store 8 bits of binary data.

The LCD screen is very similar to ASCII for values between 0–127 (there are three characters that differ from ASCII). For values greater than 128, the character set of the hd44780 driver can be used to determine the available characters that can be displayed.

As described above, when using the LCD screen, it is possible to display individual characters using the `lcd_putc()` function. Characters must be enclosed by single quotes. For example, the following will write the lowercase letter q to the LCD screen.

```
1 lcd_putc('q'); // note the use of single quotes
```

To display a character that cannot be accessed on a keyboard (for example, the degree symbol used in temperature), the decimal, binary, or hexadecimal value of the ASII character can be used to display that character on the LCD screen.

```
1 lcd_putc(223); // this will display the degree symbol
```

When using the LCD screen, it is possible to display strings (which are simply arrays of characters) using the `lcd_puts()` function. Strings must be enclosed by double quotes, as shown below.

```
1 lcd_puts("Hello!"); // note the use of single quotes
```

Arrays of characters, known as strings, can also be stored as variables (information about this is included in Appendix E).

When a numeric integer-type (integer as opposed to floating-point, not to be confused with `int`) variable needs to be saved as a string to be displayed onto a screen, that variable needs to be converted using the `itoa()` function, described below.

3.4 Datatype conversion: numeric integer to ASCII

To convert an integer value to a character string, use the following code. The variable `charBuffer` must have a sufficient number of elements to store the maximum number of digits in the desired number system, plus a sign (as needed for negative numbers), plus a terminating null character. If the minimum and maximum possible values of a variable are −5 (this would require three entries: one for the sign, one digit, and one null character) and 479 (this would require four entries: three digits and one null character), then the variable `charBuffer` will require 4 entries. If the minimum and maximum possible values of a variable are −500 (this would require five entries: one sign, three digits, one null character) and 500 (this would require four entries: three digits, one null character), then the variable `charBuffer` will require 5 entries

```
// create an array that will store each ASCII character
char charBuffer[n];

// convert value, in the given base, into the variable charBuffer[]
itoa(value, charBuffer, base);
// parameter value must be of type int or unsigned int
// parameter charBuffer must be of type char
// parameter base must be of type int or unsigned int
```

3.5 Data arrays

In the previous two labs, every value that has been used in code has been assigned as its own variable. However, this isn't always the most practical way to store data. For example, if data coming from the analog to digital converter needs to be stored, it is important to average out data to minimize noise (which is a strategy that will be used in the next lab). It would be nearly impossible to write code that contains individual variables for each data point to average (especially if many data points need to be stored and averaged). Instead, each data point from the ADC can be stored into a single array variable. Each individual value can be accessed by specifying an index. Index numbers always go from 0–(n-1), where n is the number of values in the array. (Special care must be exercised when using a variable to define the number of values that will be stored in an array. This topic will be covered in the next lab.)

3.6 Functions

It is useful to write custom functions (sometimes called subroutines) to accomplish tasks. Instead of repeating blocks of code: jump to a function, execute code, then jump back. In "proper" C a function must be declared before being used. That is not true in the Arduino IDE.

The anatomy of a function is shown below.

```
return_type function_name (parameter_list) {
  body of the function
}
```

The **return type** defines the type of variable (if any) that will be returned by the function. If a function returns no variable, the return type will be `void`, just as with the `setup()` and `loop()` functions in the body of Arduino IDE code.

If a variable is returned by the function, the return statement will cause the function to stop executing, and return to the previous function. Therefore, the return statement must be the last thing that occurs in a function!

The **function name** is the actual name of the function and is how the function will be invoked in code. (For example, `setup` and `loop` are function names.)

The **parameter list** contains any variables (and their datatype) that are needed by the function. Function parameters are optional.

The **function body** contains the code that will run every time the function is invoked.

The following example code shows a function that calculates the sum of two `unsigned char` variables and returns an `unsigned int`.

```
1  void loop() {
2    unsigned char a = 50;
3    unsigned char b = 20;
4
5    unsigned int sumValue = calcSum(a, b);
6  }
7
8  unsigned int calcSum(unsigned char x, unsigned char y) {
9    unsigned int z = x + y;
10   return z;
11 }
```

When using an array as a function parameter, it is good practice to use pointer notation. (This is something you will need to do in the next lab!) This is demonstrated in the code below.

```
1  void loop() {
2    // not shown is the declaration of n
3    // not shown is the declaration of numberArray[n]
4    unsigned char average = calcAverage(numberArray, n);
5  }
6
7  unsigned char calcAverage(unsigned char *x, unsigned char y) {
8    // the * in the parameter list is pointer notation
9    unsigned char temporaryVariable = 0;
10   for (unsigned char j = 0; j < y; j++) {
11     temporaryVariable += x[j];
12   }
13   return temporaryVariable / n;
14 }
```

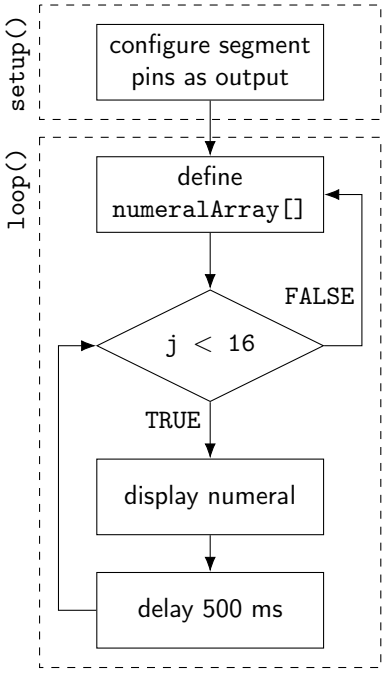
Circuit I: Single 7-segment display

> This circuit will display hexadecimal characters 0–F on a 7-segment display. Every 500 ms, the value on the display will increment. After displaying F, the display will cycle back to 0 again.

Download the file `lab3_circuit1.ino`. Use either a common-cathode or a common-anode 7-segment display. Indicate which was used below.

Connect pins a–g via current-limiting resistors to the Arduino pins chosen in the related activity. Include the numeral array you derived in the same activity in the software code.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

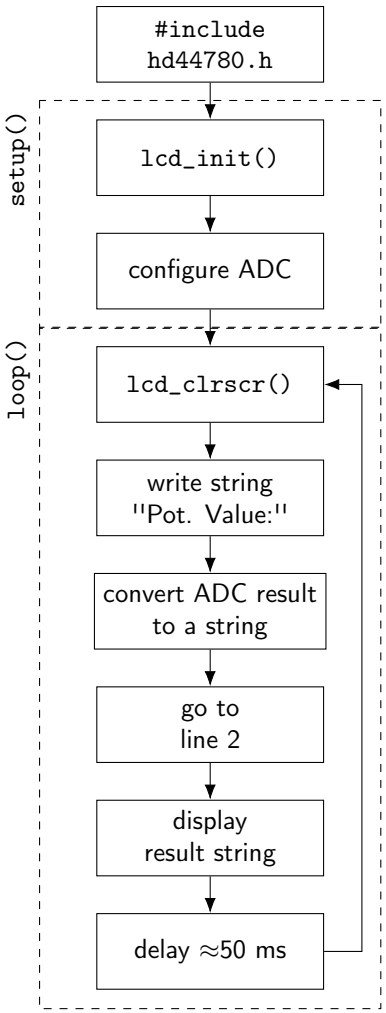
Circuit II: LCD screen

This circuit will display the value from the ADC (coming from a potentiometer) onto an LCD screen. As the potentiometer is rotated from one extreme to the next, the value on the LCD screen will vary from 0–1023.

Download `hd44780.h`, `hd44780.cpp`, and `hd44780_settings.h` and save them in the same folder as the `.ino` file. Wire up the LCD screen using the pinout diagram in Appendix B. Configure `hd44780_settings.h` with the correct port and pin locations of the R/S, E, DB4, DB5, DB6, and DB7 pins.

Use a potentiometer hooked up to one of the ADC pins in full-precision mode. Display the text `Pot. Value:` on line one of the LCD screen, and on line two of the LCD screen display the potentiometer value.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

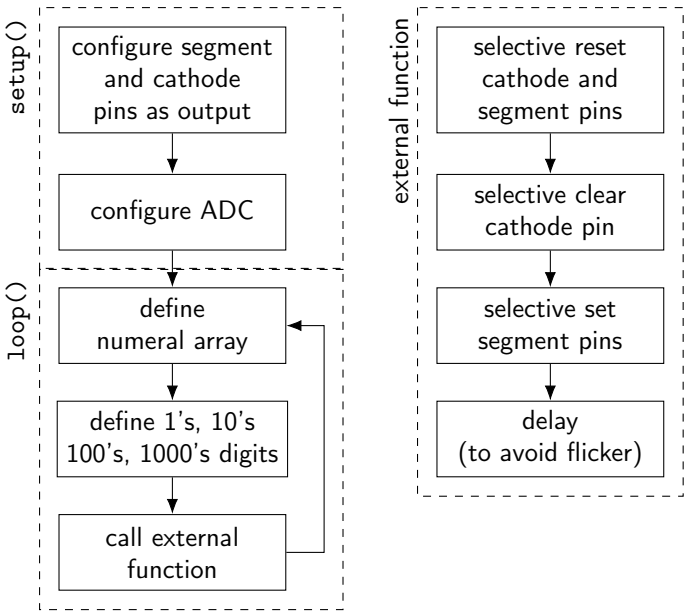
Instructor Stamp: _____

Circuit III: Multiplexed (MUX) 7-segment display

This circuit will display the value from the ADC (coming from a potentiometer) onto a MUX 7-segment display. As the potentiometer is rotated from one extreme to the next, the value on the display will vary from 0–1023.

Use a potentiometer hooked up to one of the analog pins on the Arduino to obtain an input value between 0–1023. Hook up the MUX display PCB to the Arduino. Display the output of the potentiometer on the multiplexed display, with the thousand's place appearing on display 1, the hundred's place appearing on display 2, the ten's place appearing on display 3, and the one's place appearing on display 4. Use an external function to write each digit on the respective display. (It is strongly recommended that you use a `for` loop to iterate through writing each digit. This may or may not be necessary for a stamp.)

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 3 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Include a table, like the one below, in your report, filled out with the relevant information.

Display	Ease of Use	# of I/O Pins	Ability to Display Characters	Memory Usage
7-Segment				
MUX Display				
LCD				

- In what types of designs might you use each of these types of displays?
- Specifically, how might you integrate one or more of these displays into your Smart Car project?
- In Circuit III, how did you isolate each numeral in the potentiometer ADC value? Include the equations you used.
- In Circuit III, how did you isolate each cathode to write to only a single display at a time? Include the relevant software code.

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring

in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 4

Carefully read the entirety of Lab 4, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. You will calculate temperature from a sensor in Circuit 1 and display it on an LCD screen using tenth's place precision. What is the **actual** precision of the TMP36 sensor using the ADC in 10-bit mode? (Note: this is important to know, so that you know if your circuit is displaying reasonable values or not.)

2. How many bytes of data do the following datatypes use?
   (a) `char`

   (b) `int`

   (c) `long`

   (d) `float`

3. What are two reasons not to use floating-point arithmetic?
   (a)

   (b)

4. Explain the use of the `L` in the equation `int y = (90L * x) / 8 + 37`

5. In what specific situations (if any) must a variable be defined as a `const`?

6. What is a `volatile` variable? When should they be used in software code?

7. What is a global variable? When should they be used in software code?

8. When obtaining a rolling average of an array, what undesirable outcomes might result if n is...
   (a) ...too small?

   (b) ...too large?

Lab 4: Sensors and Sensor Calibration

In this lab, several sensors will be used. Sensors provide information about how an embedded systems are working, and in what kind of environment. In the process, sensor calibration techniques will be used to calibrate each one to obtain accurate and meaningful results. **C Concepts:** `long` datatype, `float` datatype, `const` variables, `static` variables, variable scope, `volatile` variables

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

4.1 Sensors

Sensors are devices that take a piece of meaningful information corresponding to a physical quantity (for example speed, sound intensity, distance, or light level) and convert it to an electrical signal. On the most basic level, switches such as toggles and pushbuttons are sensors, outputting digital values corresponding to HIGH or LOW voltage levels based on their configuration.

4.1.1 TMP36 temperature sensor

The first sensor that will be utilized in this lab is the TMP36 temperature sensor. It is a 3-pin device that sends an analog voltage level corresponding to the ambient temperature. The TMP36 has an internal diode whose voltage level scales as a function of temperature. That voltage level can be used to provide information about the temperature of the surroundings. The TMP36 sensor outputs a voltage of 0 V (which would be converted by the ADC to a value of 0) when the ambient temperature is -50 °C. The output is 5 V (ADC of 1023) when the ambient temperature is 450 °C. In between, the device scales linearly. A pinout diagram of the TMP36 is included in Appendix B.

4.1.2 Photoresistor

A photoresistor can be used to provide information about ambient light levels. The photoresistor that will be used in this lab is made of CdS, a semiconducting material that naturally has a very high resistance. When photons (ambient light) are incident on the surface, energy imparted by the photons is absorbed by electrons which then have enough energy to better conduct electricity. Any frequency of light which meets or exceeds the requirement for this energy will be sufficient to reduce the resistance of the device. This photoresistor is sensitive to light at any frequency within the visible spectrum.

By using the photoresistor in series with another resistor of fixed value, a voltage divider is created. As the resistance of the photoresistor goes down (corresponding to high light levels), the voltage drop over the device will decrease. If no light is incident on the photoresistor, the voltage drop over the device will be maximized.

4.1.3 Soft potentiometer

The soft potentiometer (soft pot) is a pot in which your finger, a stylus, or other object touching the strip acts as the wiper. The pinout diagram of the soft pot is included in Appendix B.

4.2 Sensor calibration

All of the sensors used in this lab will be configured to provide an analog voltage between 0–5 V. These values then need to be converted into meaningful values corresponding to a physical quantity (room temperature, light level, and position for the above three sensors). The first conversion that takes place is at the ADC

inside the ATmega328P microcontroller. This leads to a value between 0–1023 that can be written to a display device.

If the output response of the device is known (and is trustworthy), then that information can be used directly to determine a conversion between the ADC value and the physical quantity. The TMP36 temperature sensor has a known response: an ADC value of 0 corresponds to a temperature of −50 °C, and an ADC value of 1023 corresponds to a temperature of 450 °C. The data in between scales linearly, as shown in Figure 4.1.

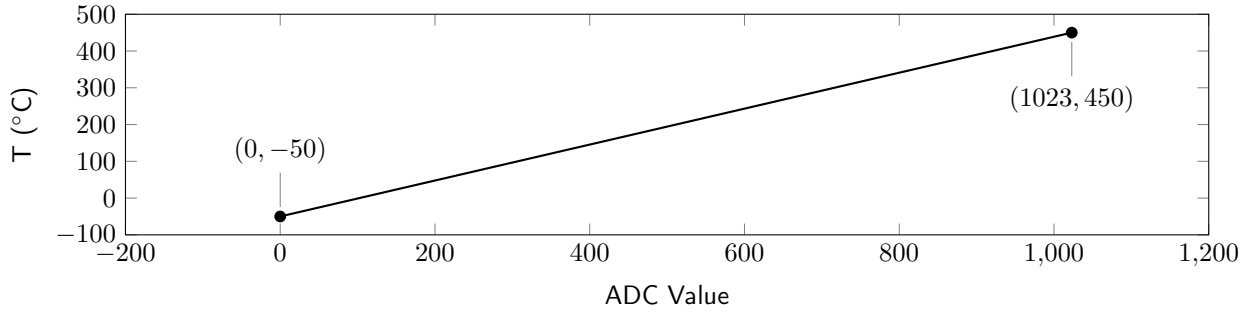


Figure 4.1: Ambient temperature plotted as a function of ADC value for the TMP36. Data is based on the device datasheet.

Using this linear data and the equation $y = mx + b$, a relationship between the physical quantity (y) and the ADC value (x) can be derived. This relationship can be used to calculate the temperature of the room. The equation for the TMP36 temperature sensor is $y = 500 * x/1024 - 50$, or $y = 500 * x/2^{10} - 50$. In code, this can be expressed as follows. (Note the use of the bitshift right operator, which divides by powers of two. Bitshifting right 10 times divides by 10 powers of 2, or 1024.)

```
1 int T = (500L * ADC) >> 10 - 50;
```

4.2.1 One-point calibration

It is possible that a sensor does not provide the exact output that is expected based on a trustworthy calibration measurement. (For example, the TMP36 sensor may not output the temperature that is expected based on a thermometer that is known to be accurate.) In these situations, it is necessary to calibrate the data.

One-point calibration is necessary when the slope of the data is correct, but there is an offset between the measured and actual data. If the measured data is greater than the ideal response, then the offset must be subtracted from the calibration equation. Otherwise, the offset must be added to the calibration equation.

4.2.2 Multiple-point calibration

If the output response of the device is not known, then the first step in sensor calibration will be to find it. Using as many known reference points as possible, determine the ADC value that results based on the corresponding physical quantity. Use plotting software such as Excel to find a best-fit line through all of the points and to obtain a calibration equation.

4.3 Datatypes: `long` and `float`

Two new datatypes will be introduced in this lab: `long` and `float`. The `long` datatype is 32 bits, and it becomes useful when working with very large integer numbers. However, every variable using a `long` datatype will take up 4-bytes of data memory. Do not use this datatype unless it is necessary to do so! Examples of how to use the `long` datatype are included in Appendix E.

Numbers that are not integers and numbers that are too large to fit into the `long` datatype must be represented as floating-point values. In the Arduino IDE, the `float` datatype is used to represent floating-point numbers. Floating-point is a method of encoding non-integer numbers into binary.

There many downsides to working with floating-point numbers (apart from requiring 4 bytes of memory for each variable). First, floating-point arithmetic is very slow and requires massive amounts of program memory to execute. In addition, floats only have 6–7 decimal digits of precision. That means the total number of digits, not the number to the right of the decimal point. Finally, floating-point numbers are not exact, and may yield strange results when compared. For example 6.0 / 3.0 may not equal 2.0. More information about the `float` datatype is included in Appendix E.

4.4 Integer vs. floating-point operations

Because of the fact that floating-point operations take a lot of time and memory to execute, it will be imperative to avoid them as much as possible. Not only should the `float` datatype be avoided, but any arithmetic including non-integer numbers should be avoided as well. For example, `y = (2 * x) / 3 + 10` avoids all use of floating-point operations. A nominally identical statement, `y = 0.67 * x + 10`, will consume much more memory and take much longer to execute. **For this reason, all calibration equations that you derive should use integer values.**

Depending on the datatype used for the final value of the calibration equations, it is possible that an overflow can occur while performing arithmetic. The intermediate results of all arithmetic is stored as an `int` (16 bits). Given the equation `unsigned int y = (500 * x) / 15 + 6`, values of `x` larger than 131 will lead to an invalid value. For this reason, the temporary results of the arithmetic should be stored in more than two bytes. Using `unsigned int y = (500L * x) / 15 + 6` will solve this problem.

4.5 Sensor value precision

The precision of the final value calculated by the microcontroller is dependent on what datatype is used to represent the information. For integer datatypes (`char`, `int`, `long`) this precision is limited by integer math; only whole numbers can be represented. For example, given an ADC value of 146, if `int temp = (500 * num) » 10 - 50`, the following will occur.

$$\begin{aligned} \text{int}(500 * 146 >> 10) - 50 = \\ \text{int}(71.29) - 50 = \\ 71 - 50 = 21 \end{aligned}$$

To obtain tenth's place precision, multiply the entire equation by 10. For hundredth's place precision, multiply by 100, etc. When displaying this number, place a decimal point on the display in the correct location. If further calculations need to be made with this number, the final result will be affected! The previous example, with tenth's place precision, would lead to `int temp = (5000 * num) » 10 - 500`, and thus the following will occur.

$$\begin{aligned} \text{int}(5000 * 146 >> 10) - 500 = \\ \text{int}(712.89) - 500 = \\ 712 - 500 = 212 \ (21.2) \end{aligned}$$

4.6 `const` variables

There are times when a variable that can never change its value must be used. In this case, the `const` keyword can be used to indicate that a variable is unchanging. The `const` keyword can be used with all datatypes. If the size of an array is to be defined by a variable (rather than just a number), then that variable **must** be a `const` variable. A compiler error will result in the Arduino IDE if a value is assigned to a variable that has already been defined as a `const`.

4.7 `static` variables

The keyword `static` in front of a variable refers to how long the variable is active in memory. Variables without this keyword are known as automatic, meaning that they come into existence when they are declared, and then expire whenever the function or loop in which they reside has finished running. A `static` variable exists in memory for as long as the program is running. This means that, even if they are declared with a certain value inside of a function or a loop, they can be changed for as long as the program runs. This allows for the creation of non-global variables that can change within a function or a loop.

4.8 Dealing with fluctuating data and sensor noise

When sensor values can fluctuate due to noise, it is important to perform some type of rolling average to obtain a steady, reliable readout. To create a rolling average, the most recent n values of a sensor will be averaged. These values must be stored in an array.

Careful consideration has to go into choosing a proper value for n. Values that are small use less memory and take less time to average than values that are large. However, larger values give much more stable readings. If n is too large, a sensor becomes much less sensitive to short-term changes.

In addition to taking a rolling average, it's possible that sampling data at longer intervals (rather than sampling every time the `loop` iterates, or sampling every time the ADC completes a conversion) may also help to filter out noise. This can be accomplished by using timed functions, or using a timer/counter to trigger ADC conversions. This will be the topic of a future lab.

4.8.1 Circular buffer

One way to take a rolling average is to use a circular buffer. A circular buffer is an array of n values where the sensor data is stored. The first data point is saved in array index 0, the second in array index 1, and so on until the n^{th} data point is saved in array index $n-1$. Then the cycle continues again, with the next datapoint saved in index 0 overwriting what is by this point the oldest datapoint. The following code demonstrates a circular buffer. Note that the `unsigned long` datatype is used for `x`, as it is incremented continuously without getting reset to 0. In addition, note the use of `static` keyword for `x`, which enables it to be a local variable without being re-initialized every time the function executes.

```
// variable n (const) is used to define the size of the array
// the declaration of arrayValues and n is not shown
static unsigned long x = 0;
arrayValues[x % n] = sensorValue;
x++;
```

To avoid using the `unsigned long` datatype, instead you can use an incrementing variable that is the same datatype as the number of values in the array (n). This saves some data memory at the expense of the extra program memory needed to include a conditional statement.

```
// variable n (const) is used to define the size of the array
// the declaration of arrayValues and n is not shown
static unsigned char x = 0; // must be same data type as n
arrayValues[x] = sensorValue;
x++;
if (x == n) x = 0;
```

4.9 Variable scope

The scope of a variable refers to what functions can access that variable. A variable defined within a function can only be accessed by that function. A variable that is defined outside of all functions can be accessed by every function in the code, and is called a **global variable**. Global variables should only be used in

situations where it is necessary. Otherwise, it is recommended that the scope of all variables be limited by keeping them within the function in which they are used.

It is important to ensure that global variable names are different from all local variable names and function parameter values!

Global variables will be necessary in this lab because you will want to update an array of values every time the ADC completes a conversion and obtains a new value using something called an interrupt. While interrupts will be the topic of the next lab, for now it is enough to state that variables cannot be sent to an interrupt service routine (ISR). An ISR is a type of subroutine that executes when an interrupt is triggered. If you wish to use one or more variables in an interrupt service routine and the `setup` or `loop` functions, then those variables must be globally defined.

The ADC is capable of triggering an interrupt when a conversion is complete. We did not need to utilize this interrupt in earlier labs as we simply wanted to use the current value of the ADC to change the blink-rate of an LED or to display on a screen. However, we wish to store an array of the most recent values of the ADC to use in a circular buffer. In order to ensure that we are only updating this circular buffer when the ADC receives newly converted data, we must use the ADC interrupt.

4.10 `volatile` variables

When a compiler takes C code and translates it into assembly language, it attempts to optimize that code. At times, it may appear that a global variable is unused by functions, especially in the case of an ISR. An ISR is never formally invoked (or called) by any function, and it may appear to the compiler as if any variables that are used within an ISR are unused (in which case it may discard or ignore the variable, rather than storing it in memory) or unchanging (in which case it saves it in program memory as a constant value, rather than in data memory as a changing value). By creating a `volatile` variable, the compiler will not discard the variable or treat it as a constant. All datatypes can be saved as `volatile` variables. The code that will be used to update your circular buffer of ADC values follows, which shows an example of an array of type `volatile unsigned int`.

```
1  const unsigned char n = 10;
2  volatile unsigned int adcValues[n] = {};
3
4  void loop() {
5    // code goes here
6  }
7
8  ISR(ADC_vect) {
9    static unsigned char x = 0;
10   adcValues[x] = ADC;
11   x++;
12   if (x==n) x = 0;
13 }
```

Circuit I: Calibrated and averaged temperature measurement

This circuit will display the ambient temperature on a display.

Consult the flowchart on page 55. Decide if you wish to use an LCD screen or a MUX 7-segment display. Use code from activities and labs as needed to set up the ADC to receive data from the TMP36 temperature sensor and display on the output device with tenth's place precision. To display units, you can try using ASCII character number 223 for the degree (°) symbol. Any delays in your code should be as brief as possible and must be justified in your lab report. (Large delays that you cannot justify with a good reason for using them will result in only partial credit on this circuit.)

Wire the temperature sensor with a bypass capacitor and 47 kΩ pull-up resistor as shown in Figure 4.2. This capacitor should be as physically close to the sensor as possible.

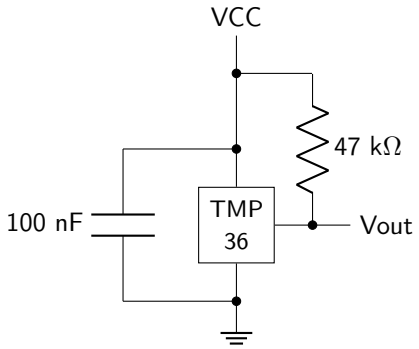


Figure 4.2: TMP36 wired with a bypass capacitor to filter out noise.

Change the value of n from approximately 10 to approximately 100. Decide on the value of n that you think is best, keeping in mind how quickly the sensor takes to setup, how much data memory is required, and how stable the output is. You will have to justify this value in your lab report using reasoning based on your tinkering with the value. Don't just pick one value and say that "it worked fine and we didn't feel like trying anything else." Record the value of n below.

Use a digital thermometer to measure the actual temperature in the vicinity of your temperature sensor. If there is a difference in temperature of over 1 °C, include a variable `unsigned char offset` in the code to reduce the difference to less than 1 °C. Record the offset value (if used) below.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Photoresistor calibration and light level measurements

This circuit will display the ambient light level on a display.

Consult the flowchart on page 55. Use the display of your choice to display the light level (with one's place precision) using the photoresistor. You will need to calibrate the device and find the best-fit equation. Find the ADC value corresponding to 0% light level by completely covering up the photoresistor with your hand. Find the value corresponding to 100% light level by aiming your cellphone flashlight directly at the photoresistor. Record these values in Table 4.1.

Light Level	ADC Value
0%	
100%	

Table 4.1: Calibration data for the photoresistor.

Plot these two data points in plotting software such as Excel, and find the best fit line. (Assume that the ADC value scales linearly with the light level.) You (and your lab partner) will need to include this graph with your lab report, so save or print a copy. Use integer math for your equation; floating-point math will not be accepted for the final version of this circuit. Record this equation below.

(Do try a floating-point equation and compile to see how much memory is used. Make note of this to compare with the integer version in your lab report.)

The photoresistor will be wired as shown in Figure 4.3

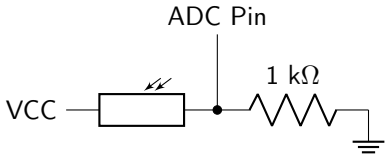


Figure 4.3: Schematic diagram for the photoresistor.

Change the value of n from approximately 10 to approximately 100. Determine the value of n that works best for **this** circuit. Record this value below. You will have to justify this value in your lab report, so take notes the effect of different values in this circuit. It very likely will not be the same value you used in the last circuit!

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: Soft potentiometer calibration and distance measurements

This circuit will display the distance of a stylus on a soft pot on a display.

Consult the flowchart on page 55. Use the display of your choice to display the distance (in mm) from the edge of the soft pot closest to the leads (with one's place precision). Calibrate the device and find the best-fit equation. Use a ruler to find the ADC value corresponding to distances between 10–50 mm, and record the ADC value in Table 4.2.

Distance	ADC Value
10 mm	
20 mm	
30 mm	
40 mm	
50 mm	

Table 4.2: Calibration data for the soft potentiometer.

Plot these two data points in plotting software and find the best fit line. You (and your lab partner) will need to include this graph with your lab report, so save or print a copy. Use integer math for your equation; floating-point math will not be accepted for the final version of this circuit. Record this equation below.

The soft pot will be wired as shown in Figure 4.4.

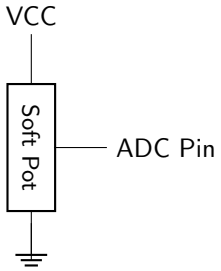


Figure 4.4: Schematic diagram for the soft pot.

Change the value of n from approximately 10 to approximately 100. Determine the value of n that works best for **this** circuit. You will have to justify this value in the lab report. Do not just use the same value as you used in the previous circuit. Record the n value you use below.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Schematics note: If there is no soft potentiometer in your schematics software, use a regular potentiometer and label it instead.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

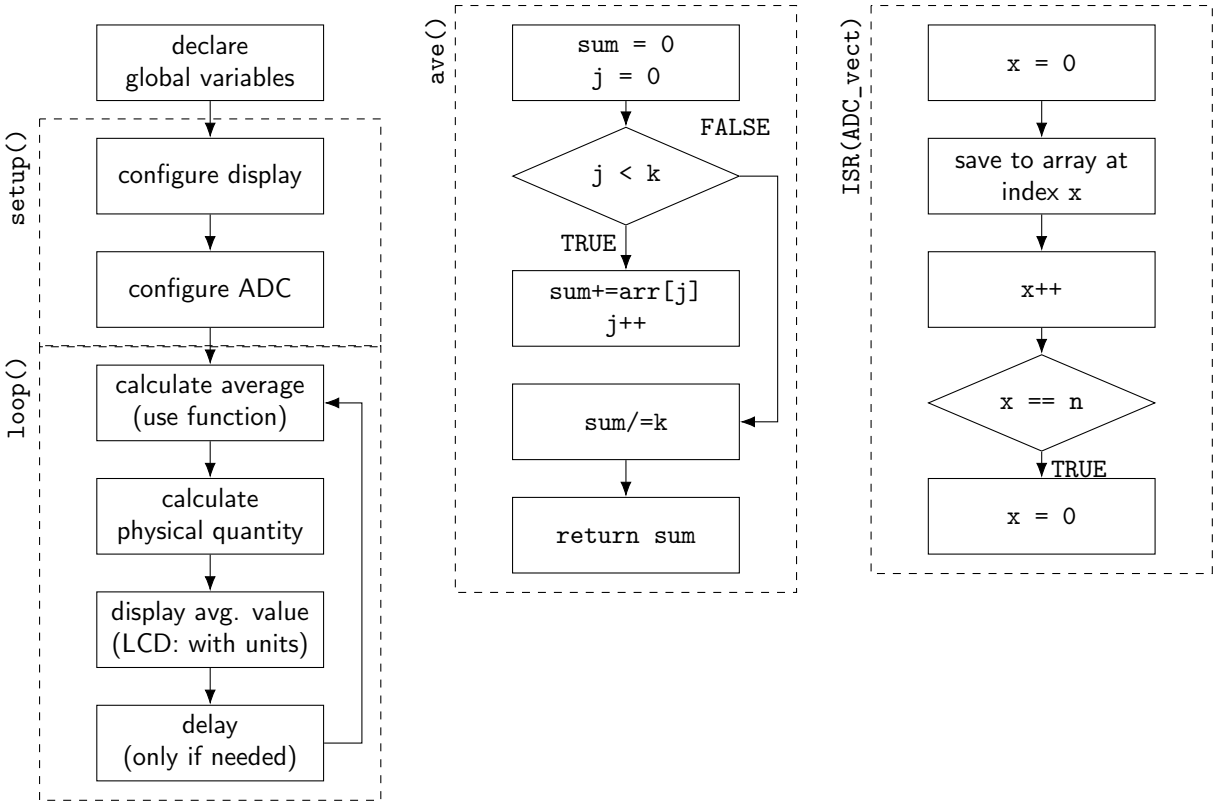


Figure 4.5: Flowchart to be used for all of the circuits in this lab.

Lab 4 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- For each circuit include the following information.
  - What, if any, delays did you use in your code? Why? (If you did not include any delay, specifically state that you didn't use one.)
  - How many elements were included in the circular buffer for averaging out sensor noise? Exactly how did you select this value? What issues did you encounter when you tried using smaller or larger values?
  - For each circuit where you were asked to create a graph, include the graph (with properly labeled axes). Be sure that all of the data points are clearly present on the graph (in other words, don't just show the fit line).
  - What was the full equation that you used to relate the ADC value to the sensor value? If the temperature sensor required an offset, include that in your equation and explain how you determined its value. (Include all multiplicative factors used to increase precision, for example with the temperature sensor!)
  - Based on your sensor equation, what are the smallest and largest possible sensor values? What datatype did you use to store this data? Was this the best choice of datatype? If you should have selected a different datatype, what should you have used instead?
- In one of the circuits, you were asked to try a floating-point operation to determine the impact on memory overhead. How much program and data memory were used with floating-point math? How much were used with integer math?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 5

Carefully read the entirety of Lab 5, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Find four errors in the following code and explain why they are problematic. (Note: there are more than four errors in the following code!)

```
unsigned char a = 0;

void setup() {
  EICRA = 0x0F;
  EIMSK = 0x03;
}

void loop() {
  unsigned char b = ISR(INT1_vect);
  //code that writes a and b to a display
}

ISR(INT0_vect) {
  a = addNumbers();
}

ISR(INT1_vect) {
  b++;
  return b;
}

unsigned char addNumbers() {
  a += b;
  return a;
}
```

   (a)

   (b)

   (c)

   (d)

2. In your own words, explain the difference between hardware interrupts and software interrupts. Give an example of each.

3. Explain how (if at all possible) to configure external interrupt registers to trigger an ISR upon a rising-edge signal only.

4. Explain how (if at all possible) to configure pin change interrupt registers to trigger an ISR upon a rising-edge signal only.

5. Why does the WDT use a separate clock from the rest of the AVR microcontroller? (This question is not answered in the lab, I want you to think about it!)

Lab 5: External Interrupts

In this lab, external interrupts on the Arduino will be used to instantly address changes detected on digital input pins. In addition, the Watchdog Timer (WDT) will be used to explore how to automatically reset devices that have stopped working properly. **C Concepts:** `cli()` and `sei()` functions, switch case **AVR Concepts:** `SREG`, External Interrupt registers `EICRA`, `EIMSK`, Pin Change Interrupt registers `PCICR`, `PCMSK2`, `PCMSK1`, `PCMSK0`, Watchdog Timer (WDT), WDT register `WDTCSR`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

5.1 Interrupts

Interrupts are used in embedded systems to respond to either asynchronous or timed events. To determine if a pushbutton is pressed, code can be written to continuously check the status of the pushbutton pin. This is called polling and can unnecessarily consume microcontroller resources. In complicated embedded systems with a lot of input devices to monitor, performance can suffer if the software code is continuously checking the status of each peripheral. If too many peripherals are being polled, the microcontroller may never have a chance to perform core operations. Advanced features such as sleep modes or power-down modes may never be activated if continuous polling is taking place.

Interrupts allow software code to respond to events as they occur. In addition to eliminating the above-mentioned performance drawbacks of using continuous polling, using interrupts decreases the lag time between the event (for example: pushing a button) and the desired action (subroutine) which is called in that situation. A subroutine that is called when an interrupt is invoked is known as an interrupt service routine (ISR). There are several types of interrupts that can be utilized with the AVR microcontroller, but they can mostly be organized in two categories: hardware and software interrupts. Several interrupts are listed below in Table 5.1.

Hardware	Software
External	Timed
Pin Change	Watchdog Timer (WDT)
Reset	Serial I/O
	ADC
	Memory

Table 5.1: Sources of hardware and software interrupts on the ATmega328P.

5.1.1 Hardware interrupts

Hardware interrupts are asynchronous and respond to external events. **External interrupts** refer to interrupts that take place on pins D2 and D3 on the Arduino. Each of these pins has the ability to be set up to trigger an interrupt upon a logic LOW signal, a toggle, a falling edge, or a rising edge.

Pin change interrupts expand on this concept by allowing all of the pins on the Arduino to trigger interrupts. The main drawback of a pin change interrupt is that it requires additional software code to

determine exactly what pin triggered the interrupt. Pin change interrupts are also triggered upon all changes in pin status and cannot be configured in hardware to select only rising- or falling-edge events.

The **reset interrupt** is a very special interrupt on the Arduino. There is a reset pin included on the Arduino Uno board that, if pressed, causes the device to reset. It is a special interrupt in that it supersedes any other interrupt on the Arduino. If the reset button is pressed, regardless of what happens in regard to any other external interrupt, the Arduino will reset.

5.1.2 Software interrupts

There are several types of software interrupts. Some of them are synchronous and pertain to the timer/-counter functionality of the Arduino. These will be used in Lab 6. However, the Watchdog Timer (WDT) is a straightforward enough timed interrupt to be included with this lab.

Other interrupts occur when the microcontroller finishes processing information. For example, when an ADC conversion is complete, an interrupt is invoked. This interrupt was utilized in Lab 4. The related ISR allows data to be saved from the ADC immediately after the ADC finishes the conversion (or upon another desired trigger event). In addition to the ADC interrupt, there are interrupts related to serial communication, as well as memory events. Appendix C has the entire ATmega328P interrupt vector table.

5.2 Interrupt service routines (ISRs)

Interrupt service routines (ISRs) are different from most functions in that they are never formally invoked. Therefore, there are no parameters that can be passed to the function, and the function cannot return any values. The only way for an ISR to communicate with the rest of the program is to use global variables. Because ISRs are never called by any functions, any global variables that are modified by an ISR must be declared as a `volatile` datatype.

ISRs generally should not contain extensive processing steps, so as not to take away from the core functioning of the microcontroller. (This is true particularly when the `loop` has important code to carry out. But this may not be universally true.) For example, function calls should not be made from within an ISR. Usually an ISR modifies or sets global variables in a way to indicate that an external event has occurred.

5.2.1 Enabling / disabling interrupts

Interrupts can be enabled or disabled at different points in the software code. This can be very useful especially when processing important information; disabling interrupts ensures that the important data is processed and stored in memory without being affected by any interrupting subroutine(s). There is a status register (`SREG`) in the AVR that contains information regarding the result of the most recently executed arithmetic instruction. Bit 7 in `SREG` is the global interrupt enable bit. This bit must be set in order for interrupts to be enabled. If the bit is cleared, then interrupts will be disabled throughout the system. The global interrupt enable bit is set by calling the function `sei()`, and is cleared by calling the function `cli()`.

The reset interrupt, being the most important interrupt on the microcontroller, cannot be disabled, even when bit 7 in `SREG` is 0. Individual interrupts can be enabled or disabled by using the corresponding mask register for the interrupt. Each of these mask registers are discussed in detail in Appendix A.

Henceforth, it is strongly recommended (and usually required) for all setup code to temporarily disable interrupts so as not to affect important configuration tasks. Sample code follows.

```
1 void setup() {
2   cli();
3   // setup code goes here
4   sei();
5 }
```

When code contains critical control flow steps (such as branching conditional logic), it is important to preserve the contents of `SREG`. When control flow occurs, a comparison operation occurs in machine language and

flags will be set in SREG that will determine where the program counter goes next. (For example, if the carry flag in SREG is set, the code may execute the if part of conditional control flow, otherwise it may execute the else part.) When an interrupt is invoked, the contents of SREG are not automatically stored. This means that when the program returns from the ISR, control flow may be negatively affected.

There are two ways to avoid the negative effects of an interrupt on control flow. The first is to mask and unmask interrupts around control flow logic using cli() and sei(). However, this is a bad idea if interrupts will be masked for a significant amount of time. Because interrupts may be asynchronous, if they occur when interrupts are masked, they will be ignored. That could lead to missing an important event!

The second way to store SREG without masking interrupts is to include a global volatile variable where SREG can be stored during an ISR so that it can be restored when the ISR is complete. This is shown in the example code below (note that saving SREG must be the very first step in the ISR, and restoring SREG must be the very last step in the ISR).

```
1 volatile unsigned char sregContents;
2
3 ISR(INT0_vect) {
4   sregContents = SREG;
5   // all ISR code goes here
6   SREG = sregContents;
7 }
```

5.3 Configuring external interrupts

There are two registers involved with configuring external interrupts: EICRA and EIMSK. If EIMSK is configured with enabled interrupts, and the corresponding pin changes as defined by EICRA, the interrupt vectors INT0 (for pin D2) or INT1 (for pin D3) will be invoked. Because INT0 has a lower memory address than INT1, it will take priority if both interrupts are invoked simultaneously. These registers are explained in more detail in Appendix A.

- EICRA – **External Interrupt Control Register A**: This register stores information about how external interrupts should be triggered on pins D2 and D3. This register enables the use of either or both pins (D2 and D3) to trigger interrupts under different conditions.
- EIMSK – **External Interrupt Mask Register**: This register contains two bits to enable interrupts on pins D2 and D3.

5.4 Configuring pin change interrupts

There are four registers involved with configuring pin change interrupts: PCICR, PCMSK2, PCMSK1 and PCMSK0. Pin change interrupts are triggered at any change in pin status (i.e. both falling and rising edge conditions will trigger an interrupt) and any information about the trigger condition must be determined in software. More details about these registers is available in Appendix A.

- PCICR – **Pin Change Interrupt Control Register**: This register contains three bits which dictate whether or not interrupts are enabled on each of the three I/O ports.
- PCMSK0 – **Pin Change Mask Register 0**: This register allows pin-change interrupts to be enabled on individual pins in PORTB when their respective bit locations are set.
- PCMSK1 – **Pin Change Mask Register 1**: This register allows pin-change interrupts to be enabled on individual pins in PORTC when their respective bit locations are set.
- PCMSK2 – **Pin Change Mask Register 2**: This register allows pin-change interrupts to be enabled on individual pins in PORTD when their respective bit locations are set.

5.5 Watchdog timer (WDT)

The so-called "Blue Screen of Death" occurs on Windows computers when they are no longer running properly. It is a visual notification that a system reboot is required. Microcontrollers do not necessarily provide any indication when they are not functioning as they should. When an ATmega328P is to be used for extended periods of time, when it is to be used for critical functions, or placed in difficult-to-access locations, it may be necessary to automatically reboot the system if it hangs up and no longer functions correctly.

The watchdog timer (WDT) on the microcontroller can be used to force a system restart. The WDT can be set to expire in a given timeframe (between 16 ms and 8 s), and if the WDT isn't refreshed within that time period, the device will reset. Consider the WDT to be like a dog. If you pet the dog periodically (before the timeframe is up), the dog will be content. However, if you do not pet the dog (refresh the WDT), then the dog will jump up and notify you that you have done something wrong!

To utilize the WDT, first globally disable all interrupts (so as not to accidentally trigger the WDT during setup), then properly configure the WDT register `WDTCSR` which is a two-step process.

1. Using a bitwise OR operation, simultaneously set the WDT change enable bit (`WDCE`) and the WDT system reset enable bit (`WDE`).
2. In the command immediately following the above, using an assignment operation, set `WDE` and write the desired values of `WDP[3:0]` to the register. (Assign a zero to all other bits in the register.)

After the WDT register has been configured, globally enable all interrupts. Periodically within `loop()`, make a call to the AVR instruction `wdr` (Watchdog Reset) by invoking the assembly command `asm volatile ("wdr")`. If the code runs for longer than the time set with the WDT prescaler without the `wdr` instruction, then the system will reboot. The watchdog timer control register is explained in more detail in Appendix A.

- `WDTCSR` – **Watchdog Timer Control Register**: This register contains information for enabling and configuring the watchdog timer (WDT).

5.6 Switch case

Sometimes code needs to take a different action based on the value of a variable. In these situations, conditional control flow is executed using a switch case statement. Typically, a switch case is used instead of if / else control flow in the case where a single variable is considered as the condition, and the conditional logic branches based on the particular value of that variable.

For instance, a switch case would be used in the situation where a variable (say, `x`) can be some number (say, between 0 and 9), and depending on that specific value, a different piece of code will execute.

However, if one or more variables are used with comparison and/or Boolean operations, then if / else should be used instead.

Circuit I: Keypad adder

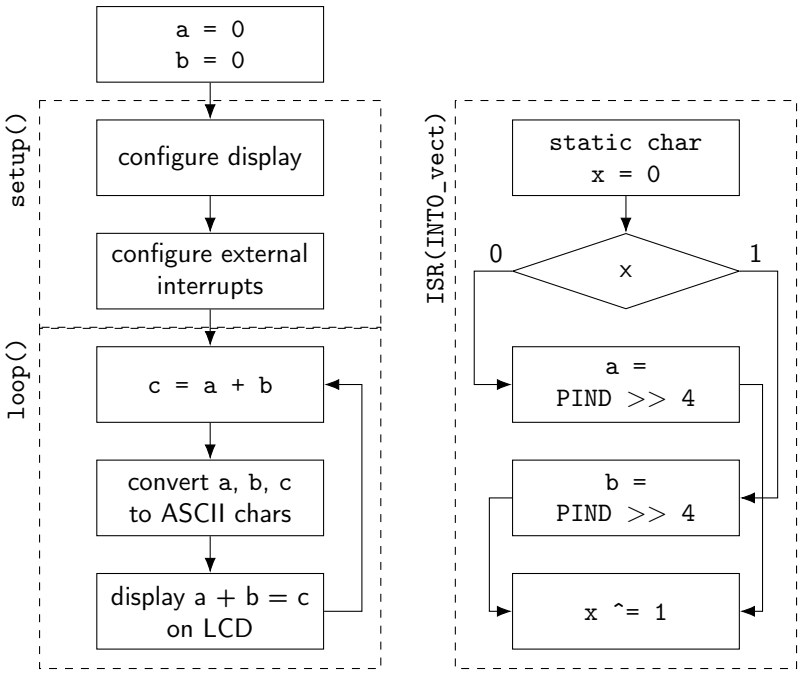
This circuit will display the sum of two values input from a keypad onto an LCD screen. It will also display the two numbers that are being added together.

Download the file `lab5_circuit1.ino` (as well as the LCD library files). Connect an LCD screen, being sure to properly configure the corresponding library files.

Connect the keypad encoder PCB to the Arduino. Output `D` from the encoder is the MSB of the result, and will connect to pin D7. Output `C` from the encoder connects to pin D6, output `B` connects to pin D5. Output `A` is the LSB and will connect to pin D4. The `DATA` signal will connect to pin D2. A full schematic of the keypad PCB is included in Appendix B.

The code will display the values of `a` and `b` and their sum on an LCD screen. Pressing a button on a keypad stores that number in the keypad memory and will cycle between updating variable `a` and variable `b`. Any update of these variables will also update variable `c`, which is the sum.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code. **DO NOT TAKE APART THIS CIRCUIT WHEN YOU ARE FINISHED!**



Program Memory: _____ bytes

Data Memory: _____ bytes

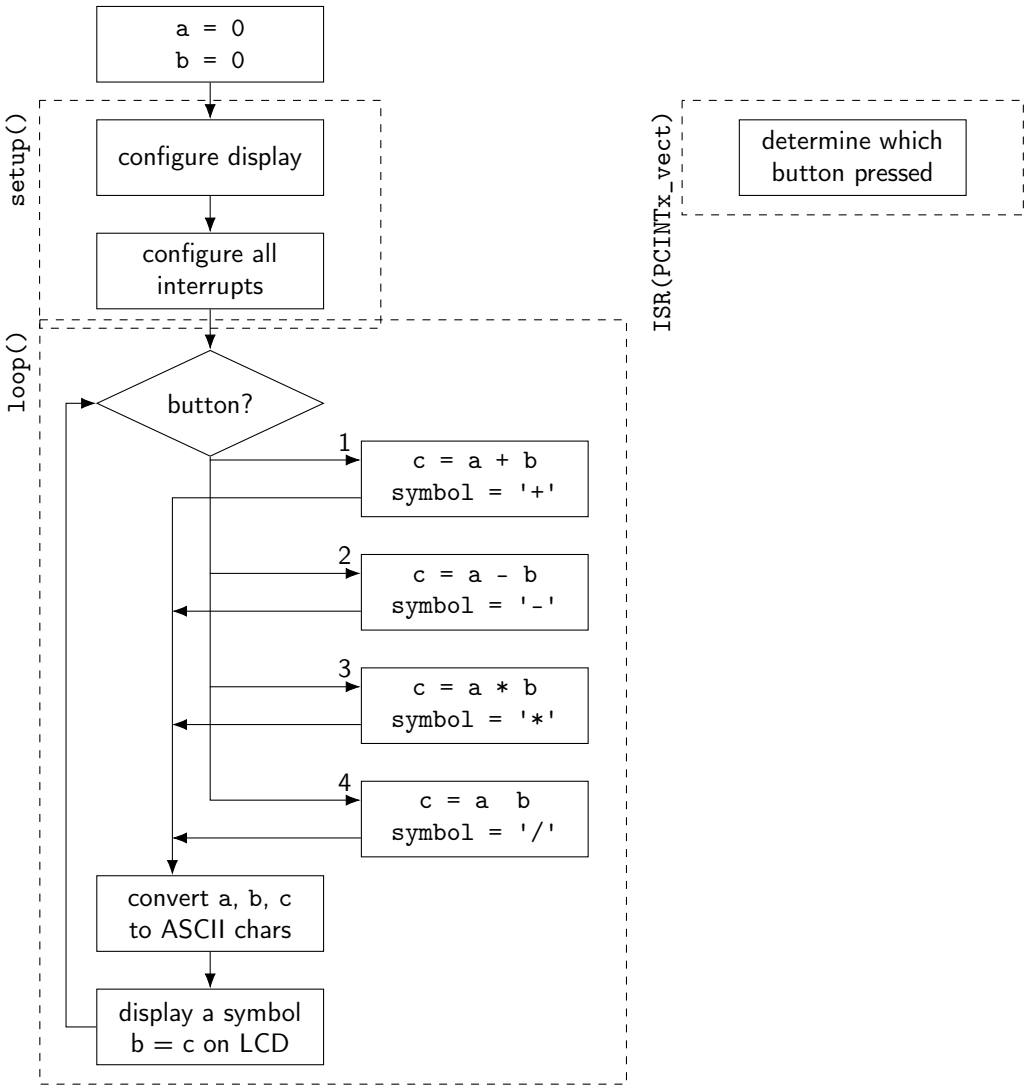
Instructor Stamp: _____

Circuit II: Keypad calculator

> This circuit will be capable of adding, subtracting, multiplying, and dividing two numbers. Both values being operated on will be displayed on an LCD screen along with the result of the operation.

Add four debounced pushbuttons to your circuit. Connect them to any available pins. Pressing any of these pushbuttons will trigger a pin-change interrupt to change the operation of the calculation being performed. The pushbuttons will select between: `c = a + b`, `c = a - b`, `c = a * b`, and `c = a / b` (taking care to avoid `b = 0` in division operations). Keypad presses will continue to trigger an external interrupt on pin D2 as with Circuit I (this is not shown on the flowchart).

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

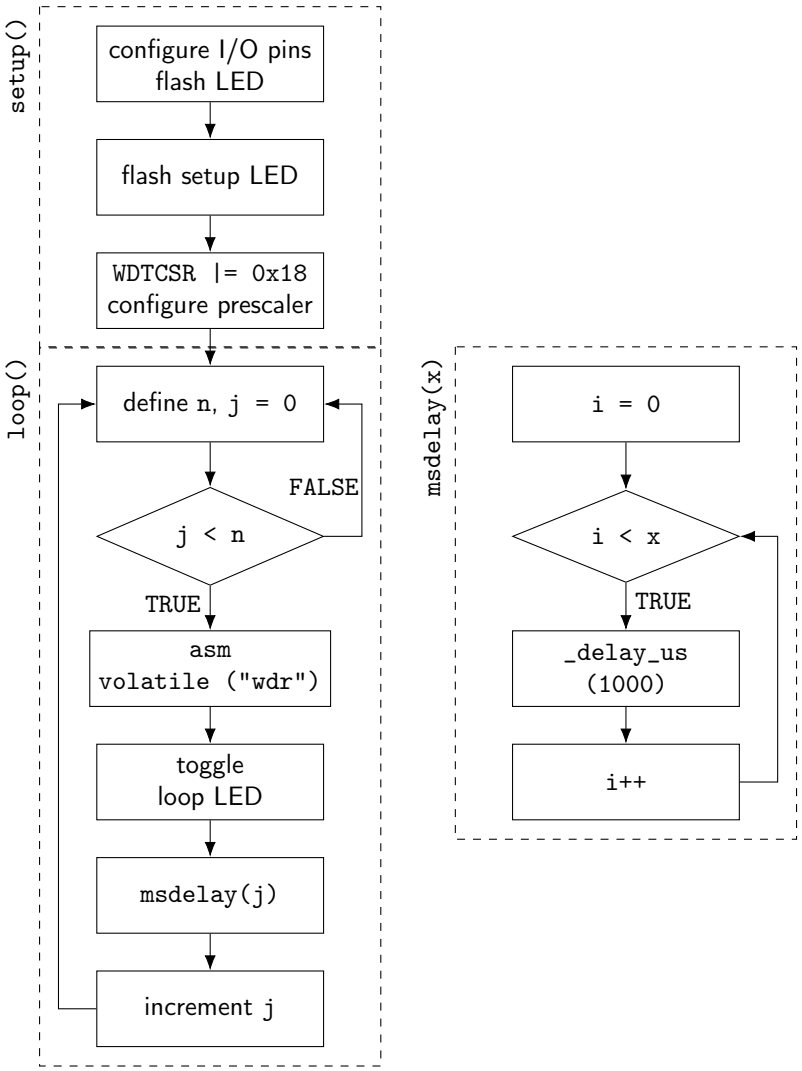
Instructor Stamp: _____

Circuit III: Watchdog timer reset

This circuit will flash an LED in the setup function. The loop function will flash a different LED for increasing intervals of time until the WDT is triggered and a reboot occurs.

Connect two different colored LEDs to different pins on the Arduino. Write code that sets up the WDT to expire after a few seconds. In the `setup()` function, flash one of the LEDs for at least 250 ms. In the `loop()` function, use a for loop that delays at increasing time intervals. At the beginning of the for loop, call the function `asm volatile ("wdr")` and have the other LED light for the duration of the delay, and then turn off for the same amount of time. After several iterations of the `for` loop, the code should not be able to complete both delays before the WDT triggers a reset. (A reset occurs if the setup LED lights).

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 5 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- For Circuit I and Circuit II, describe **exactly** how/if you could have programmed the circuits without using interrupts. (Do not just say "yes we could have but it would have been hard." Be specific about how it could be done.)
- Explain the benefits of using hardware interrupts over continuous polling.
- For Circuit II, explain how you were able to determine which pushbutton was pressed using pin change interrupts. Copy/paste the relevant C code into this report.
- For Circuit II, explain how you dealt with division by zero. Copy/paste the relevant C code into this report.

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 6

Carefully read the entirety of Lab 6, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Fill out the Table 6.1 for timer/counter 0 (an 8-bit timer) and timer/counter 1 (a 16-bit timer) operating in normal mode. For each given prescaler (N), how much time will it take for the timer/counter to overflow (i.e. reach the value of 2^n)? This will serve as a useful guide to help you determine the longest delay that is possible for each timer/counter and with each prescaler. Be sure to include units in your answers!

N	TCNT0	TCNT1
1		
8		
64		
256		
1024		

Table 6.1: Longest possible delays using TCNT0 and TCNT1 in normal mode.

2. Fill out the Table 6.2 for timer/counter 2 (an 8-bit timer) operating in normal mode. For each given prescaler (N), how much time will it take for the timer/counter to overflow (i.e. reach the value of 2^n)? This will serve as a useful guide to help you determine the longest delay that is possible for TCNT2 and with each prescaler. Be sure to include units in your answers!

N	TCNT2
1	
8	
32	
64	
128	
256	
1024	

Table 6.2: Longest possible delays using TCNT2 in normal mode.

3. Fill out the Table 6.3 for all timer/counters. For each given prescaler (N), what is the clock period? This will serve as a useful guide to help you determine the shortest delay that is possible for each timer/counter and with each prescaler. Be sure to include units in your answers!

N	$T_{CLK,I/O}$
1	
8	
32	
64	
128	
256	
1024	

Table 6.3: Shortest possible delays using each timer/counter.

4. In Circuit 1, is it necessary to use the ADC interrupt? Justify your answer.

5. Based on your answer to the previous question, what value should be stored in `ADCSRA`?

6. Explain the difference between normal mode and CTC mode for the timer/counter units.

7. In Circuit 2, what is the largest value that `x` can be? Given that largest value, what datatype makes the most sense to use?

Lab 6: Timer/Counters and Timed Interrupts

In this lab, the ATmega328P timer/counters will be used to generate timed interrupts. **C Concepts:** `long` to `char` conversion **AVR Concepts:** I/O clock, prescalers, 8-bit timer/counter 0 and registers `TCCR0A`, `TCCR0B`, `TCNT0`, `OCR0A`, `OCR0B`, and `TIMSK0`; 16-bit timer/counter 1 and registers `TCCR1A`, `TCCR1B`, `TCNT1`, `OCR1A`, `OCR1B`, and `TIMSK1`; 8-bit timer/counter 2 and registers `TCCR2A`, `TCCR2B`, `TCNT2`, `OCR2A`, `OCR2B`, and `TIMSK2`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

6.1 ATmega328P timer/counters

There are three timer/counters on the ATmega328P microcontroller: 8-bit timer/counter 0 (TCNT0), 16-bit timer/counter 1 (TCNT1), and 8-bit timer/counter 2 (TCNT2). A counter is a device that has an incrementing value. A timer refers to a counter that is synchronous (i.e. values increment at some interval of the microcontroller clock frequency).

The number of bits in the counter refers to how high it can count, and for timers relates to how much time can elapse between timed events. An n bit counter can count from 0 to a maximum value of $2^n - 1$. In synchronous operation, the value in the timer register increments every time the I/O clock ticks. The counter is able to count from zero to its top value in T seconds, described by Equation 6.1, where N is the value of the prescaler (which divides the frequency), `TOP` is the highest value that the timer can count to (see below), and $f_{CLK,I/O}$ is the frequency of the I/O clock.

$$T = \frac{N \times (\texttt{TOP} + 1)}{f_{CLK,I/O}} \tag{6.1}$$

The I/O clock frequency of the Arduino Uno is 16 MHz. If using the ATmega328P without the Arduino platform, the internal clock is specified by the fuse bits (configuration bits that will be discussed in Lab 10).

6.1.1 Timer/counter definitions

The following terms are frequently used in describing timer/counter operation and are important to know.

- `BOTTOM` – the counter reaches `BOTTOM` when it becomes 0x0000
- `MAX` – the counter reaches `MAX` when it becomes 0xFF (in an 8-bit counter) or 0xFFFF (in a 16-bit counter)
- `TOP` – the counter reaches `TOP` when it becomes equal to the highest value in the count sequence, which can be either `MAX`, a set value, or a value stored in a register

Timer/counters can operate under different modes of operation, and can be used to create waveforms by changing the value of an output pin, trigger interrupts, or do both simultaneously.

6.1.2 Creating waveforms

A waveform can be created on a timer/counter output pin by configuring the timer/counter registers to set, clear, or toggle the pin upon reaching the `TOP` value of the register. This configuration was used in the activity in which you determined the frequency of the Arduino Uno microcontroller. This configuration will also be used in Lab 7 with pulse-width modulation. This will not be used in this lab. Therefore, the

timer/counter output pins will not be used for that function and will be disconnected for each circuit in this lab.

6.1.3 Triggering interrupts

An interrupt can be triggered by a timer/counter based on different criteria. It is very useful to have scheduled interrupts that occur at fixed intervals; these can be accomplished with timer/counter interrupts. Appendix C details all of the different criteria in which a timer/counter interrupt can be invoked. These criteria are defined below.

- **Compare match A** – this interrupt is invoked when the contents of the timer/counter register are equal to the contents of `OCRnA`
- **Compare match B** – this interrupt is invoked when the contents of the timer/counter register are equal to the contents of `OCRnB`
- **Overflow** – this interrupt is invoked when the contents of the timer/counter register are equal to `MAX`
- **Capture event** (timer/counter 1 only) – this interrupt is invoked when the input capture unit senses either a rising-edge or falling-edge, as configured (this functionality will be used in Lab 12)

6.2 Modes of operation

Regardless of the desired functionality of the timer/counter (to create a waveform, trigger interrupts, or both), timer/counters have several different modes of operation that dictate how they count. In this lab, both normal and clear timer on compare match (CTC) modes will be used. All of the other modalities concern pulse-width modulation (PWM), which will be discussed in Lab 7.

All non-PWM modes of operation feature constant time intervals between timer/counter increments, as shown in Figure 6.1.

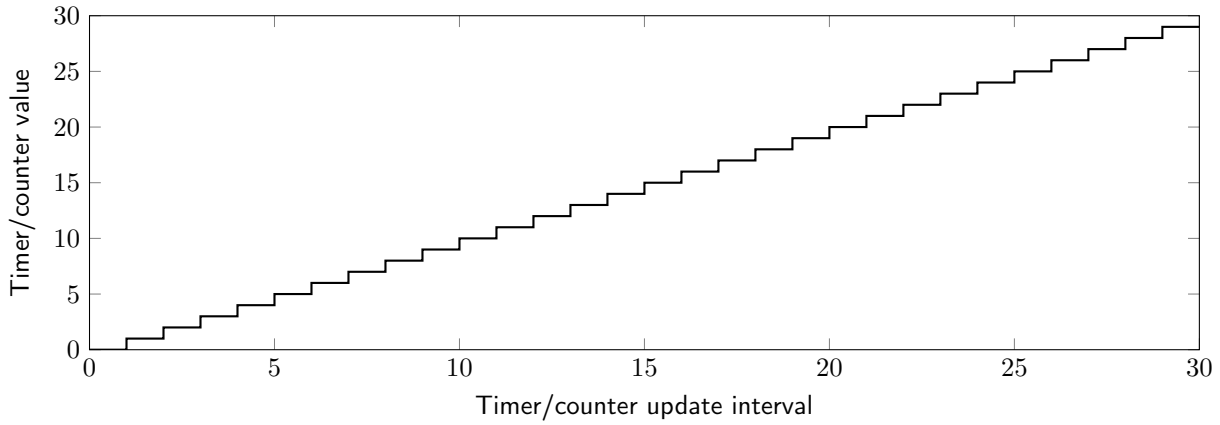


Figure 6.1: The value in the timer/counter increases every time the update time interval occurs.

The time interval between increments (t) is specified by the frequency of the microcontroller I/O clock $f_{CLK,I/O}$ and the prescaler of the timer/counter as defined by Equation 6.2.

$$t = \frac{N}{f_{CLK,I/O}} \tag{6.2}$$

Once the counter reaches `TOP` the value will jump to `BOTTOM` again and increment, repeating the process over and over.

6.2.1 Normal mode

The timer/counter counts up (incrementally) from `BOTTOM` to `MAX` and then restarts again from `BOTTOM`. The amount of time that elapses between `BOTTOM` and `MAX` is described by Equation 6.3.

$$T = \frac{N \times (\texttt{MAX} + 1)}{f_{CLK,I/O}} \tag{6.3}$$

This operation is depicted in Figure 6.2. The overflow flag (`TOVn`) is set every time the value in the timer/-counter reaches `MAX`. Compare match interrupts can be utilized in normal mode as well (this is not shown on Figure 6.2). The values of the output compare registers can be modified as the code executes if interrupts need to occur at differing time intervals.

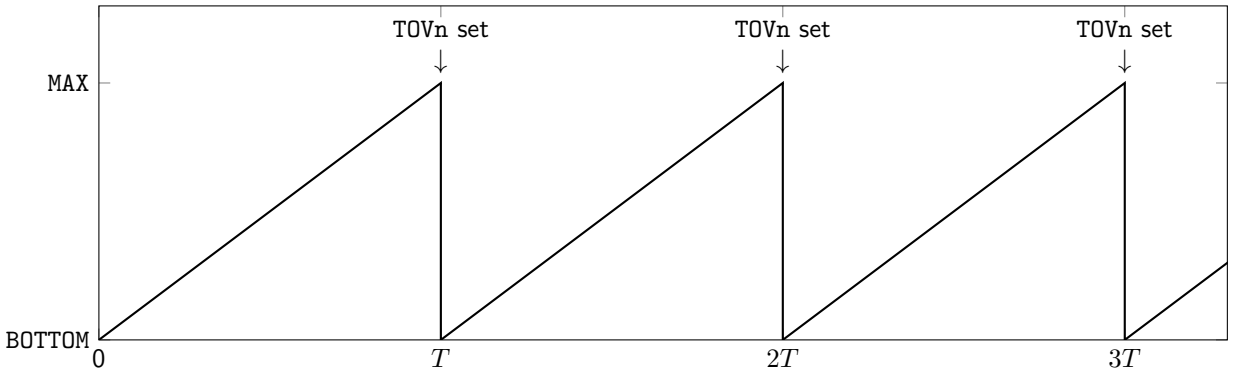


Figure 6.2: In normal mode, the timer/counter value increments from `BOTTOM` to `MAX` periodically.

It is possible to change the period as the code executes by modifying the prescaler. It is important to note that this might alter the exact timing of overflow interrupts at the time periods when the prescaler is changed.

Because `MAX` is a fixed number in a timer/counter (either 0xFF or 0xFFFF), the amount of time elapsed between events in normal mode can only be modified by changing the prescaler. For more specific timing intervals, normal mode is not desired; CTC mode should be used instead.

Normal mode is useful for counting the amount of time that elapses between events (by counting how many increments of a counter occur between events), or for creating a waveform or triggering an interrupt at fixed time intervals.

6.2.2 Clear timer on compare match (CTC) mode

In CTC mode, the timer/counter counts up (incrementally) from `BOTTOM` to `TOP`, where `TOP` is the value defined in the `OCRnA` register. (When using timer/counter 1, the input capture register `ICR1` can also be used to define `TOP` in CTC mode.) The amount of time that elapses between `BOTTOM` and `TOP` is described by Equation 6.4.

$$T = \frac{N \times (\texttt{TOP} + 1)}{f_{CLK,I/O}} \tag{6.4}$$

Counting to a `TOP` value of `OCRnA` is depicted in Figure 6.3. The compare match A flag (`OCFnA`) is set every time the value in the timer/counter reaches `OCRnA`. If `OCRnB` is set at a value between `BOTTOM` and `OCRnA`, then the compare match B interrupts can be utilized as well. The values of `OCRnB` can be modified as the code executes if compare match B interrupts need to occur at differing time intervals. Because the value of `MAX` is typically never achieved in CTC mode, **overflow interrupts cannot be used in CTC mode**.

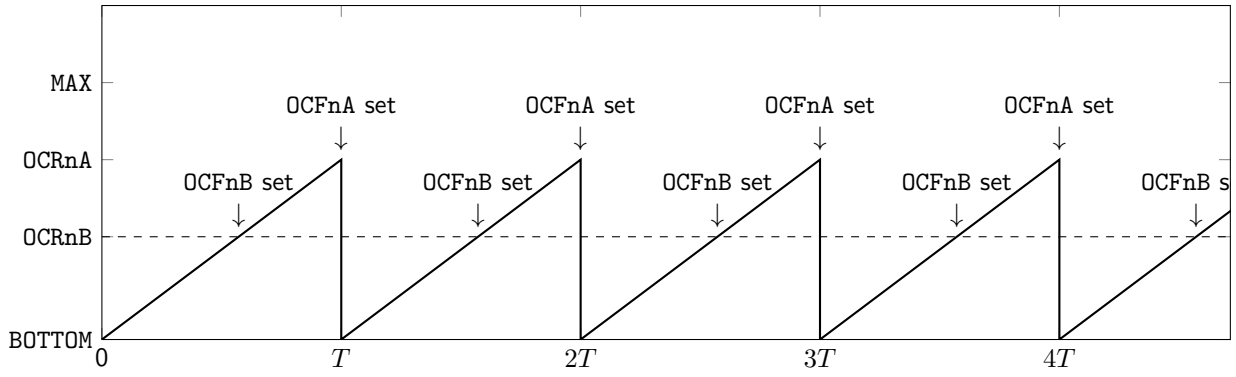


Figure 6.3: In CTC mode, the timer/counter value increments from `BOTTOM` to `OCRnA` (`TOP`) periodically.

Note that if `ICR1` is used to define `TOP` using timer/counter 1, the interrupt flag `ICF1` will be set upon reaching the value stored in `ICR1`. This corresponds to the timer/counter 1 input capture flag and timer/counter 1 capture event interrupt.

It is possible to change the period as the code executes by modifying either the prescaler or the value of `TOP`. It is important to note that this might alter the exact timing of compare match interrupts at the time periods when these values are changed.

CTC mode provides maximum flexibility over the possible value of T. This is because both N (the prescaler) and `TOP` can be configured. For more information about utilizing timer/counters in normal and CTC mode, refer to the class textbook.

6.3 8-bit timer/counter 0 (TCNT0)

Timer/counter 0 (TCNT0) has 8 bits of resolution with two independent output compare units. There are several registers that control its operation. For more detailed information about the timer/counter registers, refer to Appendix A.

- `TCCR0A` – **Timer/Counter 0 Control Register A**: This register configures the mode of operation of TCNT0, and additionally sets two of the three waveform generation mode bits.
- `TCCR0B` – **Timer/Counter 0 Control Register B**: This register sets one of the three waveform generation mode bits for TCNT0, and additionally selects the clock source and prescaler.
- `TCNT0` – **Timer/Counter 0 Register**: This register contains the current value of timer/counter 0.
- `OCR0A` – **Timer/Counter 0 Output Compare Register A**: This register contains an 8-bit value that is continuously compared with the value in `TCNT0`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC0A` pin.
- `OCR0B` – **Timer/Counter 0 Output Compare Register B**: This register contains an 8-bit value that is continuously compared with the value in `TCNT0`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC0B` pin.
- `TIMSK0` – **Timer/Counter 0 Interrupt Mask Register**: This register configures TCNT0 interrupts.

6.4 16-bit timer/counter 1 (TCNT1)

Timer/counter 1 (TCNT1) has 16 bits of resolution with two independent output compare units. There are several registers that control its operation. For more detailed information about the timer/counter registers, refer to Appendix A.

- `TCCR1A` – **Timer/Counter 1 Control Register A**: This register configures the modality of TCNT1, and additionally sets two of the three waveform generation mode bits.

- `TCCR1B` – **Timer/Counter 1 Control Register B**: This register sets two of the four waveform generation mode bits for TCNT1, and additionally selects the clock source and prescaler.
- `TCNT1H` & `TCNT1L` – **Timer/Counter 1 Register**: These registers contain the current value of timer/counter 1. Two registers are required to store the value because timer/counter 1 is a 16-bit counter.
- `OCR1AH` & `OCR1AL` – **Timer/Counter 1 Output Compare Register A**: These two registers contain a 16-bit value that is continuously compared with the value in `TCNT1H` and `TCNT1L`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC1A` pin.
- `OCR1BH` & `OCR1BL` – **Timer/Counter 1 Output Compare Register B**: These two registers contain a 16-bit value that is continuously compared with the value in `TCNT1H` and `TCNT1L`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC1B` pin.
- `ICR1H` & `ICR1L` – **Input Capture Register 1**: These two registers contain a 16-bit value that is associated with the input capture unit. The value in this register can be used to define `TOP` in CTC mode. (Note that these registers can only be written to when timer/counter 1 is configured with a `TOP` value of `ICR1`. Otherwise the registers are read-only.)
- `TIMSK1` – **Timer/Counter 1 Interrupt Mask Register**: This register configures TCNT1 interrupts.

6.5 Reading and writing 16-bit registers

With the exception of the ADC data registers (due to the different storage possibilities depending on the value of `ADLAR`), all 16-bit registers can be directly addressed in C using the register name. For example, a value of 50,000 can be written to timer/counter 1 output compare register A using the code `OCR1A = 50000;`.

If it is necessary to read or write 16-bit registers a single byte at a time, the high byte must be written before the low byte. To read a 16-bit register one byte at a time, the low byte must be read before the high byte.

6.6 8-bit timer/counter 2 (TCNT2)

Timer/counter 2 (TCNT2) has 8 bits of resolution with two independent output compare units. It has the capability of being operated asynchronously, but that option will not be used in this lab. There are several registers that control its operation. For more detailed information about the timer/counter registers, refer to Appendix A.

- `TCCR2A` – **Timer/Counter 2 Control Register A**: This register configures the modality of TCNT2, and additionally sets two of the three waveform generation mode bits.
- `TCCR2B` – **Timer/Counter 2 Control Register B**: This register sets one of the three waveform generation mode bits for TCNT2, and additionally selects the clock source and prescaler.
- `TCNT2` – **Timer/Counter 2 Register**: This register contains the current value of timer/counter 2.
- `OCR2A` – **Timer/Counter 2 Output Compare Register A**: This register contains an 8-bit value that is continuously compared with the value in `TCNT2`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC2A` pin.
- `OCR2B` – **Timer/Counter 2 Output Compare Register B**: This register contains an 8-bit value that is continuously compared with the value in `TCNT2`. A match can be used to generate an output compare interrupt, or to generate a waveform output on the `OC2B` pin.
- `TIMSK2` – **Timer/Counter 2 Interrupt Mask Register**: This register configures TCNT2 interrupts.

6.7 Datatype conversion: long integer to ASCII

Similar to the function `itoa` that was used in Lab 3 to convert an integer value to an ASCII string, a `long` or `unsigned long` can be converted to ASCII using the `ltoa` function. Note that the variable `charBuffer`

must have a sufficient number of elements to store the largest possible value of the integer number in the desired number system, plus a sign (for negative numbers), plus a terminating null character.

```
// create an array that will store each ASCII character
char charBuffer[n];

// convert value, in the given base, into the variable charBuffer[]
ltoa(long value, char charBuffer, int base);
```

6.8 Long arithmetic overflow

As noted in Lab 4, overflow is possible while calculating the intermediate result of an arithmetic operation. When doing multiplication with a result that fits into the `int` datatype, appending a literal with `L` will promote that literal to a `long` and ensure that the intermediate result does not overflow. Now that you will be doing multiplication with a result that fits into the `long` datatype, the `L` may no longer be sufficient. In this case, append a literal with `LL` to ensure that the intermediate result does not overflow. This will be necessary in Circuit II.

Circuit I: Temperature sensor with timed interrupt

This circuit will display the temperature on a display, updating the value using timed interrupts.

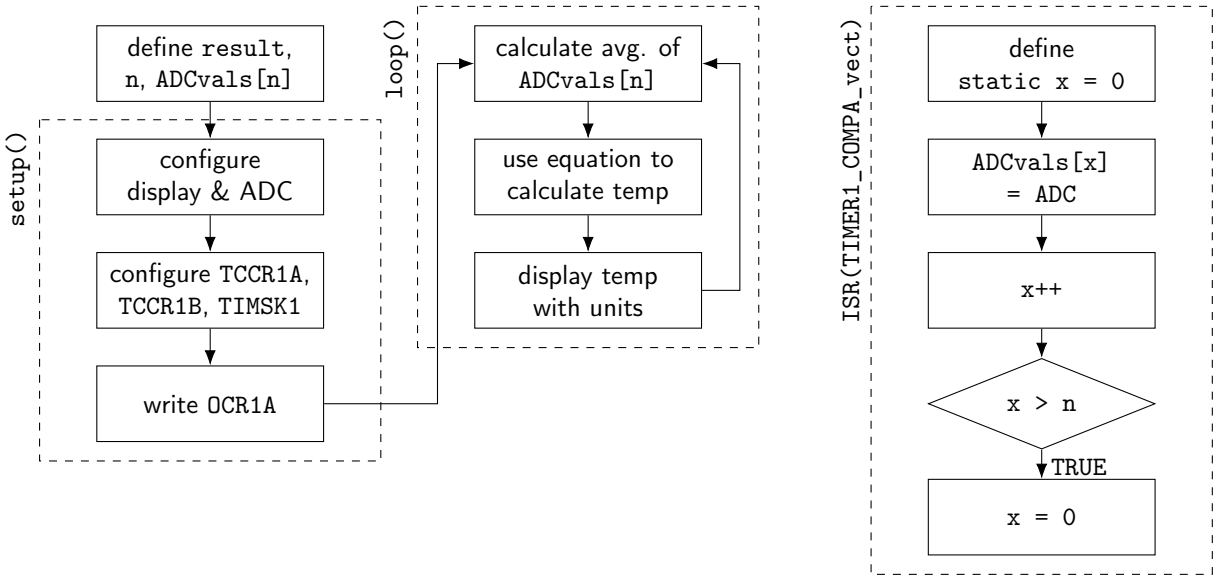
With the display of your choice, display the average temperature value of a TMP36 temperature sensor (with bypass capacitor and 47 kΩ pull-up resistor) with tenth's place precision. **DO NOT USE FLOATING-POINT NUMBERS OR ARITHMETIC!** Use a timed interrupt with TCNT1 and `OCR1A` to update the temperature array at given intervals. In order to configure your timed interrupts, you first need to decide how long you would like to wait between sensor updates. (Try values greater than 250 ms.) Record this value below.

Use Equation 6.4 to determine the smallest possible prescaler (N) value that can be used, as well as the corresponding value for the `OCR1A` register. Record these values below.

You will need to determine a value for n. Use a value smaller than what you used in Lab 4, and record it below. You will need to justify this value in your lab report, so try multiple values and take notes on the responsiveness and data memory requirements of your code.

Code delay can be used to minimize LCD flicker, or to write each digit in a MUX display, but each of those delays should be as short as possible. Delays will not be accepted for any other reason (including temperature value fluctuation; if the temperature value is fluctuating unacceptably, increase n, the timing of your delays in timer/counter 1, or both)!

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Calculating time between events

This circuit will display the amount of time (in ms) that has elapsed between button presses.

The flowchart for this circuit is given on page 81. Use the LCD screen. Ensure that you have downloaded and properly configured the LCD library files. Use a debounced pushbutton on either pin D2 or D3 to trigger external interrupts. Configure TCNT2 to have pins `OC2A` and `OC2B` disconnected, normal mode, prescaler of 256, and enable interrupts on timer overflow.

The variable `ticks` stores the number of ticks that have elapsed since the start of the code execution. An overflow interrupt will be used to add 256 to this value; every time TCNT2 overflows it indicates that TCNT2 has increased from 0–255. A circular buffer consisting of two entries (`tks[2]`) will contain the number of counter ticks that have elapsed between the start of the code execution and pressing the pushbutton. The initial values will both be 0. Every time the pushbutton is pressed, one of the array entries will update (`ticks + TCNT2`) using an external interrupt. Use `unsigned long` datatypes for all of these values. Do not use any floating-point math in this circuit! Your circuit will display the number of milliseconds that have elapsed between button presses.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

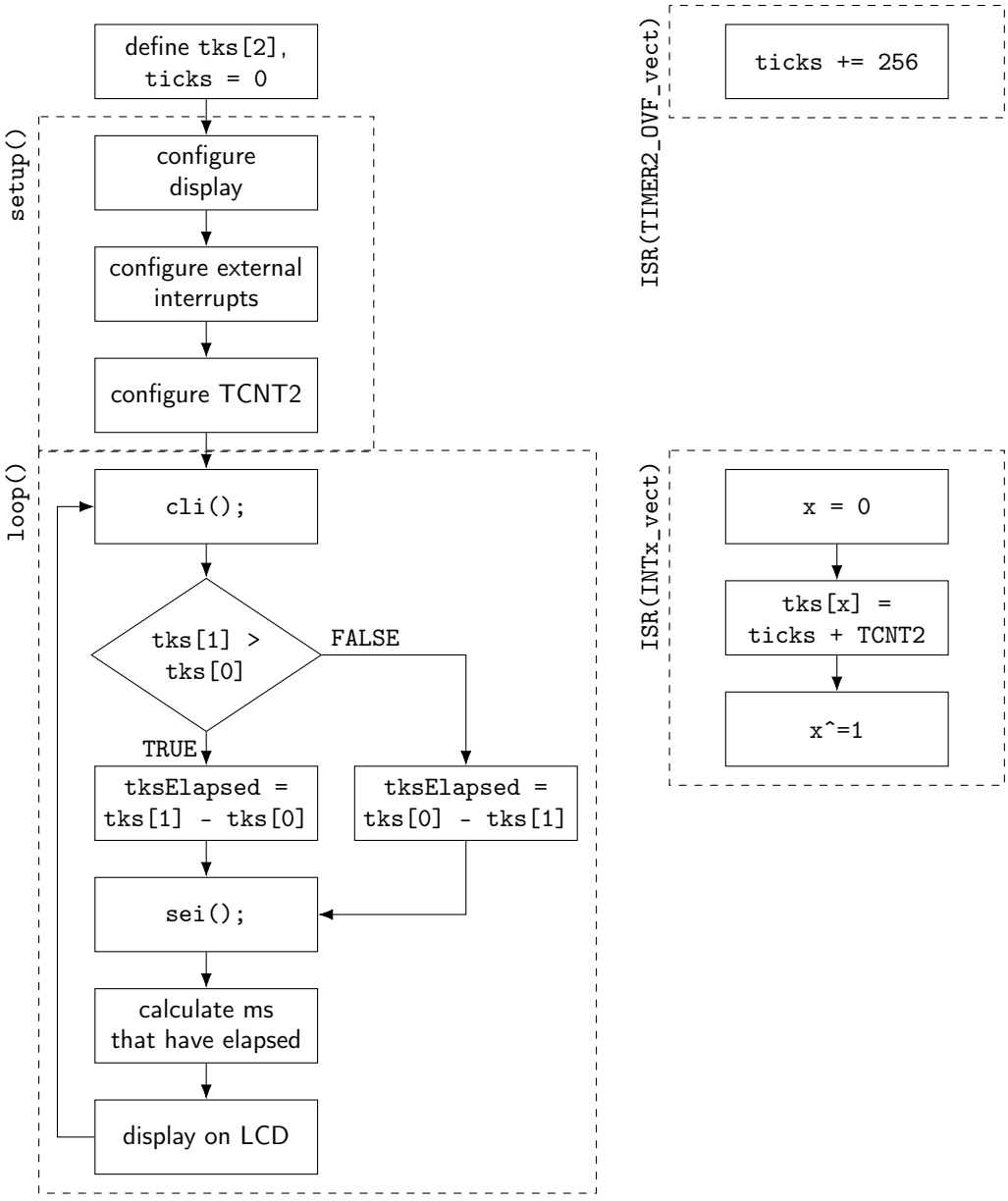


Figure 6.4: Flowchart for Circuit II.

Lab 6 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Why did you need to use CTC mode in Circuit I?
- Record and explain the derivation of the prescaler and `OCR1A` values used in Circuit I.
- In Circuit I, you experimented with different values of n for your rolling temperature average. What value did you use in this lab? Why?
- Why did you need to use normal mode in Circuit II?
- In Circuit II, what is the maximum amount of time between button presses that can be calculated by your design? Explain how you determined this value. What are two different things you could change to extend this span of time?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 7

Carefully read the entirety of Lab 7, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Record the pins that are controlled by each output compare register.
   (a) `OC0A`

   (b) `OC0B`

   (c) `OC1A`

   (d) `OC1B`

   (e) `OC2A`

   (f) `OC2B`

2. Why is a separate power supply used in Circuit III?

3. What special precaution needs to be taken to ensure that the Arduino and external power supply will work together in Circuit III?

4. Why is a flyback diode used in Circuit III?

Lab 7: Pulse-Width Modulation and Motors

In this lab, two different types of motors will be controlled using pulse-width modulation (PWM). PWM is a digital way to create signals that can have varied average voltage levels. The two motors to be used in this lab are servomotors and DC motors. **AVR Concepts:** Fast PWM, phase-correct PWM

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

7.1 Pulse-width modulation (PWM)

Pulse-width modulation (PWM) is a technique to encode analog data into a digital signal. Instead of sending a continuous voltage signal of 0 V or 5 V, voltage is instead turned ON and OFF in timed pulses. PWM may be used for controlling the brightness of an LED, controlling the angle of a servomotor, controlling the speed of a DC motor, or any other application that requires some type of digital control signal. Three key properties of PWM signals are the **duty cycle**, **average voltage**, and **signal frequency**.

An example PWM waveform is shown in Figure 7.1. Shown in the graph are the high period of the wave (T_{HIGH}), the low period of the wave (T_{LOW}), the total period of the wave (T), and the average voltage of the wave ($\overline{V}$).

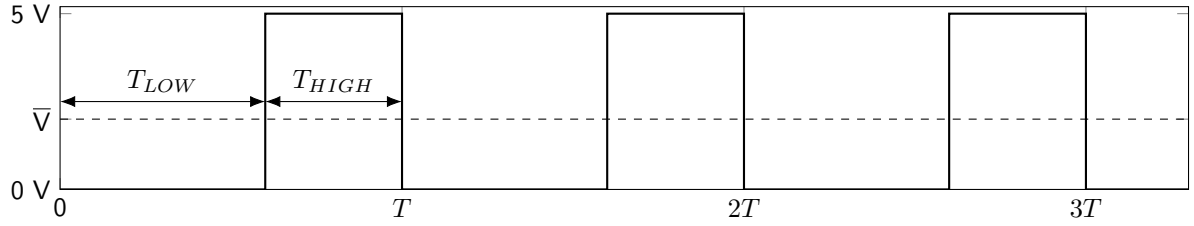


Figure 7.1: An example PWM waveform, showing the high period, low period, total period, and average voltage.

To implement PWM, a timer/counter must be used. There are three different types of PWM modes. Two of those types (fast PWM and phase-correct PWM) will be utilized in this lab.

7.1.1 Duty cycle

The amount of time in which the signal is high (T_{high}) divided by the total period (T) of the signal is known as the duty cycle, defined in Equation 7.1.

$$D = \frac{T_{high}}{T} \tag{7.1}$$

By varying the duty cycle, the effective intensity of a signal can be varied from OFF (0% duty cycle) to ON (100% duty cycle) as the average voltage changes between 0–5 V.

7.1.2 Average voltage

The average voltage is defined by Equation 7.2, where D is the duty cycle, V_{max} is the maximum voltage (usually 5 V in digital circuits), and V_{min} is the minimum voltage (usually 0 V).

$$\overline{V} = DV_{max} + (1 - D)V_{min} \tag{7.2}$$

When $V_{min} = 0$, Equation 7.2 reduces to $\overline{V} = DV_{max}$.

7.1.3 PWM frequency

The frequency of a PWM signal is equal to the number of complete cycles that occur per period of time. Frequency is measured in units of Hz (cycles per second), and is the inverse of period, as defined by Equation 7.3.

$$f = \frac{1}{T} \tag{7.3}$$

PWM frequency must be chosen with careful consideration to the device with which it will be used. Every piece of hardware has its own characteristics that must be taken into account. The three devices that will be used in this lab are an LED, a servomotor, and a DC motor.

For an LED, the PWM frequency must be fast enough that it is not seen as a visible flicker to the human eye. This puts a lower range on the PWM frequency somewhere around 300 Hz for applications where the viewer and LED are stationary. For moving LEDs (or in cases where people will move past a stationary LED), a frequency of 1 kHz should be considered instead. The upper limit on PWM frequency is given by the architecture of the diode itself. At frequencies that are too high, the LED never fully turns on between PWM cycles. Therefore, it is recommended to use a frequency of 1 kHz when using LEDs.

Most servomotors operate at a frequency of 50 Hz. While this is a fairly typical value, it may vary, so the datasheet should be consulted before deciding on an operating frequency.

DC motors require a PWM frequency that is high enough not to cause the motor to jerk between pulses, and it should be low enough to avoid causing the motor to be unable to fully turn on and off in between pulses. In addition, DC motors should use a PWM frequency outside of the range of human hearing (which is 20–20,000 Hz) so that they don't cause audible noise while running. Therefore, a frequency of approximately 25 kHz is recommended.

7.2 Controlling PWM frequency with timer/counters

As discussed in Lab 6, there are generally two things that can be done timer/counters: generate timed interrupts or generate periodic waveforms (or both). Lab 6 focused on generating interrupts. Interrupts can be used with PWM. To do this, the corresponding interrupt mask register will need to be enabled.

In this lab, instead of generating interrupts, periodic waveforms will be generated. Each timer/counter is capable of generating two independent waveforms on their output compare match pins. Appendix D has a list of all of the alternate pin functions, including the output compare match pins.

7.2.1 Fast PWM frequency

In fast PWM mode, the timer/counter counts from `BOTTOM` to `TOP` and then jumps directly back to `BOTTOM`. Then this cycle repeats. This is depicted in Figure 7.2.

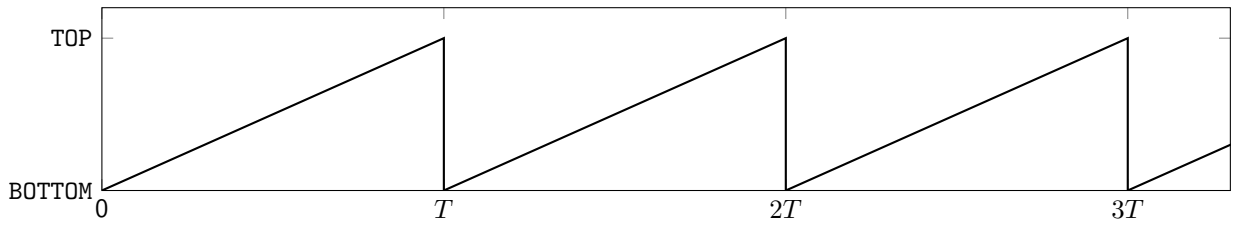


Figure 7.2: The value of a timer/counter used in fast PWM increments from `BOTTOM` to `TOP` and then jumps back to `BOTTOM` again.

`TOP` can be set to the following values.

- `MAX` – The counter can count to the resolution of the counter. (This is only available on timer/counters 0 and 2).
- 8-bit, 9-bit, 10-bit mode – Timer/counter 1 can be used in 8-bit, 9-bit, or 10-bit mode.
- `OCRnA` – The counter can count to a value stored in the output compare register.
- `ICR1` – Timer/counter 1 can count to the value stored in the input capture register. (The `ICR1` register can only be written to if it is defined as `TOP`. Otherwise, it is a read-only register.)

The first two possibilities for the value of `TOP` are similar to normal mode in that the flexibility of the frequency is limited by the resolution of the counter and can only be altered by varying the prescaler. The last two options are similar to CTC mode in that the frequency can be tuned using both the prescaler and the output compare register (or `ICR1`) value. In any case, the period of the fast PWM signal will be defined by Equation 7.4, where $f_{CLK,I/O}$ is the frequency of the I/O clock, N is the value of the prescaler, and `TOP` is the largest value that the timer/counter will count to before resetting to zero.

$$T_{fast} = \frac{N \times (\texttt{TOP} + 1)}{f_{CLK,I/O}} \tag{7.4}$$

7.2.2 Phase-correct PWM frequency

Using phase-correct PWM, the timer/counter counts from `BOTTOM` to `TOP` and then decrements (rather than jumps) to `BOTTOM` again. This is depicted in Figure 7.3.

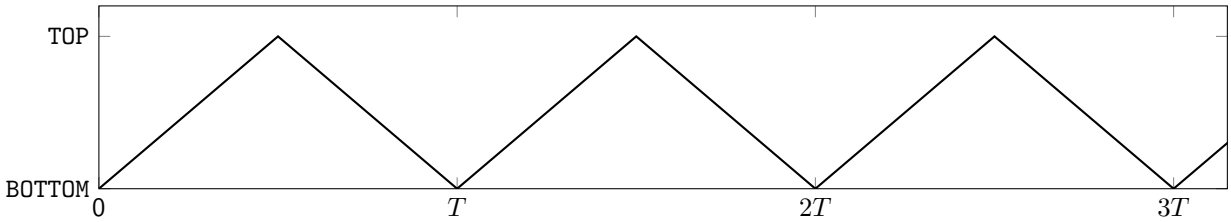


Figure 7.3: The value of a timer/counter used in phase-correct PWM increments from `BOTTOM` to `TOP` and then decrements back to `BOTTOM`.

The value of `TOP` is determined in the same manner as with fast PWM. The period obtainable (defined by Equation 7.5) is now nominally double that of fast PWM mode (hence it is "not fast!"), giving twice the resolution of fast PWM.

$$T_{phase-correct} = \frac{2 \times N \times \texttt{TOP}}{f_{CLK,I/O}} \tag{7.5}$$

7.3 Controlling PWM duty cycle with timer/counters

The duty cycle of a PWM signal is controlled by using an output compare register. This lab will focus on non-inverting PWM modes. As the value of the timer/counter increases from `BOTTOM`, at some point it will reach the value that is saved in the output compare register (as long as that value is lower than `TOP`, otherwise a compare match will never occur). At that point, the signal in the corresponding output compare pin will become LOW.

In fast PWM mode, the signal will become HIGH again when the timer/counter overflows from `TOP` to `BOTTOM`. Figure 7.4 depicts a fast PWM waveform in non-inverting mode.

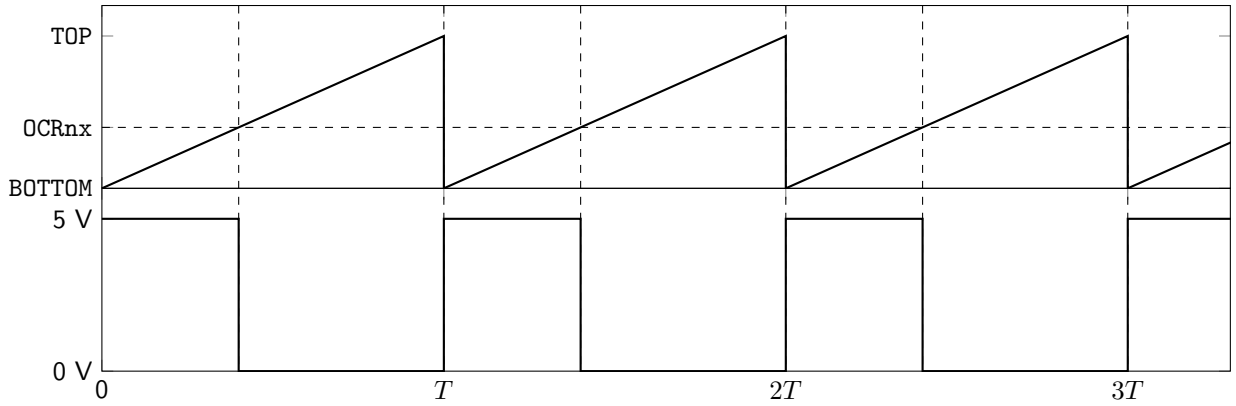


Figure 7.4: Fast PWM. The value stored in the timer/counter (top) will cause a PWM waveform on the output compare pin (bottom).

In phase-correct PWM mode, the signal will become HIGH again when the timer/counter decrements through the value in the output compare register again. Figure 7.5 depicts a phase-correct PWM waveform in non-inverting mode.

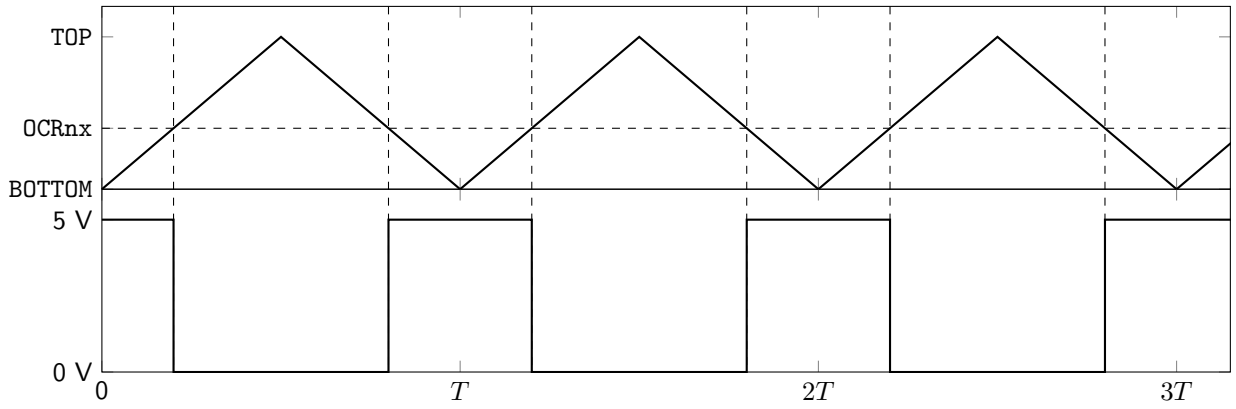


Figure 7.5: Phase-correct PWM. The value stored in the timer/counter (top) will cause a PWM waveform on the output compare pin (bottom).

For additional information, PWM waveforms are explained in more thorough detail in the class textbook.

7.3.1 Calculating duty cycle

How can the duty cycle be calculated? And which output compare registers should be used to obtain the PWM signal?

If using `MAX`; 8-bit, 9-bit, or 10-bit mode (with timer/counter 1); or `ICP1` (with timer/counter 1) as value of `TOP`, then it is possible to create two PWM signals (that have the same frequency) with duty cycles controlled by either `OCRnA` (in which case the output waveform will be available on `OCnA`) or `OCRnB` (in which case the output waveform will be available on `OCnB`). The equation for duty cycle in this case is defined by Equation 7.6.

$$D = \frac{\texttt{OCRnx}}{\texttt{TOP}} \tag{7.6}$$

When using `OCRnA` as `TOP`, it would make no sense to use the output compare A register to set the duty cycle (as it would always be 100%!). In this case, the duty cycle is controlled with `OCRnB`, and the output

waveform is available on the OCnB pin. The equation for duty cycle in this case is defined by Equation 7.7.

$$D = \frac{\texttt{OCRnB}}{\texttt{OCRnA}} \tag{7.7}$$

Note that this is only valid when OCRnB is less than or equal to OCRnA. If OCRnB is greater than OCRnA, then the duty cycle will be 100%. More thorough detail on duty cycle, timing, and issues that can occur with pulse-width modulation are explained in the class textbook.

7.4 Timer/counter registers for PWM operation

All of the registers used in the previous lab will be used in this lab, expect that they will be used in PWM modes rather than in non-PWM modes. Refer to Appendix A for the full functionality of all registers.

7.5 Motors

There are several types of motors, but the two that will be used in this lab are servomotors and DC motors.

7.5.1 Servomotors

Servomotors (sometimes referred to simply as "servos") are DC motors coupled with a position feedback control sensor. This allows them to be positioned more accurately than DC motors. There are three wires in servo motors: power, ground, and control. PWM on the control wire controls not the speed at which the motor turns, but the angle to which the motor will rotate.

The servomotors in this lab are restricted to ≈160° rotation. The servomotor period T is 20,000 μs. The pulse-width (T_{high}) required for rotation to 10° is 600 μs, and the pulse-width required for rotation to 170° is 2,400 μs. A rotation to 90° will be the average of these two values.

To control a single servomotor, the current provided by the Arduino board is sufficient. If many servos are to be used, external power may be required.

7.5.2 DC motors

DC motors use electric currents and magnetic fields to create a deflecting force which turns a rotor. They have two wires: one for power and one for ground. (It does not matter which is which; supplying voltage in an opposing polarity simply reverses the spin direction of the motor.) Once power is supplied to the motor, it will rotate continuously until power is removed. The speed at which the motor turns is controlled using PWM. You can find DC motors in your computer controlling the fan, as well as in many toys.

DC motors can require a significant amount of current, more than the Arduino board can provide. For this reason, an external power supply must be used to provide the motor with a sufficient amount of current. In addition, due to the fact that DC motors provide an inductance to the circuit, a flyback diode must be used to prevent current spikes from damaging lab equipment.

Circuit I: Variable intensity LED

This circuit will have a potentiometer control the brightness of an LED using PWM. As the potentiometer is rotated from one extreme to the next, the LED will go from being completely off to having full brightness.

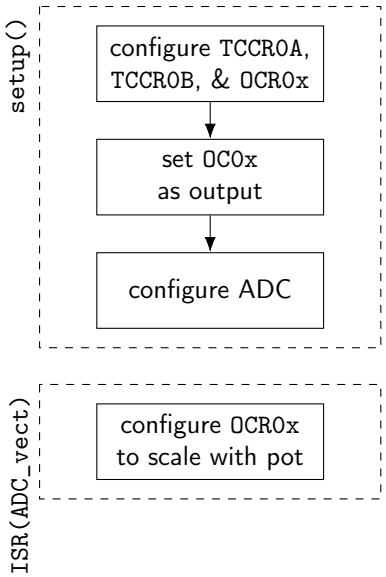
Using information that you determined in the PWM activity, write code that uses fast PWM in non-inverting mode to change the duty cycle of an approximately 1 kHz signal using TCNT0 counting to `MAX`. Because you are counting to `MAX`, you may use either `OC0A` or `OC0B` as the PWM pin. Record the value of your prescaler and your choice of output compare pin in Table 7.1.

Parameter	**Value**
Prescaler	
Output compare pin	

Table 7.1: Timer/counter settings used to control the brightness of an LED.

Use a potentiometer connected to one of the ADC pins to control the LED brightness. As you rotate the potentiometer from one extreme to the other, the LED will either become brighter or dimmer. You will use the ADC in **8-bit mode** to allow you to directly change the duty cycle using the `ADCH` register.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Servomotor control

This circuit will have a potentiometer control the position of a servomotor using PWM. As the potentiometer is rotated from one extreme to the next, the servomotor will move from one of its extremes to the other.

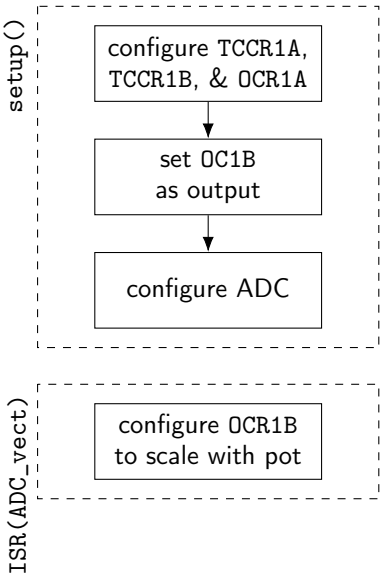
Amend your code from Circuit I to use phase-correct PWM in non-inverting mode with a PWM frequency of 50 Hz using timer/counter 1. You should have determined in an activity what the values of the prescaler and `OCR1A` must be accomplish this. In addition, you should have derived an equation to scale the value from the ADC to obtain the value of `OCR1B`, which must have a period of 0.6 ms when the ADC value is 0, and a period of 2.4 ms when the ADC value is $2^n - 1$. Decide what ADC resolution you'd like to use. Record these values in Table 7.2.

Parameter	**Value**
Prescaler	
`OCR1A`	
`OCR1B`	
ADC resolution	

Table 7.2: Timer/counter settings used to control a servomotor.

Hook up a potentiometer with the wiper connected to one of the analog pins to control the duty cycle. Connect the servomotor so that the signal pin connects to pin D10. As you rotate the potentiometer from one extreme to the other, the servomotor position will update.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: DC motor control

This circuit will have a potentiometer control the speed of a DC motor using PWM. As the potentiometer is rotated from one extreme to the next, the DC motor will go from off (or very slow) to rotating at full speed.

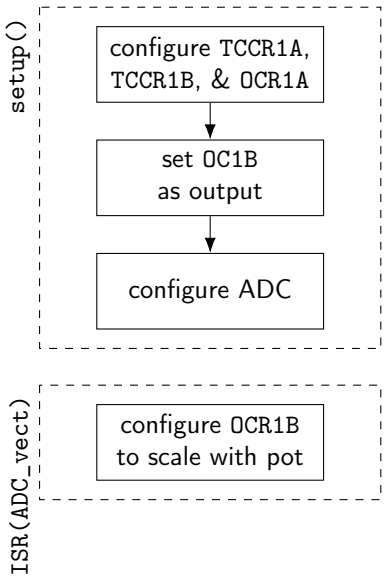
You will use TCNT1 to control the DC motor using phase-correct PWM with a frequency of 25 kHz. You should have determined in an activity what the values of the prescaler and `OCR1A` must be accomplish this. In addition, you should have derived an equation for `OCR1B`, which must have a duty cycle of 20% when the ADC value is 0, and a duty cycle of 100% when the ADC value is $2^n - 1$. Decide what ADC resolution you'd like to use. Record these values in Table 7.3.

Parameter	**Value**
Prescaler	
`OCR1A`	
`OCR1B`	
ADC resolution	

Table 7.3: Timer/counter settings used to control a DC motor.

Wire up a DC motor as shown in Appendix B. VCC and ground supplied to the motor **must** come from an external power supply. **Connect a wire between a ground pin on the Arduino with the ground from the power supply.** This ensures that the Arduino and all externally powered components share a common ground.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 7 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Include equations for `OCRnB` for all three circuits.
- Explain each of the terms in the equations given for `OCRnB`. How were these equations derived?
- What is the difference between fast PWM and phase-correct PWM?
- Why do you use phase-correct (and not fast) PWM with motors?
- What is the fastest possible frequency that you can get with fast PWM? (Consider all possible prescaler values.) What is the slowest? Show your work or justify your answer.
- What is the fastest possible frequency that you can get with phase-correct PWM? (Consider all possible prescaler values.) What is the slowest? Show your work or justify your answer.
- In what subsystems of your Smart Car do you think you'll need to use PWM, and why?
- What are the benefits of using a servomotor rather than a DC motor?
- What are the benefits of using a DC motor rather than a servomotor?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 8

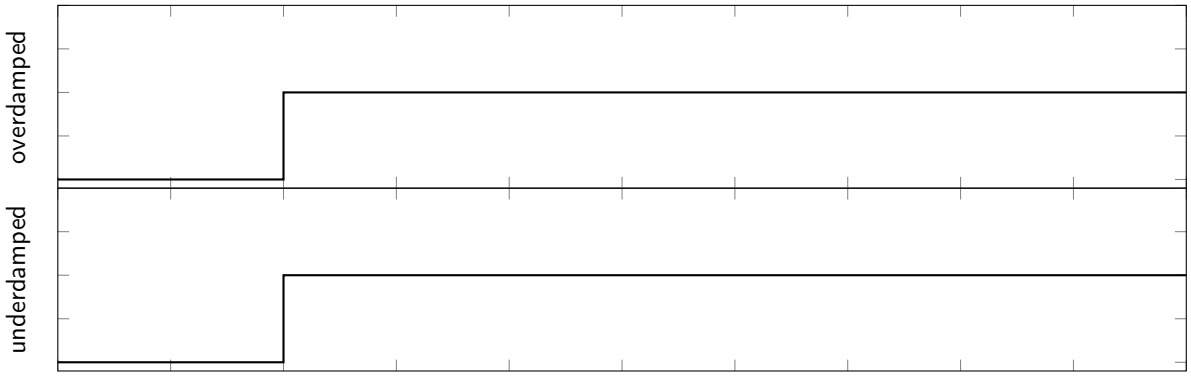
Carefully read the entirety of Lab 8, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Explain why it is necessary to implement feedback control in embedded system applications.

2. What may happen if either K_P or K_I are set to values that are too large?

3. What integer-type of datatype must be used for $e(t)$? Why? Should it be signed or unsigned? Why?

4. Qualitatively sketch the response from an overdamped and underdamped system, using the graphs below. The thick line indicates the ideal step response.



Lab 8: Proportional and Integral Control

Feedback is essential to provide proper device output values. In this lab, the concepts of proportional control and proportional-integral (PI) control will be explored. The intent of this exercise is to provide a means to create feedback mechanisms for the Smart Car steering and propulsion systems.

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

8.1 Closed-loop feedback

Closed-loop feedback is essential to provide proper output values on devices. Consider a refrigerator. There must be a temperature sensor located in the fridge to provide the current value of the temperature. Based on this information, if the temperature is too hot, a compressor will run to cool the interior of the fridge. If the temperature is too cold, the compressor will not run. If there is no feedback control, or if there is improper feedback control, the compressor may run too often or not enough or may even oscillate between these two extremes without providing a proper fridge temperature.

8.2 Proportional control

The most straightforward mechanism for providing feedback is called proportional control. This takes into account the present value of the error and attempts to fix it. An error term ($e(t)$) is calculated by taking the difference between a setpoint value ($V_{setpoint}$) and the current value ($V_{current}$), as defined in Equation 8.1.

$$e(t) = V_{setpoint} - V_{current} \tag{8.1}$$

This error term is then fed back into the control of the device to provide a correction, as defined in Equation 8.2, where K_P is known as the proportionality constant.

$$V_{new} = V_{current} + K_P e(t) \tag{8.2}$$

The proportionality constant K_P is chosen based on the circuit properties and requirements. Values that are too low lead to sluggish, unresponsive feedback (this is known as an overdamped system). Values that are too high become unstable and can oscillate rapidly between values (this is known as an underdamped system). In the worst case, an underdamped system won't ever settle down to a proper value.

8.3 Proportional-integral (PI) control

Including integral control in a feedback system allows the circuit to compensate for past error in the system in addition to compensating for any present error. The error term continues to be defined by Equation 8.1. However, the correction for the device control changes. It is now defined by Equation 8.3, where K_I is known as the integral constant.

$$V_{new} = V_{current} + K_P e(t) + K_I \int_0^t e(\tau)\, d\tau \tag{8.3}$$

In a system that takes samples at discrete time intervals (rather than continuously) a sum, rather than an integral, will describe the correction term. This is defined by Equation 8.4, where τ defines the length of

time over which the past error will be sampled.

$$V_{new} = V_{current} + K_P e(t) + K_I \sum_{t=-\tau}^{0} e(t) \tag{8.4}$$

The value of τ must be chosen with care. If τ is small, processing time and memory requirements will be reduced, but long-term errors will not be corrected. If τ is large, more memory and processing time will be required to make the calculations, but long-term error will be corrected. In this lab, τ will be determined empirically (i.e., many values will be chosen until a good compromise between error-correcting and memory/processing time is made).

Adding an integral constant K_I (which must be less than K_P) helps the feedback system achieve a steady-state value much quicker, based on the fact that it helps compensate for errors that have not yet been cleaned up by the proportionality constant. However, K_I still must be carefully chosen, as a poorly chosen value can still lead to an overdamped or underdamped output response.

8.4 Proportional-integral-derivative (PID) control

A full proportional-integral-derivative (PID) feedback system also takes into account anticipated future values of the error based on taking a derivative of the current error. The PID control equation is defined by Equation 8.5, where K_D is known as the derivative constant.

$$V_{new} = V_{current} + K_P e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \frac{d}{dt} e(t) \tag{8.5}$$

Including a derivative constant makes the feedback mechanism much more sensitive to noise, and is only recommended in cases where noise will be minimum. For that reason, it will not be utilized in this lab.

8.5 Serial plotter

In previous labs, the serial monitor may have been used to obtain diagnostic information about code without having to connect the microcontroller to a separate display. The serial plotter takes data and plots it in a graphical format. A graph is an excellent way to determine if a critically damped feedback system has been achieved. To use the serial plotter, each printed variable should be displayed using the `Serial.println` command. Example code follows.

```
1 Serial.print(pV);
2 Serial.print('\t');
3 Serial.println(sP);
```

Circuit I: System information

Hook up a photoresistor and LED as shown in the circuit diagram (Start with $R_L = 10$ kΩ). Tilt the photoresistor and LED so that they physically face each other. (The light from the LED must reach the surface of the photoresistor in order for the feedback control to have any effect in this circuit.)

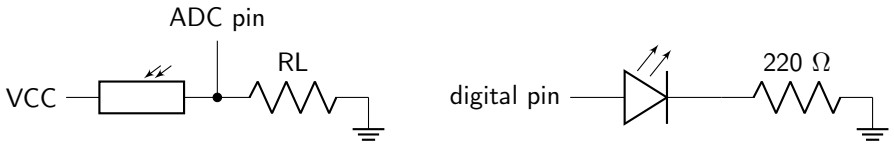


Figure 8.1: Schematics for the photoresistor (left) and LED (right).

Use the ADC in 8-bit mode. You will be collecting data about your system so that in Circuit II you can set up a feedback system where the LED is dimmed or brightened to obtain a desired light level on the photoresistor. To start, you will need to determine which color LED provides the greatest contrast with the photocell you are working with. You will do this by turning the LED on and recording the ADC value of the photocell. Then turn the LED off and record the ADC value. Subtracting these values gives the contrast. Record all of these measurements and calculations in Table 8.1.

Color	On ADC value	Off ADC value	Contrast
Red			
Yellow			
Green			
Blue			
White			

Table 8.1: Color contrast data.

Chose the LED with the greatest contrast and record the color below.

Now you will use the ADC values (corresponding to the LED color that you chose) to calculate the value of the resistance of the photocell when the light is ON (R_{ON}), and when the light is OFF (R_{OFF}). This will help you chose the best resistor to use in the place of R_L, using Equation 8.6.

$$R_{CELL} = R_L \left(\frac{256}{\text{ADC value}} - 1 \right) \tag{8.6}$$

Download `lab8_workbook.xlsx` to determine the value of R_L. Record all of this information in Table 8.2.

Parameter	Value
Ron	
Roff	
RL	

Table 8.2: Resistor data.

With the circuit rebuilt with the best color LED and best load resistor (R_L) in place, once again turn the LED on and record the photocell ADC value. Multiply that value by 85% which gives you the setpoint value, and record the setpoint below. You are now ready to move to Circuit II.

NOTE: Any change to your system configuration after this point (LED and photoresistor) will require you to repeat the setpoint measurement!

8.6 A note on system noise

As you start to display data on the serial monitor, it is important to note that there will likely be some noise (oscillations around an average) present in the process variable (`pV`). As the feedback system attempts to compensate for the current level of error, there may be some small fluctuations in the output. This is okay as long as the fluctuations are relatively small. Figure 8.2 shows the setpoint (black line) and process variable (gray oscillations) as viewed on the serial plotter. This level of noise is normal and should not be a cause of concern. It is not necessarily indicative of an underdamped system.

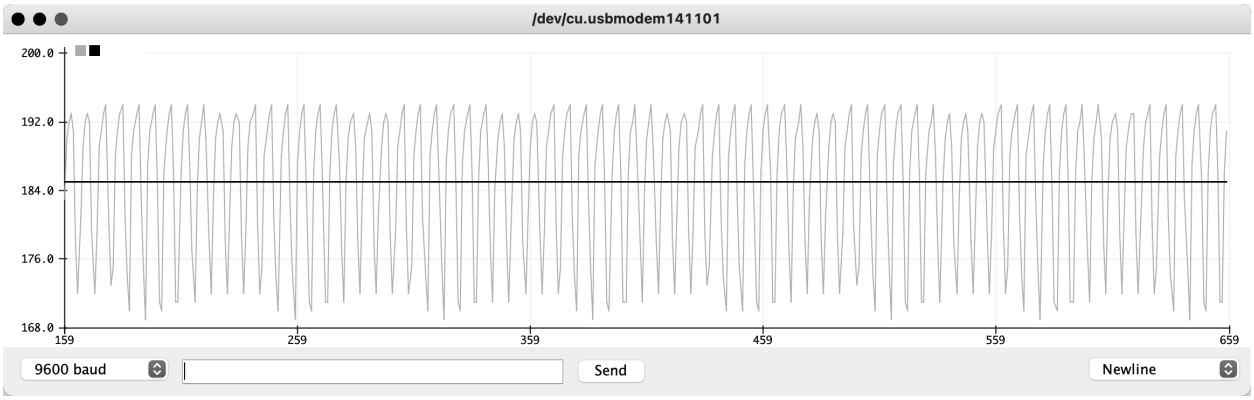


Figure 8.2: A normal amount of noise on the process variable output as seen on the serial monitor.

The output response of the LED may be subtle. Visible light-level oscillations on the LED means that you have an underdamped system and should reduce K_P, K_I, or τ values as needed.

Figure 8.3 shows a proportional control only (no integral control) output that is somewhere between critically damped and overdamped in the output response. The general trend of the process variable is to return to the setpoint when the light levels are changed by introducing and removing a flashlight from the system.

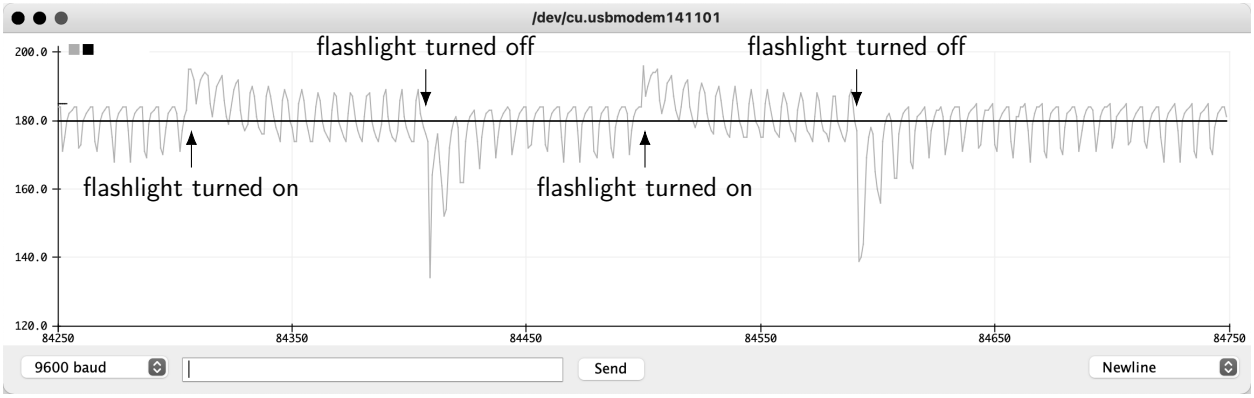


Figure 8.3: The process variable is somewhere between critically damped and overdamped.

Circuit II: Proportional control of light levels

This circuit will vary the brightness of an LED to keep the light intensity hitting a photoresistor at a constant value. If extra ambient light hits the photoresistor (from a flashlight, for example), the LED will be dimmed in response. As the extra light goes away, the LED will brighten to maintain a constant level on the photoresistor.

The flowchart for this circuit is given on page 107. Use proportional control to alter the brightness of the LED using 8-bit fast PWM on timer/counter 0. The brightness of the LED can be lowered by decreasing the duty cycle (which is modified with `OCR0A`), and the brightness of the LED can be raised by increasing the duty cycle. Ideally, the photocell will always measure a brightness value equal to the setpoint (`sP`), which was determined in the previous circuit. If the value read from the photocell from the ADC is too high, then the duty cycle of the PWM signal will need to be decreased, and vice versa. The ADC value is therefore the process variable (`pV`). Because we are using 8-bit PWM, it makes sense to use the ADC in 8-bit mode to have compatible signals.

The value of the proportionality constant (K_P) can be set to 0 to start to see the effect of no feedback control. Experiment with different values of K_P to see what leads to the most desirable outcome in the circuit. Try at least 5 different values between approximately 0 and 250 (K_P, after being divided by 10, will actually go between 0 and 25), and make notes on the system responsiveness. You will be asked to justify your choice of K_P in the lab report. You will ideally want a value of K_P that leads to a critically damped response, which means you need to know which values lead to underdamped results, and which lead to overdamped results. Make notes about your selection of K_P value below, as needed.

Run the code, and use the Serial Plotter to track the value of `pV`, the process variable, and `sP`, the setpoint value. These two values should ideally be equal. **In addition, view the PWM signal on an oscilloscope!** Shine a flashlight on the photoresistor and track how well the feedback system works in response to the new light level. (Note: holding the flashlight too close to the photoresistor may lead to unstable feedback control!) Tune K_P so that it leads to a desirable response time when the flashlight is turned on and off.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: Proportional-integral control of light levels

This circuit will vary the brightness of an LED to keep the light intensity hitting a photoresistor at a constant value. If extra ambient light hits the photoresistor (from a flashlight, for example), the LED will be dimmed in response. As the extra light goes away, the LED will brighten to maintain a constant level on the photoresistor.

The flowchart for this circuit is given on page 107. Using the value of K_P that you determined in the last circuit, amend your code from Circuit II to use proportional-integral control. Now vary the value of K_I between 0 and 250 (K_I, after being divided by 100, will actually go between 0 and 2.5). Use your cellphone flashlight to track how well the feedback system works. Tune K_I so that it leads to a desirable response time when the flashlight is turned on and off.

Try 5 different values of K_I, and make notes on the system responsiveness. You will be asked to justify your choice of K_I in the lab report. Make notes about your selection of K_I below, as needed.

With your chosen value of K_I (in other words: do not change K_I at this point!), try 5 different values of τ and make notes on both the system responsiveness and the memory used by the software. If changing τ has no effect on the output response, then you likely have a coding error that is not actually creating an integral-control term. You will be asked to justify your choice of τ in the lab report. Make notes about your selection of τ below, as needed.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

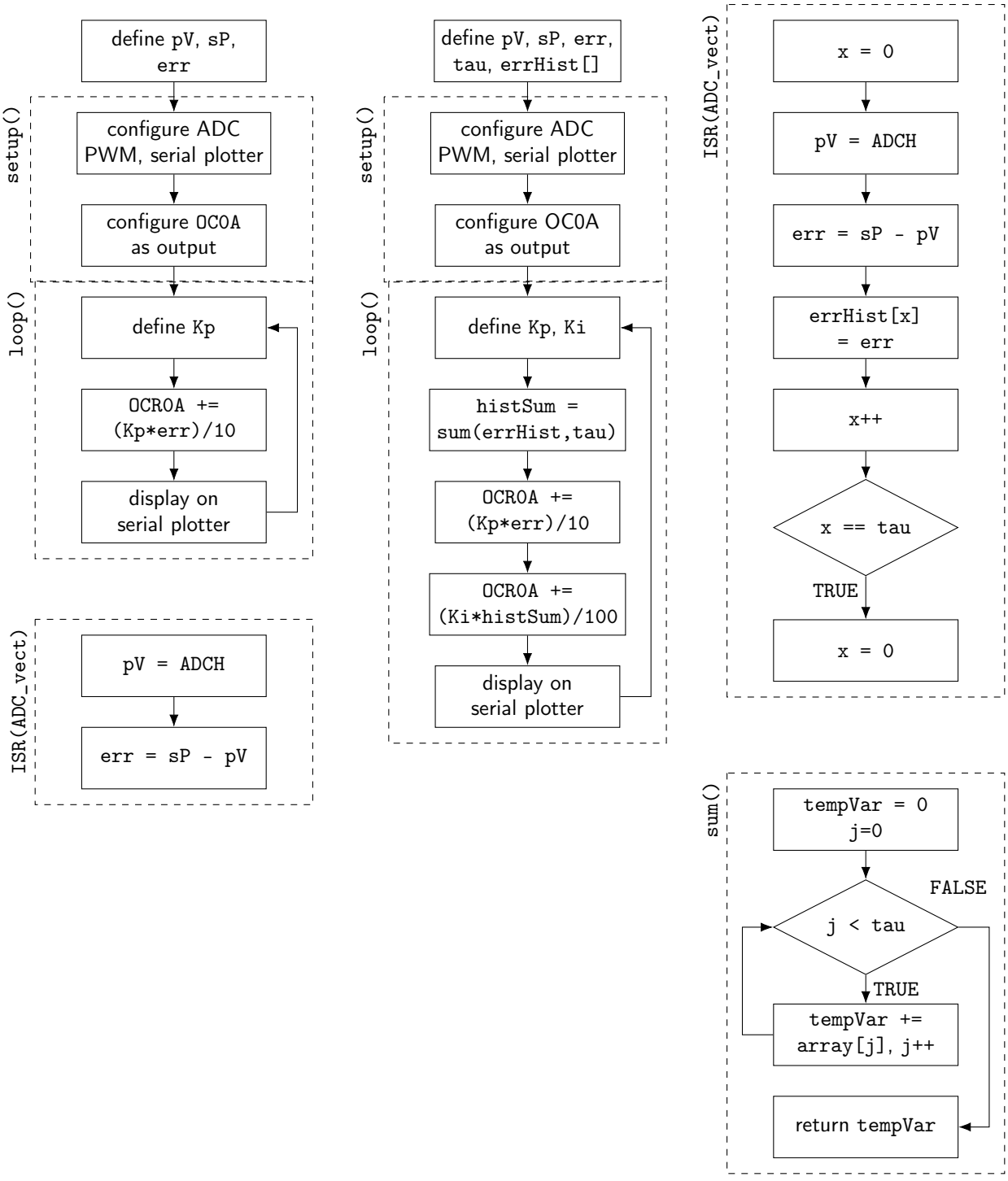


Figure 8.4: Flowcharts for Circuit II (left) and Circuit III (right).

Lab 8 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Explain and justify your choices of LED color and R_L value in Circuit I.
- Why are `Kp` and `Ki` stored as 10 and 100 times their actual values, respectively? Why not save these variables as floats?
- Present your data on different values of `Kp`, `Ki`, and `tau` in tabular form.
- Explain and justify your choice of `Kp`, `Ki`, and `tau`.
- Was there an advantage to using PI control rather than just P control? If so, what was the advantage?
- When might PI control benefit your Smart Car project?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 9

Carefully read the entirety of Lab 9, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. On a primary device, explain why $\overline{\texttt{SS}}$ has to be configured as an output pin, and why this must occur before configuring the SPI control register. (This is not directly addressed in the lab. Think about it!)

2. How will each of the SPI pins need to be configured (using `DDRB`) in Circuit I?

3. How will each of the SPI pins need to be configured (using `DDRB`) in Circuit II?

4. How will each of the SPI pins need to be configured (using `DDRB`) in Circuit III for the primary device?

5. How will each of the SPI pins need to be configured (using `DDRB`) in Circuit III for the secondary device?

6. In Circuit III, you will need to use an external interrupt to generate a random number. How will you trigger the interrupts (on rising edge only, falling edge only, or on toggle)? Why?

7. Find the final value of `x` after each loop has been executed.

```
unsigned int x = 500;
while (x < 500) {
  x--;
}
```

```
unsigned int x = 0;
do {
  x++;
} while (x <=20);
```

```
unsigned char x = 2;
while (x < 5) {
        x *= x;
}
```

Lab 9: SPI: Serial Peripheral Interface

This lab will introduce the serial peripheral interface (SPI) serial I/O protocol. **C Concepts:** `while` loop, `do`/`while` loop **AVR Concepts:** SPI and registers `SPCR`, `SPSR` and `SPDR`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

9.1 SPI communication

SPI communication on the ATmega328P is capable of full duplex synchronous communication using four wires. Each of the four logic signals is associated with a particular pin on Port B. These signals and their associated pins are defined in Table 9.1. (Note that if using the ATmega328P in secondary mode, the $\overline{\texttt{SS}}$ pin is active-LOW, hence the overbar in the signal name. This is not universally true for all SPI secondary devices.)

Name	Pin	Description
$\overline{\texttt{SS}}$	D10	Secondary select (output from primary)
MOSI	D11	Primary output, secondary input
MISO	D12	Primary input, secondary output
SCK	D13	Serial clock (output from primary)

Table 9.1: Each of the logic signals used in AVR serial peripheral interface communication.

When two circuits are connected together as primary-secondary devices, they will be connected as shown in Figure 9.1.

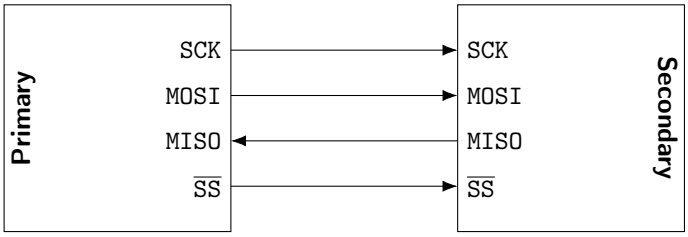


Figure 9.1: Connection between one primary device and one secondary device using SPI.

Multiple secondary devices can be supported with SPI, either independently or daisy-chained. This is discussed in the course textbook.

9.2 Transmitting and receiving data using SPI

Transmission of data is initiated upon writing to the `SPDR` register. When the primary device is sending data, the value written to `SPDR` will send through the `MOSI` pin. When the secondary device is sending data, the value written to `SPDR` will send through the `MISO` pin. Once all of the data has been transmitted, the `SPIF` bit in the `SPSR` register will be set.

If full-duplex mode is being used, initiating data transfer will also initiate a reception of data. That is, data written to SPDR will transmit out, and any data being received will be present in the SPDR register once the transmission is complete. This process flow is depicted in Figure 9.2.

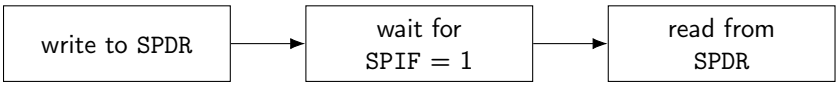


Figure 9.2: Transmit / receive process flow in SPI.

Because data can be received by writing to SPDR, it is not always necessary to use an interrupt to receive data. An interrupt may be useful if the timing of the signal is truly unknown (possibly if using the Arduino as a secondary device, with a primary device that cannot have its programming altered) or if writing to SPDR and polling the result is not otherwise feasible in the program flow.

9.3 ATmega328P SPI primary and secondary modes

As mentioned, interrupts are not always necessary on either the primary or the secondary device. Therefore, the software code only requires polling after sending data to determine if the transmission has been completed.

An important consideration is to set the $\overline{\text{SS}}$ pin as an output pin on the primary device. This has to be done before SPI is enabled in the SPCR register.

The program flow for both a primary and secondary device using SPI follows.

1. Configure output pins.
2. Configure SPI using SPCR and SPSR.
3. Write to SPDR.
   (a) If transmitting data, write the character to be transmitted directly to SPDR.
   (b) If receiving (but not transmitting) data, write any 8-bit value to SPDR.
4. Wait for SPIF to be set by the microcontroller.
5. Read from SPDR.
   (a) If receiving data, save this value to a variable.
   (b) If transmitting (but not receiving) data, you do not need to read the contents of SPDR.

To configure SPI communication on a secondary device, all SPI pins except for MISO must be input pins. To operate as a secondary, the MSTR bit in SPCR must be zero. Any of the clock rate select bits will be ignored. All of the other settings in SPCR must be the same as those used for the primary device.

9.4 SPI registers on the ATmega328P

There are three registers associated with SPI communication on the ATmega328P. They are described in Appendix A.

- SPCR – **SPI Control Register**: This register configures and enables the SPI communication protocol.
- SPSR – **SPI Status Register**: This register contains information about the most recent transfer/receive process. It is also used in connection with SPCR to set the SPI data transfer speed.
- SPDR – **SPI Data Register**: This read/write register is used for data transfer and receiving. When the SPDR register is written to, data transfer is initiated. When data is received, it is written to this register.

9.5 74595 8-bit SIPO shift register

The 74595 is an 8-bit serial-in / parallel-out (SIPO) shift register. This means that a 1-bit data stream (coming from the `MOSI` pin) consisting of 8 bits of information is capable of driving 8 independent output devices. The output pins on the 74595 are labeled Q_A–Q_H. In order to properly synchronize the Arduino and the shift register, a clock signal (connected to the `SCK` pin) will be required to ensure that the devices operate at the same frequency. Finally, a latch signal (connected to $\overline{\texttt{SS}}$ pin) on the shift register is also required to signal to the register when to load data and when to latch data.

The pinout diagram of the 74585 shift register is included in Appendix B. Table 9.2 outlines which pins on the 74595 register must be connected to each SPI signal. ($\overline{\text{OE}}$ is an active-LOW enable output enable pin, and $\overline{\text{SRCLR}}$ is an active-LOW enable shift register clear pin. These two pins are not associated with the SPI protocol.)

74595 Pin	SPI Signal
RCLK	$\overline{\texttt{SS}}$
SER	`MOSI`
QH'	`MISO`
SRCLK	`SCK`

Table 9.2: Connections between the 74595 shift register and the SPI protocol.

9.5.1 74595 modes of operation

Depending on the values on the SRCLK, RCLK and SER pins, the 74595 can operate as defined in Table 9.3. Because data is sampled on the rising edge, the clock phase must be zero in the SPI control register.

SER	SRCLK	RCLK	Description
LOW	↑	LOW	Data shifts and $Q_A = 0$
HIGH	↑	LOW	Data shifts and $Q_A = 1$
×	×	HIGH	Data is stored (latched)

Table 9.3: Modes of operation of the 74595 shift register.

9.6 74165 8-bit PISO shift register

The 74165 is an 8-bit parallel-in / serial-out (PISO) shift register. This means that 8 bits of data coming from an external device can be converted into a serial data stream and be input to the ATmega328P using only a single pin. In effect, this allows for an expansion of digital input pins, much in the same way that using the 74595 allows for an expansion of output pins.

The pinout diagram of the 74165 chip is included in Appendix B. Note that H is the MSB of the chip, and A is the LSB. Table 9.4 outlines which pins on the 74165 register must be connected to each SPI signal. (CLK INH is an active-HIGH clock inhibit signal. This pin is not associated with the SPI protocol.)

74165 Pin	SPI Signal
SH/$\overline{\text{LD}}$	$\overline{\texttt{SS}}$*
SER	`MOSI`
QH	`MISO`
CLK	`SCK`

Table 9.4: Connections between the 74165 shift register and the SPI protocol. (*See note below about the secondary select signal for this chip.)

It is important to note that the secondary select pin that on the 74165 chip is labeled SH/$\overline{\text{LD}}$. This means that data is loaded (parallel load) when the pin is LOW, and will shift through to the output when the pin is HIGH. This implies that the secondary select is active-HIGH! Therefore, the secondary select signal must be written LOW to inhibit data transfer, and written HIGH to enable it. This is the opposite of how the 74595 pin works.

9.6.1 74165 modes of operation

Depending on the values on the SH/$\overline{\text{LD}}$, CLK and SER pins, the 74165 can operate as defined in Table 9.5. Because data is sampled on the rising edge, the clock phase must be zero in the SPI control register.

SH/$\overline{\text{LD}}$	CLK	SER	Description
LOW	$\times$	$\times$	Parallel load
HIGH	LOW	$\times$	Data stores (latches) data
HIGH	$\uparrow$	HIGH	Data shifts and $Q_A = 1$
HIGH	$\uparrow$	LOW	Data shifts and $Q_A = 0$

Table 9.5: Modes of operation of the 74165 shift register.

9.7 Control flow: iterative

A microcontroller is capable of executing specific segments of code a certain number of times (or infinitely). This is known as iterative flow. The conditional flow functions used in C are `for` loops, `while` loops, and `do`/`while` loops.

9.7.1 `while` and `do`/`while` loops

`while` and `do`/`while` loops are a type of iterative control flow. They are used when a piece of code will be repeated until a certain condition is satisfied. In a `while` loop, a variable is checked against a condition. When that condition is satisfied, the code inside the loop executes. Then the condition is re-checked. This process continues until the condition is not satisfied, at which point the code leaves the loop.

```
1 while (condition) {
2   // this code will execute if the condition is satisfied
3   // then, the condition will be checked again
4 }
```

Slightly different from the regular `while` loop is the `do`/`while` loop, which executes the body of the code once before the condition is checked. The syntax is shown below.

```
do {
  // this code will execute once
  // then, the condition will be checked again
} while (condition);
```

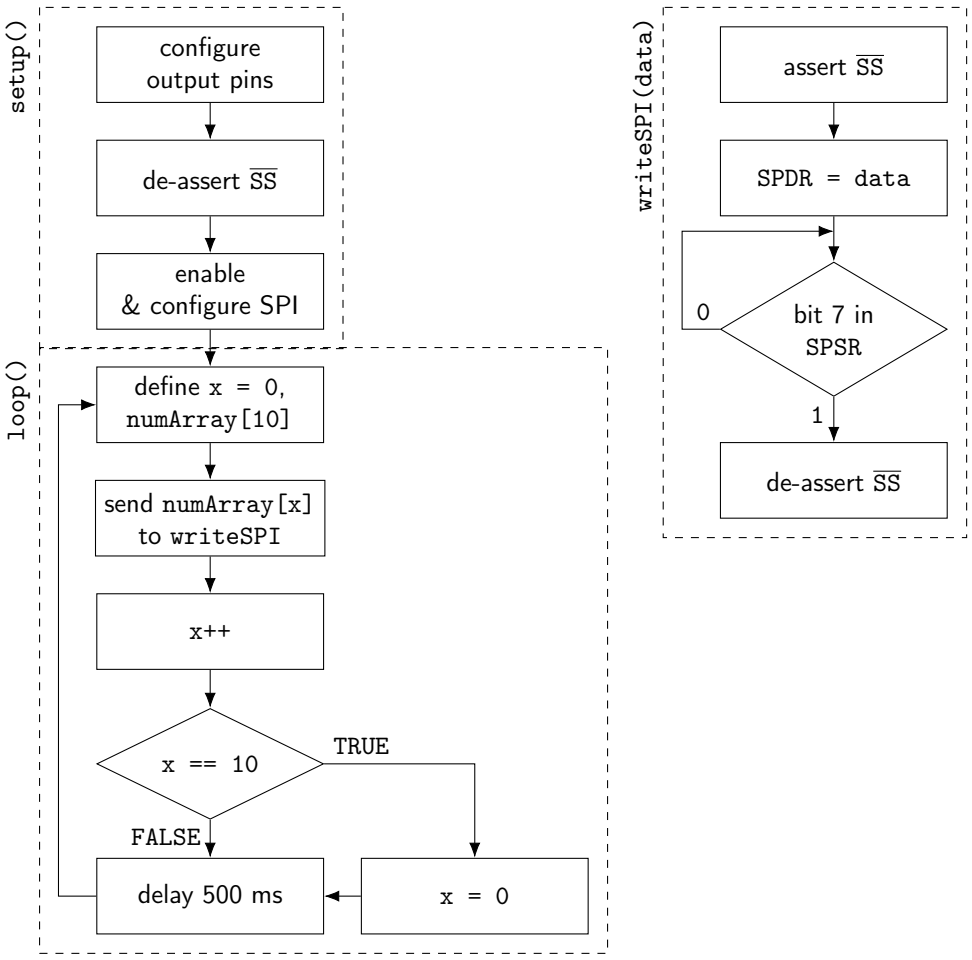
In this lab, you will use `while` or `do`/`while` to check the value of the `SPIF` flag. While the flag is NOT set (serial communication has not concluded), a loop will cause the code to wait until the message has been completely transmitted or received. Only after the flag is set will the code continue to execute.

Circuit I: SPI control of 7-segment display

This circuit will display decimal characters 0–9 on a 7-segment display. Every 500 ms, the value on the display will increment. After displaying 9, the display will cycle back to 0 again.

Download the file `lab9_circuit1.ino`. Connect segments a–g via current-limiting resistors to pins Q_A–Q_G on the 74595 chip (alternatively, you may use the SPI 7-segment display PCB). Run the software code. Once running, you should see similar results as from Lab 3 Circuit I (however, this particular circuit counts from 0–9 and does not include additional hexadecimal characters).

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.



Program Memory: _____ bytes

Data Memory: _____ bytes

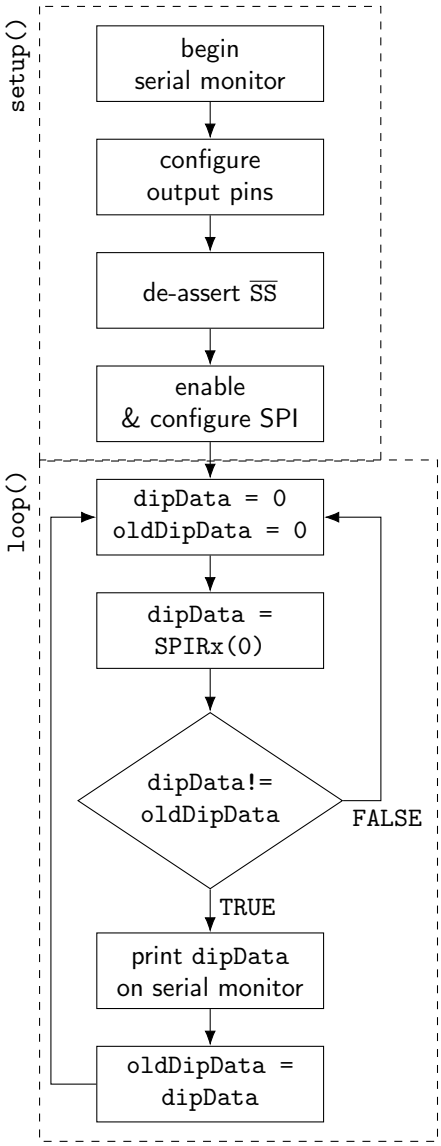
Instructor Stamp: _____

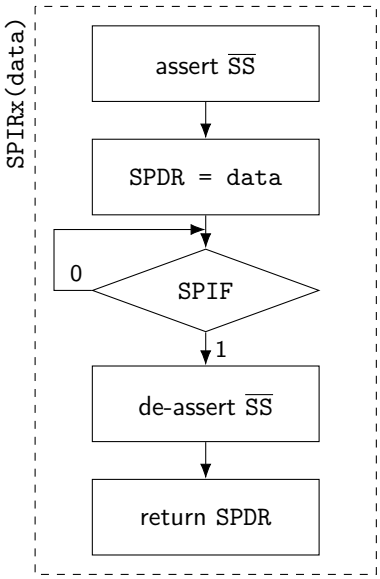
Circuit II: 8-bit serial input from DIP switch

This circuit will display the decimal value entered into an 8-bit DIP switch on the serial monitor. Data will only update on the serial monitor when there has been new data entered into the DIP switch.

Use a 74165 PISO register along with a DIP switch with eight 10 kΩ pull-down resistors. Each of the DIP switch outputs should connect to register inputs A–H. When the code is working properly, changing the value on the DIP switch will trigger the serial monitor to display the DIP switch value.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.





Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: Arduino-Arduino SPI communication

This circuit will use two Arduinos to play a dice-rolling game. When a button is pressed on either Arduino, a random dice-roll will be generated and displayed on a 7-segment display connected to that Arduino. The two Arduinos will compare values and determine the winner, or if a tie occurred. The winning Arduino will light a green LED. The losing Arduino will light a red LED. If a tie occurs, both Arduinos will light a blue LED.

This circuit's flowchart is provided on page 121. This circuit will require two Arduino Uno boards. Both of them will have the same hardware: pushbutton, RGB LED, and BCD to 7-segment decoder PCB. Connect each pushbutton to one of the external interrupt pins. Use 1 kΩ resistors as current-limiting resistors for the RGB LEDs.

Use cli() at the beginning of your setup functions and sei() at the end of your setup functions to ensure that interrupts do not cause errors with the configuration of your peripherals. Because the primary Arduino will only ever be addressing a single secondary Arduino, the $\overline{SS}$ pin does not need to be connected on the primary. On the secondary, the pin should be connected directly to ground.

By pressing the pushbutton on each Arduino, a random number will be generated and sent to the other Arduino. These two random numbers will be compared. The Arduino with the larger number wins. When an Arduino wins, a green light turns on. When an Arduino loses, a red light turns on. During a tie, a blue LED turns on.

All data that will be transmitted and received by each Arduino will occur upon use of the `spiTxRx()` function. You will not need to use SPI interrupts. This also means that **both** the primary and the secondary will need to have this function included in their code.

You will need to ensure that power is always being supplied to both Arduinos via their USB connections. A common ground signal between both Arduinos is necessary for this circuit to function properly. Be careful not to accidentally upload code onto the wrong board! You will need two independent `.ino` files, one for the transmitting Arduino, and one for the receiving Arduino. You may switch between boards using the `Tools > Port` menu in the Arduino IDE or use two computers. **Note:** this circuit may not work properly if the secondary is turned on first. Ensure that the primary code is loaded first, and then load the secondary code.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Primary Device

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Secondary Device

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

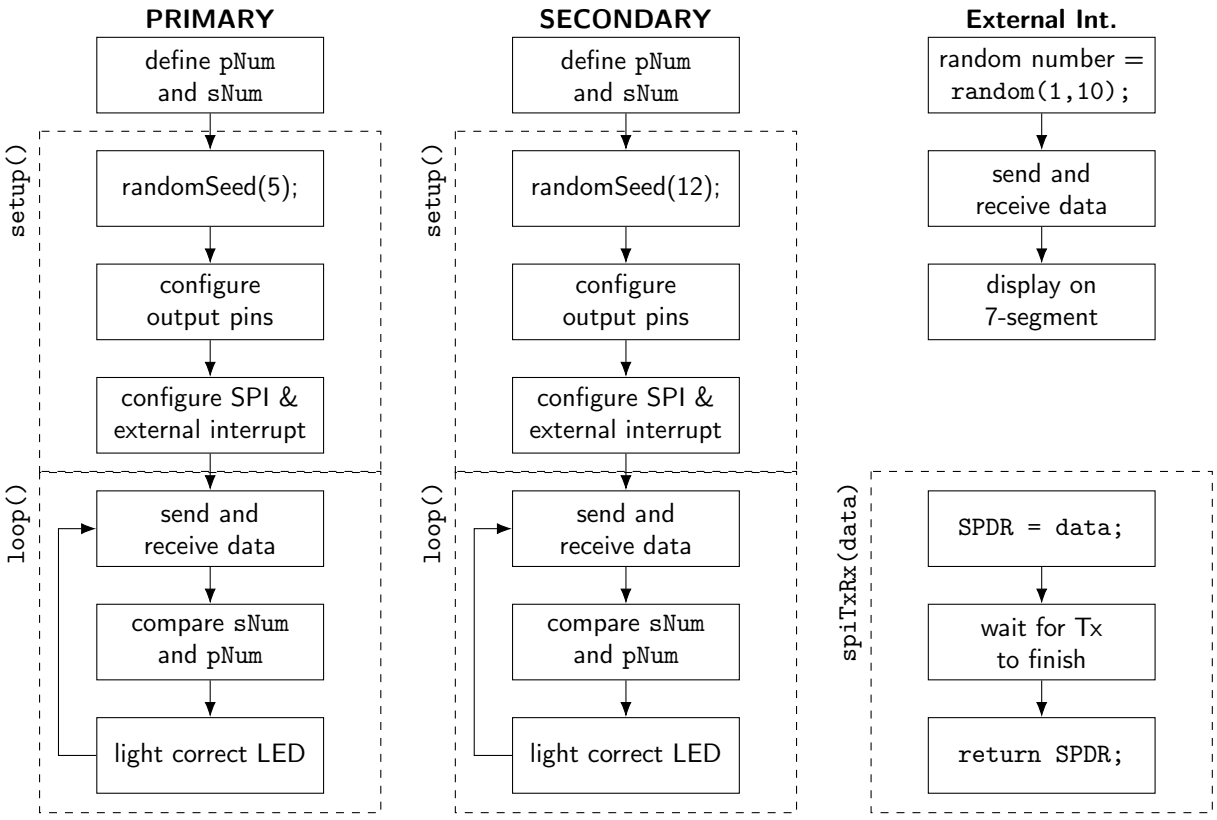


Figure 9.3: Flowchart for the Arduino in Circuit III.

Lab 9 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Describe **at least** two benefits of SPI over parallel I/O.
- Describe **at least** one specific enhancement that you can include in your Smart Car project that uses SPI.

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 10

Carefully read the entirety of Lab 10, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. How does changing the clock frequency affect the power consumed by the microcontroller?

2. How does changing the operating voltage (VCC) affect the power consumed by the microcontroller?

3. What are the minimum and maximum operating voltages of the ATmega328P? (Refer to the datasheet if needed.)

Lab 10: Power Consumption and ATmega328P without Arduino

In this lab, the concept of power consumption, and techniques to reduce it, will be explored. The ATmega328P chip will be programmed independently of the Arduino to realize even more power reduction, as well as to have the ability to program independent of the Arduino environment in future projects. **AVR Concepts:** Sleep modes, system clock speed, prescaler and register `CLKPR`, power reduction and register `PRR`, system fuses (low byte, high byte, extended byte)

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

10.1 Sleep modes

The ATmega328P contains six sleep modes that allow different modules of the microcontroller to be shut down, causing a reduction in power consumption. Each sleep mode will allow certain peripheral features to run, and can be woken up from different sources. These are outlined in the class textbook. Entering a sleep mode is a two-step process.

1. Set `SE` in the `SMCR` register while configuring the `SM` bits for the desired sleep mode.
2. Use the `SLEEP` instruction (`asm volatile("sleep");` in the Arduino IDE).

On the Arduino Uno, the extended fuse byte is programmed to enable brownout detection. This will kick in during a sleep mode to monitor for low VCC voltage levels. While the BOD can be a useful feature, it does consume extra power. To disable BOD during a sleep mode, complete the following process.

1. Set both `BODS` and `BODSE` in the `MCUCR` register.
2. Set the `BODS` bit and clear the `BODSE` bit in the `MCUCR` register.

BOD will become activated again once the microcontroller wakes up from sleep. (Note that the BOD is disabled by default on the factory fuse settings on the ATmega328P.)

10.2 Clock speed

The clock speed on the ATmega328P can be varied by either dividing the present clock source by a prescaler using the `CLKPR` register or by using a different clock source. There are many possible clock sources that can be used with the microcontroller; the device must be configured by programming the appropriate fuse bits.

The Arduino Uno uses an external clock with a frequency of 16 MHz. However, it is possible to use various clock speeds with the microcontroller. An internal 8 MHz RC oscillator and an internal 128 kHz RC oscillator are both available built-in to the device. The highest possible clock frequency is dictated not only by the physical properties of the clock, but also by the value of VCC used on the microcontroller. This relationship between VCC and maximum clock frequency is given in Figure 10.1, where the safe operating area is shaded grey.

10.2.1 `CLKPR` – Clock prescale register

This register configures the system clock prescaler value. More information about this register is available in Appendix A. It will affect every synchronous peripheral on the ATmega328P microcontroller. Setting the prescaler is a two step process. First, the `CLKPCE` bit must be set. Then, the register must be set equal to the corresponding value with `CLKPCE` clear and the `CLKPS` bits configured as needed. For example, to use

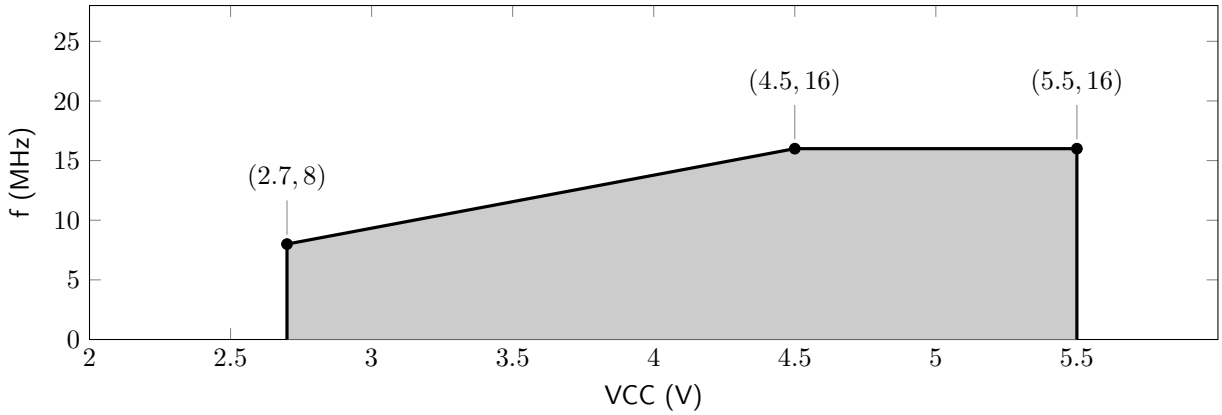


Figure 10.1: Relationship between VCC and the clock frequency. For a given value of VCC, the clock frequency must be low enough to reside in the shaded grey area of the graph.

a prescaler value of 4, the following code should be executed (note that interrupts are disabled during this process).

```
1 cli();
2 CLKPR |= 0x80;
3 CLKPR = 0x02;
4 sei();
```

10.3 PRR – Power reduction register

This register enables and disables peripheral devices on the microcontroller. When peripherals are disabled, less power is consumed, but functionality is reduced. More information about each bit in this register is available in Appendix A.

10.4 Unused pins

According to the ATmega328P datasheet, unused pins should always have a defined voltage level. Unused pins are input pins by default. A floating value is undefined. The datasheet suggests activating the internal pull-ups on all unused pins to reduce power consumption.

10.5 ATmega328P fuses

Fuses are bits of nonvolatile memory in the ATmega328P microcontroller that configure the functionality of the device not pertaining to the actual program code. This data is preserved even when the device is reprogrammed, but can itself be changed using an external programming device. For example, fuses contain information about the clock source used on the microcontroller. There are three fuse bytes on the ATmega328P: the low byte, the high byte, and the extended byte. Information about the three fuses are available in Appendix A.

Programmed fuses have a value of 0, while unprogrammed fuses have a value of 1.

- **Extended Fuse Byte**: This fuse configures the brown-out detection unit.
- **High Fuse Byte**: This fuse configures features that are relevant to programming and debugging the ATmega328P microcontroller.
- **Low Fuse Byte**: This fuse configures features that are relevant to the clock source and startup time of the microcontroller.

Circuit I: Arduino power consumption: baseline

This circuit will turn an LED either ON or OFF for the purposes of collecting power consumption data.

Using an external power supply, connect a multimeter as shown in Figure 10.2.

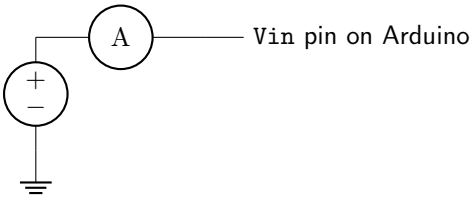


Figure 10.2: Circuit diagram for connecting an external power supply and ammeter to the Arduino.

The Arduino should NOT be connected to the USB port when you take measurements! Therefore, disconnect the Arduino from the USB after you have uploaded your code to the microcontroller. Write code that turns on an LED connected to pin D7. The LED will switch between ON and OFF when INT0 is triggered by a pushbutton. Measure the average current in this situation and record it in Table 10.1. Then, change your code so that the LED turns off. Record the average current in Table 10.1.

Parameter	LED ON	LED OFF
VCC (V)	5.0	5.0
I (mA)		
P (mW)		

Table 10.1: Voltage, current, and power data for Circuit I.

Calculate the power consumption using Equation 10.1, and record for both the ON and OFF conditions in Table 10.1.

$$P = IV \tag{10.1}$$

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Sleep modes

This circuit will utilize two pushbuttons. One pushbutton will put the microcontroller into a sleep mode. The second pushbutton will exit the sleep mode and turn an LED either ON or OFF.

With VCC at 5 V, utilize each of the sleep modes of the microcontroller to determine their effect on power consumption. In addition to the pushbutton used to toggle the LED between ON and OFF using INT0, use a second pushbutton and INT1 to put the microcontroller into the sleep state. (Instead of triggering the sleep condition in the ISR, it is recommended that you set a flag in the INT1 ISR, which will cause conditional logic in the loop function to activate the sleep mode.) To ensure that the brown-out detection (BOD) unit does not work, be sure to deactivate that functionality as well. The LED should be OFF when the microcontroller is asleep.

Record all power data in Table 10.2.

	awake		**asleep**
Sleep mode	**LED ON**	**LED OFF**	**LED OFF**
Idle			
ADC noise reduction			
Power down			
Power save			
Standby			
Extended standby			

Table 10.2: Power data for Circuit II.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: CLKPR and PRR registers

This circuit will utilize two pushbuttons. One pushbutton will put the microcontroller into a sleep mode. The second pushbutton will exit the sleep mode and turn an LED either ON or OFF.

Now that you are comfortable with using sleep modes, make an attempt to reduce power consumption as much as possible by making additional changes to your software and hardware. Use all available methods at your disposal to reduce power consumption as much as possible. The LED must remain visibly ON when enabled, and the pushbuttons must always be capable of triggering interrupts. (You will know if this functionality fails because the LED will no longer toggle between ON and OFF states.) **Only change one thing at a time so you can track its effect on power consumption.** Make any notes below as needed. Once you have optimized your power consumption as much as possible, record your data in Table 10.3.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

	awake		**asleep**
Parameter	**LED ON**	**LED OFF**	**LED OFF**
VCC (V)			
I (mA)			
P (mW)			

Table 10.3: Voltage, current, and power data for Circuit III.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit IV: ATmega328P

This circuit will utilize two pushbuttons. One pushbutton will put the microcontroller into a sleep mode. The second pushbutton will exit the sleep mode and turn an LED either ON or OFF. This circuit will use the ATmega328P without the Arduino Uno platform.

Using your optimized code from Circuit III, go to Sketch > Export compiled Binary. You will find in your sketch folder two new files, one of them a .ino.standard.hex, and the other with a bootloader included. Open the standard.hex file in the programmer software to program it onto an ATmega328P chip. In the programming software, go to the Config tab to configure the fuse bytes.

Connect the ATmega328P chip as given in the pinout diagram provided in Appendix B. Unless you have disabled the $\overline{\texttt{RESET}}$ pin in the high fuse byte, connect the $\overline{\texttt{RESET}}$ pin directly to VCC. At this point, feel free to try changing different fuse settings until you are satisfied that you have achieved the lowest possible power consumption where the LED is able to remain visibly ON when enabled, and as long as the pushbuttons are capable of triggering interrupts. **Only change one thing at a time so you can track its effect on power consumption.** VCC may not be less than 2.7 V. Record all voltage, current, and power data in Table 10.4.

	awake		**asleep**
Parameter	**LED ON**	**LED OFF**	**LED OFF**
VCC (V)			
I (mA)			
P (mW)			

Table 10.4: Voltage, current, and power data for Circuit IV.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.

Instructor Stamp: _____

Lab 10 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Circuit I/II (which are the same and can share one schematic), Circuit III, and Circuit IV should have their own schematic, clearly including the power supply and voltage used, ammeter, LED color, current-limiting resistor value, and all other hardware and I/O connections.
- In addition to discussing changes you would make to the code that you submitted, list any changes that you would make (if any) to further reduce your power consumption. If you would not make any further changes, justify that choice.
- Explain all of the methods you used to reduce power consumption by the microcontroller. (Include this in a separate section and include all of the methods in a bullet-point list, table, or other easy-to-read format.)
- Of all the methods you used, what was the single most effective means you used to reduce the power consumption of your circuit?
- Give at least two advantages of using an Arduino Uno over using the barebones ATmega328P.
- Give at least two advantages of using an ATmega328P over using the Arduino Uno.
- Search online for the cost of a (genuine) Arduino Uno, give the cost in your report and cite your source. Then, search online for the cost of an ATmega328P DIP chip, give the cost in your report and cite your source. Explain the monetary advantage of using an ATmega328P over using an Arduino system.
- What HEX values for each of the fuse bytes did you use when programming the ATmega328P?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 11

Carefully read the entirety of Lab 11, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. What command is used to display ASCII characters on the serial monitor? (Note: if you completed the USART lab, then you will be using the USART to write this command, but you are still responsible for answering this question.)

2. What ASCII character corresponds to a line feed?

3. How will you connect the secondary select pin on the receiving Arduino?

4. Why is the secret message transferred starting with the numeral 8?

5. What information will you need from the other group before you can send and receive messages?

Lab 11: Transmitting and Receiving a Secret Message

In this lab, the SPI protocol will be used to send a secret message to another group, and in return receive a message that the other group is transmitting. The message will consist of three ASCII characters.

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

Circuit I: Transmit secret message

This circuit will repeatedly transmit a secret message using SPI to another Arduino.

The flowchart for this circuit is shown in Figure 11.1 (left) on page 137. **You will use an Arduino set to SPI primary mode to accomplish this circuit.** Decide with the other group how you'd like to configure the SPI protocol (prescaler, data order, clock phase, and clock polarity).

Write code that uses the SPI protocol to transmit a three character secret message (provided by your instructor) to the assigned lab group. You will want to send four characters. The numeral 8 will be sent first to indicate that it is the beginning of the message. Either send the value 8 as a numeral or send the character `'8'`. Either way, be sure that you and the other group are in agreement about how that value will be sent. **Send the secret message repeatedly (that is to say, not just once).**

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Receive secret message

This circuit will repeatedly receive a secret message using SPI from another Arduino.

The flowchart for this circuit is shown in Figure 11.1 (right) on page 137. **You will use a separate Arduino set to SPI secondary mode to accomplish this circuit.** Interrupts will be required on this device. Connect the secondary select pin directly to ground.

Write code that receives the three-character secret message from your assigned lab group. Display the received message on the serial monitor. You will need to use the `Serial.write()` command to print ASCII characters to the serial monitor. If you completed the USART lab, then you will manually configure and use the USART in lieu of using the "cheater functions."

To create a new line on the serial monitor, you may use the command `Serial.print('\n');`. If you are using the USART, then you will transmit the ASCII code that corresponds to a line feed. Use the website `https://www.ascii-code.com` to find the corresponding character.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

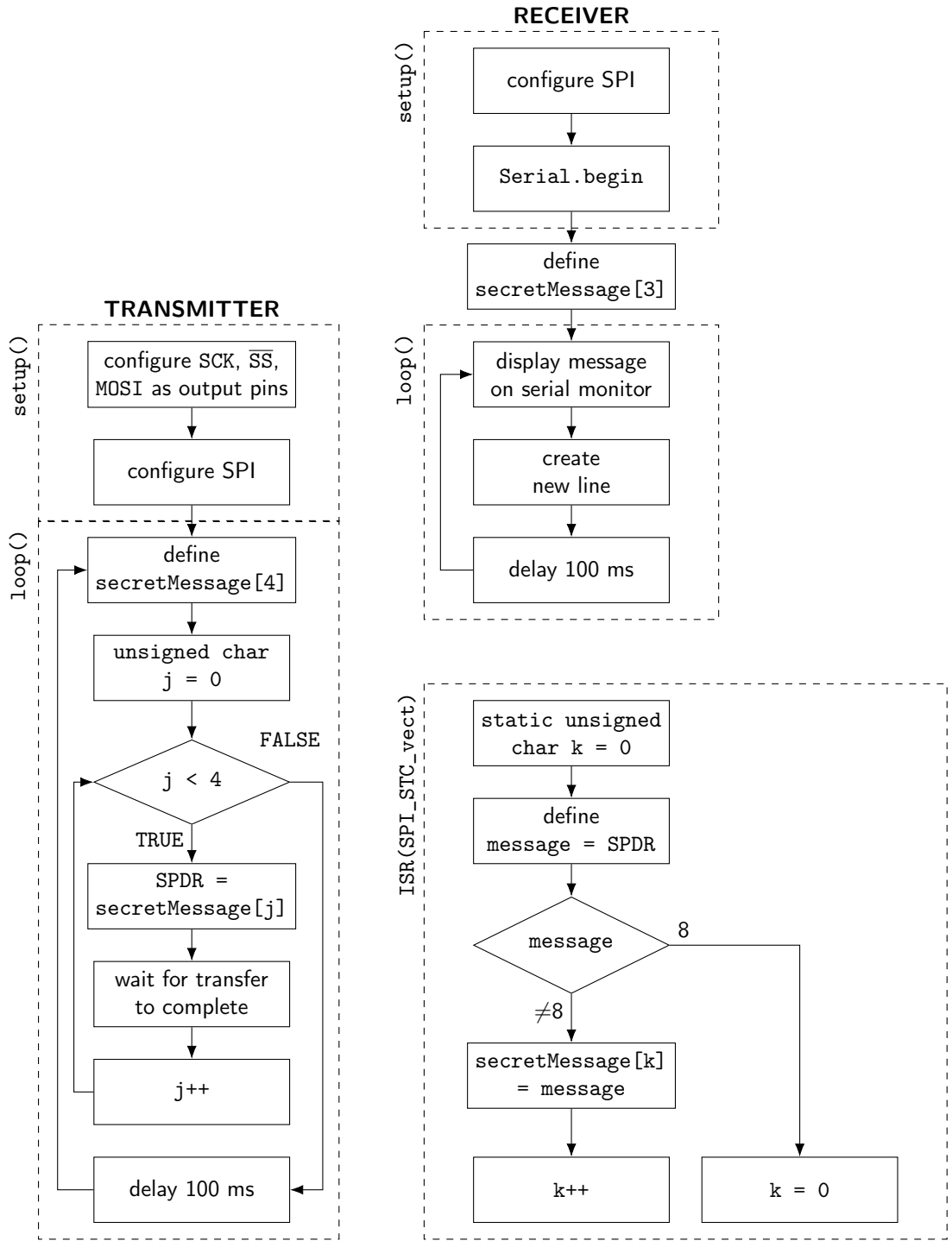


Figure 11.1: Flowcharts corresponding to the transmitter circuit (left) and receiver circuit (right).

Lab 11 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- (there are no special additions to make to this lab report)

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 12

Carefully read the entirety of Lab 12, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Which pin on the Arduino Uno corresponds to `ICP1`?

2. In this lab the CPU clock has a frequency of 16 MHz, and a prescaler of 256 will be used.
   (a) Calculate the **minimum** pulse-width that can be measured.

   (b) Calculate the corresponding **maximum** frequency of that minimum pulse-width.

3. If `count` is an unsigned long, how long will it take for the variable to overflow?

4. Why is timer/counter 1 used in normal mode and not CTC mode when using the input capture unit?

Lab 12: Ultrasonic Sensor

In this lab, the input capture unit on timer/counter 1 will be used to measure the width of a pulse. After verifying the measurement of pulse-widths, the HC-SR04 ultrasonic sensor will be used to create a distance-measuring circuit. **AVR Concepts:** timer/counter 1 input capture unit

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

12.1 Input capture unit

The input capture unit on timer/counter 1 can be used to determine the following timing properties of a square wave input into `ICP1` (D8).

- Period T
- High period T_{HIGH}
- Low period T_{LOW}

The timer/counter will most likely be used in normal mode (although this is not required). The prescaler of the timer/counter will dictate the precision of the measurements that can be made. The smaller the prescaler, the larger the precision, as a shorter interval of time will elapse between timer/counter increments. Precision is not always the most important consideration, however, as a high precision will lead to very large numbers being used to store timing data, requiring extra memory. Higher precision will also lead to more frequent overflow of the timer/counter register as well as any variables storing an accumulated sum of elapsed intervals on the timer/counter unit.

The input capture unit of timer/counter 1 is capable of triggering upon a falling edge (`ICES = 0`) or a rising edge (`ICES = 1`) depending on the value of the input capture edge select bit (`ICES`) in `TCCR1B`. Interrupts can be enabled to trigger upon an input capture. Therefore, recording two subsequent events will provide information about the square wave that is input into the device. Table 12.1 indicates what each value of `ICES` will need to be to record each of the different timing properties of a square wave. (Note that to measure the period of a square wave, the value of `ICES` does not matter, as long as it remains the same.)

Measurement	**Initial** `ICES`	**Subsequent** `ICES`
Period	0	0
Period	1	1
High period	1	0
Low period	0	1

Table 12.1: Required values of `ICES` to measure different square wave timing properties.

In software, calculations can be done to determine the timing of the wave. First, the number of counter increments can be calculated by using Equation 12.1, where K_1 is the initial value stored in `TCNT1` and K_2 is the subsequent value stored in `TCNT1`.

$$K = K_2 - K_1 \tag{12.1}$$

To convert this into an interval of time, the prescaler of the timer/counter unit (N) and the microcontroller I/O clock frequency ($f_{CLK,I/O}$) must be known. This calculation is defined by Equation 12.2.

$$T = \frac{KN}{f_{CLK,I/O}} \tag{12.2}$$

12.2 HC-SR04 ultrasonic sensor

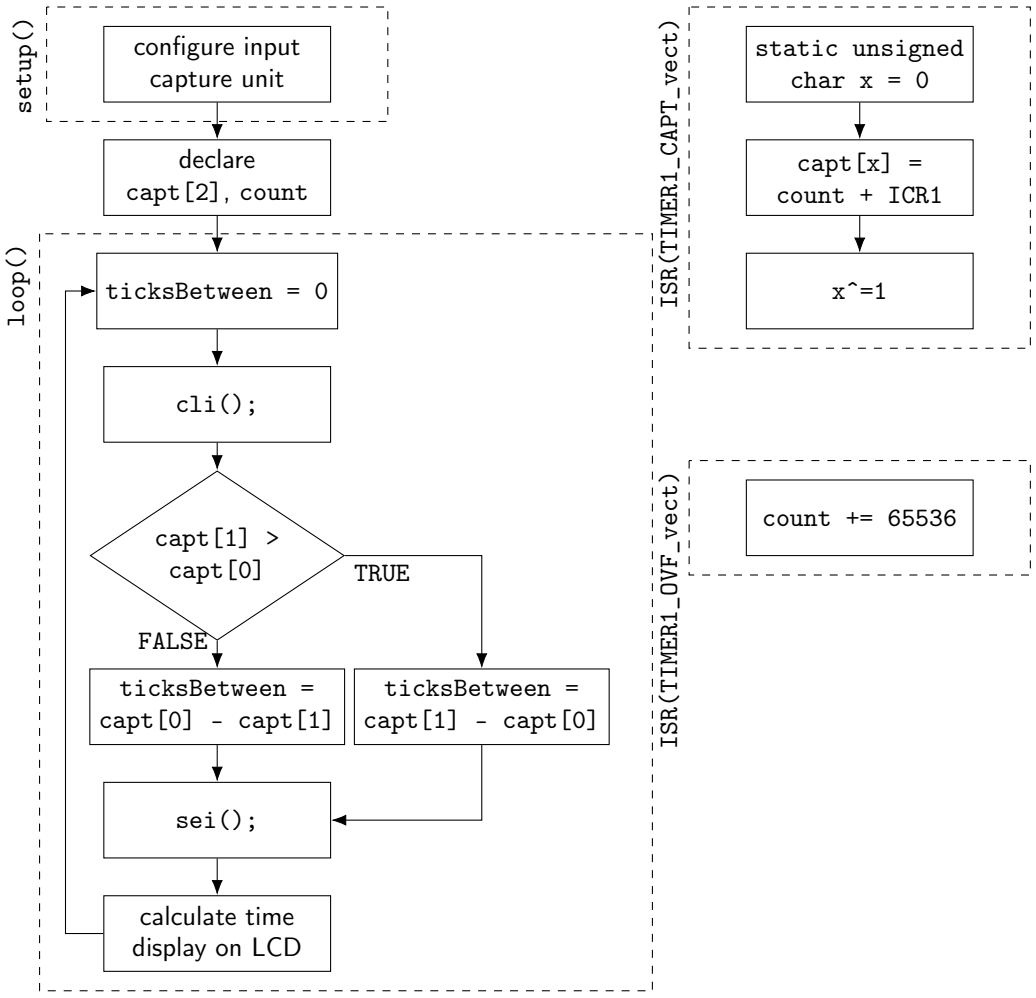
An ultrasonic detector is able to calculate the distance to the nearest object by emitting an ultrasonic pulse and calculating how much time t it takes for the pulse to return to the detector. If the speed of sound v is known, then the distance $d = vt$. After receiving a HIGH pulse of 10 μs to the `Trig` pin, the ultrasonic detector emits a pulse and then outputs a HIGH signal to the `Echo` pin. The HIGH signal is proportional to the distance of the nearest object. The datasheet for the ultrasonic sensor recommends a minimum 60 μs of delay between triggers on the sensor. The pinout diagram for the HC-SR04 is included in Appendix B.

Circuit I: Pulse-width measurement

This circuit will calculate the period of a square wave input to `ICP1` and display the value with units of μs to an LCD screen.

Write code that displays the period of a square wave (in units of μs) onto an LCD screen. The square wave will come from a function generator (use the TTL/CMOS output pin) and will go into the `ICP1` pin AND to an oscilloscope. To ensure that the pulse-width measurements are valid for as long as possible, ensure that the variable `count` (as depicted in the flowchart) is of type `unsigned long`. Do not use any floating-point math in this circuit! Vary the frequency of the function generator waveform and ensure that your LCD continues to give the correct period.

You will build off of this hardware and software code to complete the next two circuits, so do not take it apart yet! When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

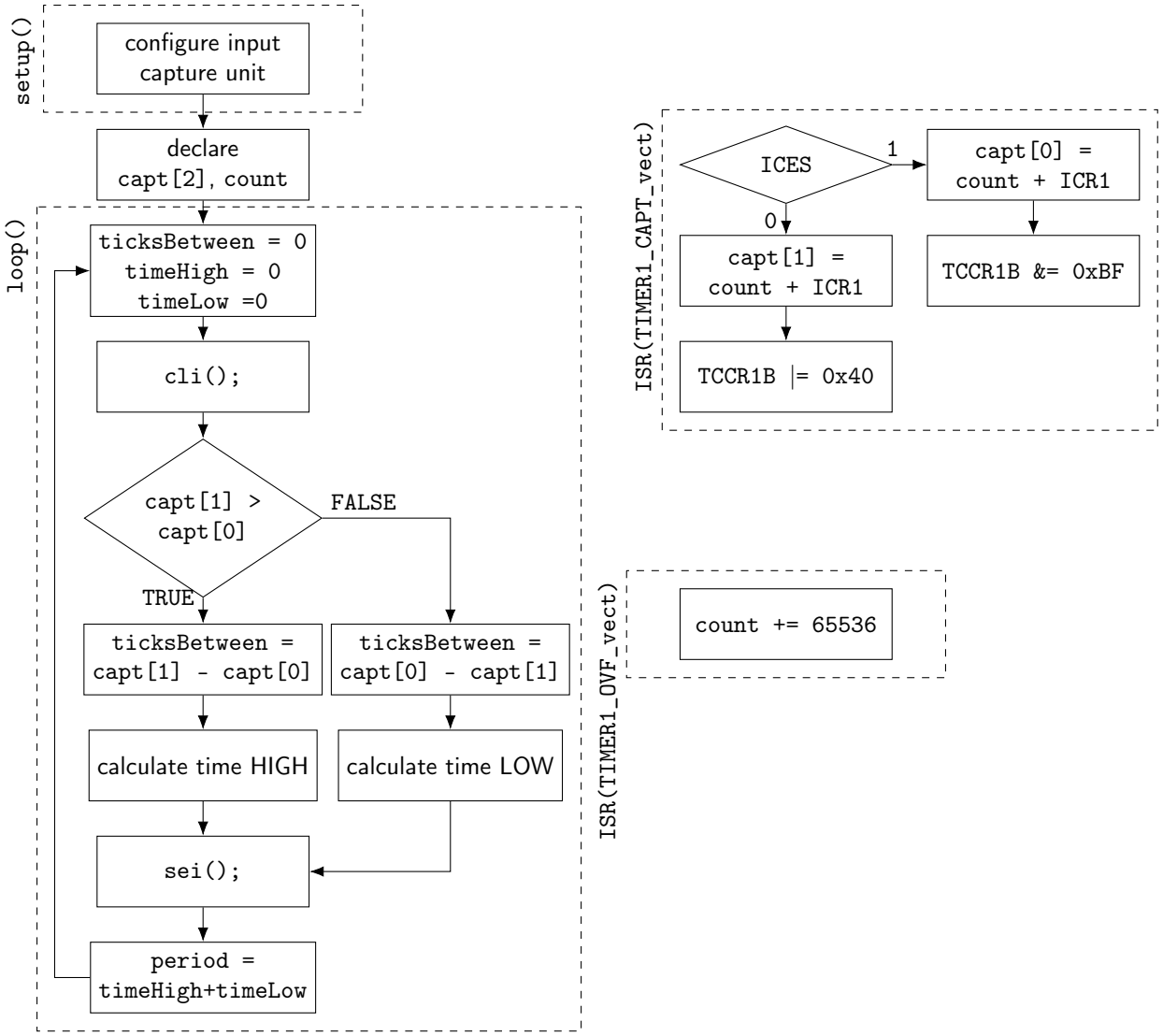
Instructor Stamp: _____

Circuit II: Advanced pulse-width measurement

This circuit will calculate the period, high period, and low period of a square wave input to `ICP1` and display the values with units of μs to an LCD screen.

Amend the code from Circuit I to display the period of the signal in addition to the time that the signal is HIGH and the time that the signal is LOW. The LCD screen should show all times in μs (display units, if they fit). Vary the frequency and duty cycle of the waveform and ensure that your LCD continues to give the correct times.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: Distance measurement

This circuit will use an ultrasonic sensor to calculate the distance to the nearest object and display the value with units of cm and tenth's place precision to an LCD screen.

Connect the ultrasonic detector so that the `Trig` pin is connected to an output pin and the `Echo` pin is connected to `ICP1` (pin D8) AND an oscilloscope. Calculate the time that the pulse from the `Echo` pin is HIGH. Then, use the following code to determine the distance in centimeters.

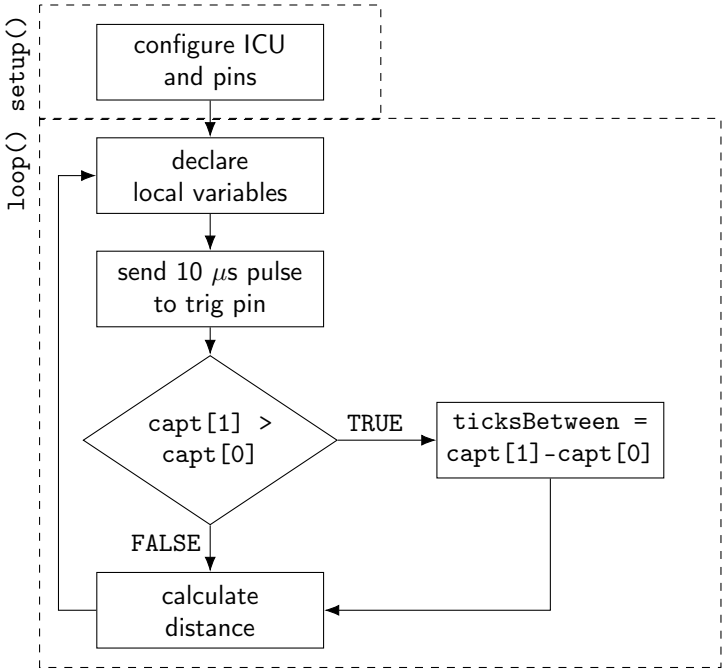
```
distance = ticksBetween * 34LL / 125; // distance = cm
```

Display the distance to the nearest object **in cm** on an LCD screen using tenth's place precision. Use an oscilloscope to ensure that a minimum of 60 μs elapses between triggers on the ultrasonic sensor. Insert a delay as needed to satisfy this requirement.

Use a ruler to verify distances. You will be asked in the lab report to comment on the accuracy of your ultrasonic sensor in a quantitative manner. Compare the measured and calculated distance of an object from your sensor at various ranges (close-range, long-range, medium-range, etc).

Note that to save space, the flow chart below does not show the following things: configuring global variables, masking interrupts during conditional logic (alternatively: saving and restoring `SREG` in the ISR), the input capture unit ISR, and any commands relating to the LCD screen.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 12 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- How accurate was the HC-SR04 ultrasonic detector? Be quantitative.
- What is the shortest distance that you can theoretically measure with the ultrasonic sensor as configured in this lab? (Refer to your code and the sensor datasheet to help determine this value.)
- What is the longest distance that you can theoretically measure with the ultrasonic sensor as configured in this lab? (Refer to your code and the sensor datasheet to help determine this value.)
- How might you integrate using the ultrasonic detector in your Smart Car?
- Give one advantage of the ultrasonic detector over using wheel encoders or the accelerometer.

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 13

Carefully read the entirety of Lab 13, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. In this lab, will you use polling or interrupts to determine the status of a pushbutton? How do you know?

2. List all of the AVR assembly instructions that work directly with I/O registers. (These registers include, but are not limited to, the `PINx`, `PORTx`, and `DDRx` registers. A full listing of I/O registers is available in Appendix G.)

3. In Circuit II, you will be asked to set or clear LED anodes based on the status of three pushbuttons. What I/O instructions can be used to conditionally move around in code if a single one of the three pushbuttons is pressed / not pressed?

Lab 13: Introduction to Assembly

In this lab, assembly language will be used to accomplish I/O tasks on the Arduino Uno.

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

13.1 Assembly program flow

Unlike the Arduino IDE, which executes code as described in the flowchart in Figure 1.2, to write code in assembly is to create subsequent instructions that will be stored in memory. In the absence of any jump or branch instructions, the first instruction will be executed, and then the next, and the next, until the end of the program has been reached, at which point the program will stop.

To create a program flow that contains setup instructions and then indefinitely loops through a set of instructions to be repeated, the JMP instruction can be used. The JMP instruction will change locations in memory without returning to the original location again.

The following assembly code can be used to imitate the program flow of the Arduino IDE.

```
; list any configuration instructions here

main:
; list any repeated instructions here
JMP main
```

13.2 Subroutines in assembly

In assembly, just as with C, any piece of code that needs to be regularly accessed should be stored as its own subroutine. The process for writing a subroutine in assembly is different than C. Unlike a `JMP` instruction that unconditionally moves to another location in memory, a subroutine is accessed using a `CALL` instruction. When complete, use the `RET` instruction to return back to the original memory location again.

The following code shows how a subroutine can be used in assembly.

```
main:
 ; do a bunch of things
 CALL subRoutine
 ; do a bunch of other things
 JMP main

subRoutine:
 ; do a common task
 RET
```

13.3 Microchip Studio

Microchip Studio will be used to write assembly code and upload it onto the Arduino. In order to accomplish this, an external tool must be generated in the Microchip Studio software in order for it to be able to communicate with the Arduino. Information on configuring this is given on the resources page on GitHub.

13.3.1 Creating and running a program

To create a program in Microchip Studio, do not just double-click on a file and open it. You will have to create an assembly file so that the assembler can properly assemble the code into machine language. In Microchip Studio, select File > New > Project. Select Assembler, and give your file a name and save location. In the Device Selection window, type in ATmega328P. This will create the new file.

Write your code as needed in the file labeled main.asm. When you are ready to run the code, first build the project by selecting Build > Build Solution, or using the hotkey F7. Check for any compiler errors. If there are no errors, you can upload your code to the Arduino using the external tool you configured using the instructions provided on the class GitHub page.

13.3.2 Determining program memory usage

In Microchip Studio, it is possible to determine how much program memory was used by your program. However, it is not as simple as it was in the Arduino IDE. On the right-hand side of the screen in Microchip Studio, there should be a box labeled "Solution Explorer." If you click on that box, you will find a folder called "Output Files." Click on that folder and find a file called `circuitname.lss` Click on that file and scroll all the way to the bottom of the page. Find a line that looks like the one shown below.

```
1 [.cseg] 0x000000   0x000QRS   XY   0   XY   32768   z%
```

The beginning statement `[.cseg]` indicates that the program memory (code segment) space is being explored. The first hex number is the address of the first location used in the program memory. The second hex number is the address of the last location used. The numerals that are noted as `XY` will be actual numbers; those show the number of bytes of program memory that the code has used. The number `32768` is the capacity of the flash memory. The percentage is what percent of program memory that your code is using.

13.3.3 Determining data memory usage

It is not as simple to calculate the amount of data memory used. Every time data is written to memory (which will not be done in this lab, but may be done in future assembly labs), one byte of information is stored in data memory. Therefore, to determine how much data memory has been used, keep track of how many writes to SRAM are made in the assembly code.

Circuit I: Pushbutton and LED

> When a pushbutton is pressed, an LED will light up. Otherwise, the LED will remain off.

Download the file `lab13_circuit1.asm`. Connect a pushbutton (with 10 kΩ pull-down resistor) and LED (with current-limiting resistor) to the appropriate pins as dictated in the file comments. The LED should light when the pushbutton is pressed.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Pushbuttons and RGB LED

> This circuit will turn on each anode of an RGB LED when the corresponding pushbutton is pressed. When the pushbutton is not pressed, the anode will turn off.

Now, connect three pushbuttons to pins of your choice. Use an RGB LED with 3 current-limiting resistors connected between each anode and their Arduino pins, and the cathode connected to ground. Write assembly code where if one pushbutton is pressed, the red LED lights, if another is pressed, the green LED lights, if the third is pressed, the blue LED lights, and any combination of presses results in secondary colors or white.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: 7-Segment display

> This circuit will display decimal characters 0–9 on a 7-segment display. Every 500 ms, the value on the display will increment. After displaying 9, the display will cycle back to 0 again.

Use assembly language to display numerals 0–9 on a 7-segment display (as you have done multiple times using parallel and serial I/O already using C code). After each number is displayed, include a delay of approximately 500 ms, using the following website to determine what code to include: `http://darcy.rsgc.on.ca/ACES/TEI4M/AVRdelay.html`

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 13 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- How much program memory is used for each of your circuits?
- Explain one benefit of using assembly over using C code
- Explain one benefit of using C code over using assembly

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 14

Carefully read the entirety of Lab 14, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. Is the USART synchronous or asynchronous?

2. How many pins does the USART use? Specifically, which pins on the ATmega328P does it use?

3. `UBRR0` is a 12-bit register. What are the largest and smallest possible baud rates? Show all of your work.

4. Using asynchronous normal mode (as you will use in circuits 1 and 2), what value do you need to store in `UBRR0` to achieve a baud rate of 9600? Show all of your work.

5. What is the ASCII character for 'a' in **binary**? Use the website `http://www.ascii-code.com`

6. What is the order for configuring the USART in SPI mode? (There are three steps.)

7. In Circuit III, you will use a Baud rate of 1 Mbit/s. What should you set as the value of `UBRR0`?

8. In Circuit III, how should you configure `UCSR0B`?

9. In Circuit III, how should you configure `UCSR0C`?

Lab 14: USART: Universal Synchronous / Asynchronous Receiver / Transmitter

In this lab, the USART (Universal Synchronous / Asynchronous Receiver / Transmitter) unit on the ATmega328P will be introduced. **AVR Concepts:** USART and registers `UBRR0`, `UCSR0A`, `UCSR0B`, `UCSR0C` and `UDR0`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

14.1 USART on the ATmega328P

In this lab, the USART (which stands for universal synchronous / asynchronous receiver / transmitter) functionality of the ATmega328P microcontroller will be used. As the name implies, there is flexibility in the USART that allowed it to be run either synchronously or asynchronously. In asynchronous mode, `START` and `STOP` bits are used to determine when a message begins and completes. In synchronous mode, a clock signal is used. A third mode of operation of the USART is in SPI mode.

14.1.1 Protocol

The USART protocol on the ATmega328P specifies three pins that can be used depending on the mode of operation. It is possible to use only a single pin (if data is to be received only, or transmitted only, in asynchronous mode) or all three (if data is to be received and transmitted simultaneously in SPI mode). Each of the USART pins is defined in Table 14.1.

Name	Pin	Description
`RXD`	`D0`	Data input pin
`TXD`	`D1`	Data output pin
`XCK`	`D4`	Clock (SPI mode)

Table 14.1: Each of the logic signals used in AVR USART communication.

14.1.2 Receiving data

The USART checks for a received signal at every clock cycle of the microcontroller. In asynchronous communication (in which the USART acts similarly to a shift register), a baud rate is agreed upon by both devices beforehand. If the signal is LOW for a long enough time, it is registered as a `START` bit. (If, however, the signal was LOW for less than half of the bit rate, it is considered to be a spurious signal and ignored.)

After receiving the `START` bit, the remaining character is clocked in to the receive shift register and into `UDR0` to be sent to the data bus. A busy flag is set during this process to signal that the device is busy receiving data. The USART then signals that it has received new data by setting the USART Receive Complete flag.

14.1.3 Transmitting data

A message is deposited into the transmit shift register from the data bus. At this point, the USART generates a `START` bit, which is sent, followed by the data to be transmitted. If requested, a parity bit is sent, followed

by a STOP bit. During this process, a busy flag is set, signaling that the device is busy transmitting data. Once all data has been shifted out of the transmit buffer, the USART Transmit Complete flag will be set.

14.2 Modes of operation

There are three modes of operation for using the USART module.

- Asynchronous
- Synchronous
- SPI mode

Synchronous mode will not be used in this lab or discussed in this lab manual. Before discussing the other two modes, it is important to understand the distinction between baud rate and bit rate. When using the USART, the term baud rate is used.

14.2.1 Bit rate and baud rate

In serial communication, bit rate indicates how many bits of data are sent every second. However, in the USART, there are also STOP bits, START bits, and possibly parity bits. These bits do not contribute to the actual message being sent but are nonetheless important parts of the signal. These are called non-data bits. Baud rate refers to how many signal changes occur per second. A signal change could be a change in voltage, frequency, or phase. Because binary data is used in microcontroller circuits, only voltage can change in a signal, and the baud rate will be equal to the bit rate.

The baud rate specifies the data transmission / receive rate on the USART when used in asynchronous mode. The baud rate is a function of the microcontroller clock frequency, the value stored in the baud rate register UBRR0, and the mode (normal mode, or double speed mode).

14.2.2 Asynchronous mode

Asynchronous mode allows the USART protocol to send and/or receive messages at a defined baud rate without using an external clock signal. The USART in this capacity can be operated in full-duplex mode, although it is possible to operate it simplex mode if desired. Equations for baud rate for asynchronous mode are given in the course textbook.

To use the USART in asynchronous mode, the following steps are to be taken (in order).

1. Initialize the USART by setting the baud rate first using UBRR0.
2. Then configure the control and status registers.

14.2.3 SPI mode

SPI mode allows the USART protocol to function in synchronous primary mode, with a clock supplied on the XCK pin. This is a full-duplex protocol. Note that it is not possible to address independent secondary devices without using external signals to select between the different secondary devices. Equations for the baud rate in SPI mode are given in the course textbook.

To use the USART in SPI mode, the following steps are to be taken (in order).

1. First, configure the clock pin as an output pin.
2. Then, configure the control and status registers for the USART.
3. Finally, write the baud rate value to the UBRR0 register.

14.3 USART registers

There are several registers that control the operation of the USART module. For more detailed information about each register, refer to Appendix A.

- `UBRR0H` & `UBRR0L` – **USART Baud Rate Registers**: These registers contain the baud rate to be used in the USART module. Two registers are required to store the value because it is a 12-bit value.
- `UCSR0A` – **USART Control and Status Register A**: This register stores information about how the USART is to be used. It is used in conjunction with `UCSR0B` and `UCSR0C`.
- `UCSR0B` – **USART Control and Status Register B**: This register stores information about how the USART is to be used. It is used in conjunction with `UCSR0A` and `UCSR0C`.
- `UCSR0C` – **USART Control and Status Register C**: This register stores information about how the USART is to be used. It is used in conjunction with `UCSR0A` and `UCSR0B`.
- `UDR0` – **USART I/O Data Register**: This register contains either information that was received from the USART in receive mode, or contains information to be transmitted out by the USART in transmit mode.

Circuit I: USART data transmission

This circuit will will send the character 'a' to the serial monitor. An oscilloscope will be connected to the `TXD` pin to view the signal transmission.

Using the USART in asynchronous normal mode with a baud rate of 9600, transmit the 8-bit character 'a' to the `TXD` pin. Use two `STOP` bits. Display the signal on an oscilloscope. There should be no delay in your loop function. To confirm that this is working, you should be able to see the character 'a' printed in the serial monitor as well.

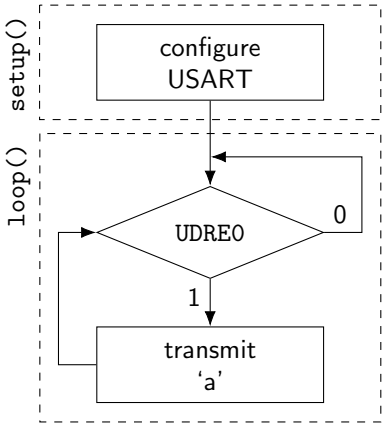
Before sending data through the transmit pin, you must ensure that the data register is empty. Remember that there is a flag for this in register `UCSR0A`!

In a separate file, use `Serial.begin(9600)` in the setup and `Serial.print('a')` in the loop and record the program and data memory used in that situation in Table 14.2. (This is the memory we would consume by using the Arduino "cheater functions" instead of writing our own code by configuring the USART manually.)

Memory	Bytes
Program	
Data	

Table 14.2: Program and data memory consumption using `Serial` commands.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

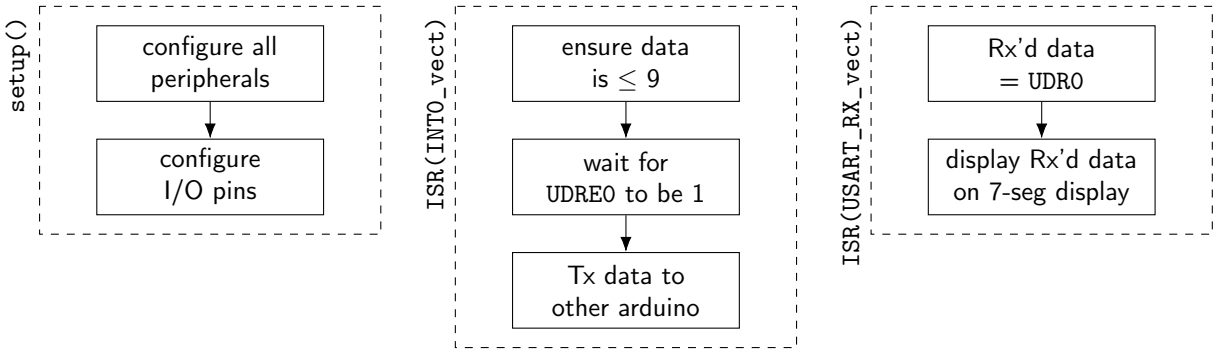
Circuit II: Arduino-Arduino communication

This circuit will use two Arduinos connected using the USART. A keypad connected to the first Arduino will light a 7-segment display accordingly on the second Arduino. A DIP switch connected to the second Arduino will light a 7-segment display accordingly on the first Arduino.

You will now connect together two Arduinos, with pin D0 on one connected to D1 on the other, and vice versa. Arduino 1 will contain a keypad and a 7-segment display. Arduino 2 will contain a DIP switch, a debounced pushbutton, and a 7-segment display. Pressing a button on the keypad (rising edge on the data pin) will transmit that number to Arduino 2, which will then show up on its 7-segment display. Pressing the pushbutton (rising edge) on Arduino 2 will send the value of the DIP switch (which should only be a 4-bit number) to Arduino 1, which will then show up on its 7-segment display. Ensure that the numeral being transmitted is less than or equal to 9, so that it can show up on a single display.

Both Arduinos will use asynchronous normal mode with equal baud rates. Use external interrupts to trigger the transmission of data over the USART. Use the `USART_RX_vect` interrupt to receive data. Use any method that you want to display the data on the 7-segment displays. Both Arduinos have identical flowcharts, their hardware will be different.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Arduino 1

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Arduino 2

Program Memory: _____ bytes

Data Memory: _____ bytes

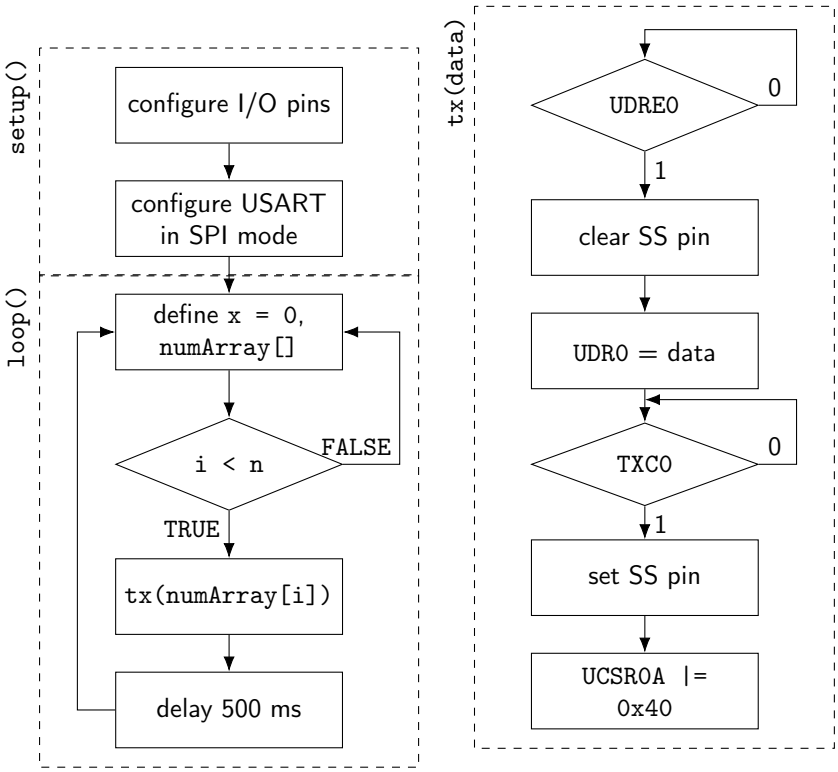
Instructor Stamp: _____

Circuit III: Using the USART in SPI mode

This circuit will display decimal characters 0–9 on a 7-segment display. Every 500 ms, the value on the display will increment. After displaying 9, the display will cycle back to 0 again.

Repeat Circuit I from Lab 9 using the USART in SPI mode, rather than the dedicated SPI module. Because the USART does not have support for a secondary select pin, you will choose any I/O pin to act as an active-LOW secondary select signal. Clear the SS pin to enable data transmission before writing to `UDR0`, then wait until the transmission is complete (by checking the `TXC0` flag in the `UCSR0A` register) before setting the SS pin again. Note: the `TXC0` flag will **not** automatically reset itself after data transmission is complete. In order to clear the flag, write a one to the `TXC0` bit.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 14 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Based on your calculations of memory usage in Circuit I, which is a more efficient means of writing data to the serial monitor: using the USART registers or by using the Arduino "cheater" functions of `Serial.print`?
- Based on the binary value of the ASCII character 'a', and the oscilloscope trace when 'a' was sent through the USART, how is data sent through the USART? MSB first or LSB first? How do you know?
- Based on the value you stored in UBRR0, what is the actual baud rate of Circuit I and Circuit II? Using 9600 baud as your accepted value, calculate the percent error.
- How did you choose to display the received data onto the 7-segment display in Circuit II? Consider all parallel and serial output options, and determine if there could have been a better way to send data to the display using fewer I/O pins.
- Compare Circuit III in this lab to Circuit I in Lab 9. What are the benefits of using the dedicated SPI module? What are the benefits of using the USART in SPI mode?

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 15

Carefully read the entirety of Lab 15, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. At what memory address can data start being written to SRAM?

2. What is the highest allowable memory address at which data can be written to SRAM?

3. Where are the ADC registers located, in the I/O register space, or in the extended I/O register space?

4. Based on the answer to the above question, can instructions such as `IN` and `OUT` be used to write to the ADC status and control registers and read from the ADC result registers?

5. In Circuit II and Circuit III, you will have to wait a certain number of ADC clock cycles to initialize the ADC and to complete a conversion on the ADC. Convert the following ADC clock cycles into either CPU clock cycles or a period of time consistent with the CPU clock and ADC prescaler.

   (a) 25 ADC clock cycles.

   (b) 13 ADC clock cycles.

6. Use the question setup for Circuit II and Circuit III to help determine the values of the following registers. Note that if the value of a register is `0x00`, then that register does not have to be configured in code.

   (a) What will be the value of `ADCSRA` when you wish to initialize the ADC but not start a conversion?

   (b) What will be the value of `ADCSRA` when you are ready to start a conversion?

   (c) What will be the value of `ADCSRB`?

   (d) What will be the value of `ADMUX`?

Lab 15: Pointers and ADC in Assembly

In this lab, the pointer register `X` will be used to write data to different displays. This will demonstrate the use of indirect addressing and accessing the data memory. In addition, the ADC will be configured and used with AVR assembly. This lab is intended to build off of the knowledge introduced in Lab 13. **AVR Concepts:** pointer registers, data memory

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

15.1 Pointer registers

There are three pointer registers in the general-purpose register space in SRAM on the ATmega328P. These registers are 16-bit address pointers used for indirect addressing of the data memory space. Because each register is 16 bits wide, it requires two general purpose registers in data memory. These registers are listed in Table 15.1 for each of the pointer registers.

Pointer Register	High Byte	Low Byte
`X` register	r27	r26
`Y` register	r29	r28
`Z` register	r31	r30

Table 15.1: Register locations of each of the pointer registers.

The contents of each of the pointer registers can therefore be written directly by using load immediate instructions. Pointer register `X` will be used in this lab. The pointer registers can then be used to indirectly address the data space (which will be explained below).

15.2 SRAM data space

The SRAM on the ATmega328P is where temporary data is stored, as SRAM is a type of non-volatile memory. Although data memory has been used frequently in previous labs using C, in C the read/write process to data memory is a "behind-the-scenes" process. In assembly, it is necessary to have a larger understanding of data memory and how it can be written to in order to utilize it in code.

Because assembly requires such a "hands-on" process for writing to data memory, this provides the freedom to choose the exact addresses where data will be stored in memory. The data memory space of the ATmega328P is shown in Figure 15.1.

The lowest 256 addresses are reserved for general-purpose registers (including the pointer registers), I/O registers (such as the data direction, port, and pin registers), and extended I/O registers (including registers used to configure many of the peripheral features). In order to store data to the SRAM, the lowest address that can be used is `0x0100` to avoid overwriting any of the other registers.

Data memory can be addressed directly, by including the address of the stored data's memory location in the assembly instruction, or indirectly, by using a pointer register. Both direct and indirect addressing will be used in this lab.

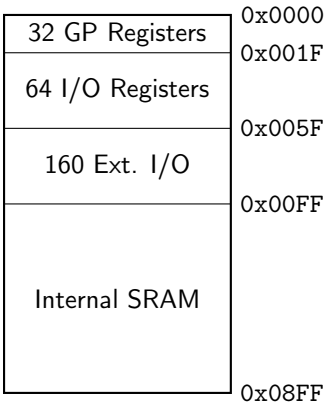


Figure 15.1: Data memory space of the ATmega328P microcontroller.

15.3 Data direct addressing

When configuring an I/O register, it is simplest to directly address these registers to write their configuration values. The two data direct addressing instructions in the AVR instruction set follow.

- `LDS` – Load direct from data space
- `STS` – Store direct to data space

15.4 Data indirect addressing

To access data outside of the GP and I/O registers, direct addressing can no longer be used. Because there are so many memory locations, the machine instruction must use several bits to point to the address in memory that needs to be read from or written to. This is called indirect addressing. Data indirect addressing occurs when the operand address is the contents of the `X`, `Y` or `Z` pointer register. This mode is used to access extended I/O registers or the internal SRAM.

Indirect addressing can also be necessary when using a variable to address data (i.e. reading data from an address that depends on the value of a variable), rather than reading/writing from/to a fixed location in memory.

The two data indirect addressing instructions that will be used in this lab follow.

- `LD` – Load indirect
- `ST` – Store indirect

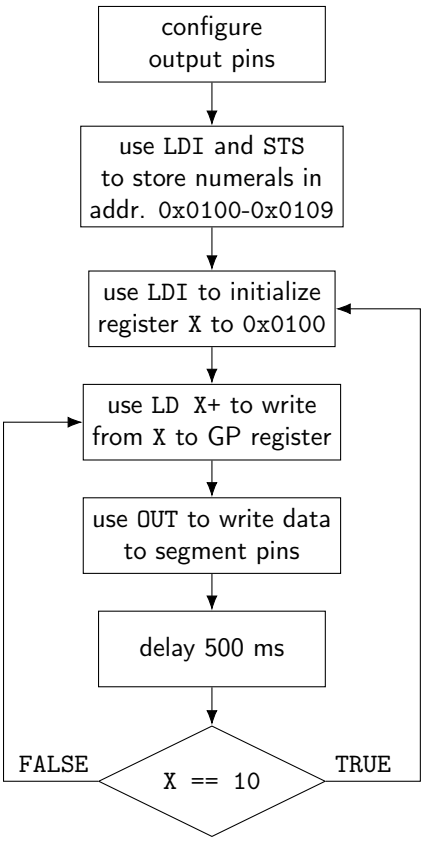
When many pieces of data are stored in subsequent memory addresses (such as an array would do in C), it can be very useful to use post-incrementing to avoid having to increment the pointer register after each use. Pre-decrementing is also a possibility in indirect addressing.

Circuit I: Writing numerals to 7-segment display

This circuit will display decimal characters 0–9 on a 7-segment display. Every 500 ms, the value on the display will increment. After displaying 9, the display will cycle back to 0 again.

Use data indirect addressing to write numeral encodings into data memory, and then display numerals 0–9 onto a 7-segment display with a 500 ms delay in between each numeral. At this point, you have created a similar piece of code in many other labs using both parallel and serial I/O in both C and assembly. This is another way of accomplishing the same objective. Use the following website for the code to use for a delay: `http://darcy.rsgc.on.ca/ACES/TEI4M/AVRdelay.html`.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit II: Conditional lighting of LEDs from the ADC

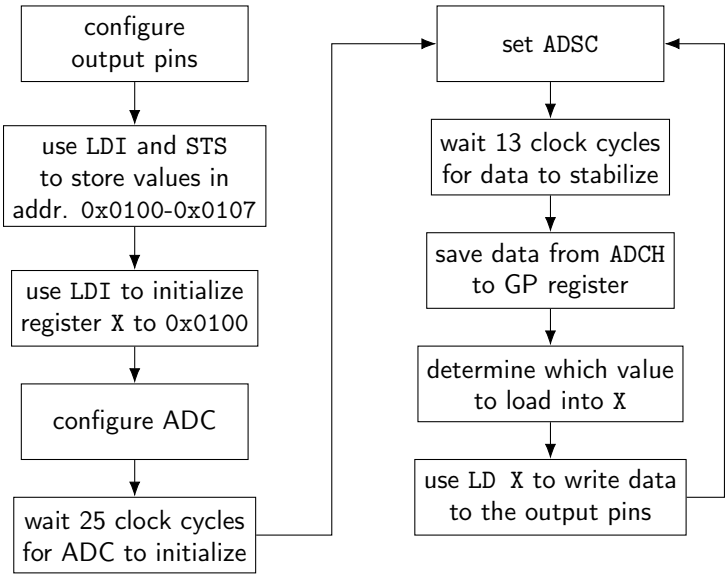
This circuit will light up one LED if the value from the ADC (connected to a potentiometer) is between 0–31; it will light two LEDs if the value from the ADC is between 32–63; it will light three LEDs if the value from the ADC is between 64–95; and so on until all eight LEDs will be lit if the value from the ADC is between 224–255.

Progressively light up a row of 8 LEDs based on the 8-bit ADC value. If the value is between 0–31, only one LED will light. If the value is between 32–63, then two LEDs will light. If the value is between 64–95, three LEDs will light. This continues until values between 224–255 will light up all eight LEDs.

The ADC will be configured with in 8-bit free-running mode with a prescaler of 128 and no auto-triggering. You will not be using the ADC interrupt in this lab.

When the ADC is initially configured (`ADEN` is set), the ADC requires at least 25 clock cycles to initialize the analog circuitry. (Note: remember that the ADC has a different prescaler than the rest of the circuit. Therefore, 25 ADC clock cycles is not equal to 25 CPU clock cycles!) After writing the start conversion bit (`ADSC`) to obtain a new value from the ADC, 13 ADC clock cycles are required to obtain the converted data. Therefore, you must wait 13 ADC clock cycles between writing `ADSC` and reading `ADCH`.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

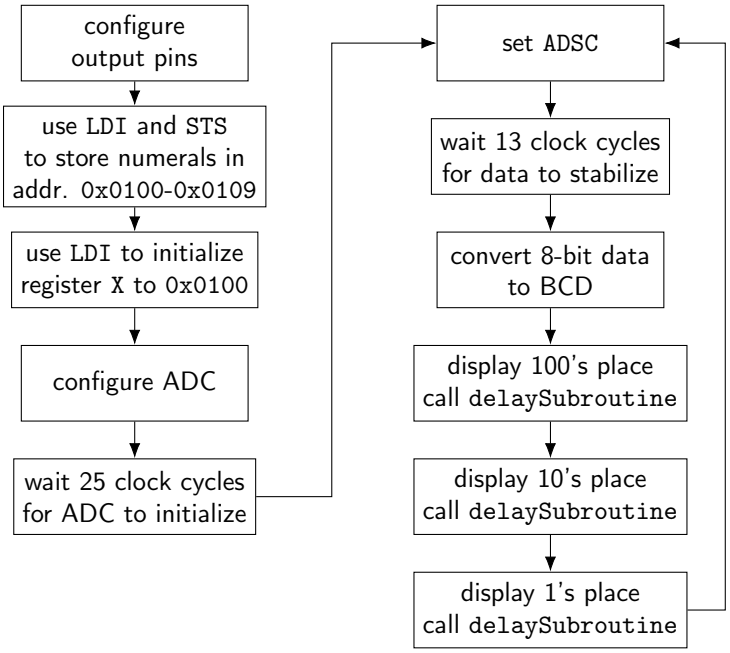
Instructor Stamp: _____

Circuit III: Displaying potentiometer value on MUX display

This circuit will display the value from the ADC (coming from a potentiometer) onto a MUX 7-segment display. As the potentiometer is rotated from one extreme to the next, the value on the display will vary from 0–255.

Display the 8-bit value from the ADC on a MUX 7-segment display. After writing to each cathode, call a subroutine to delay for 5 ms. The ADC will be configured exactly the same as in Circuit II.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 15 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- Compare the data and program memory used in Circuit I to that used in Lab 3, Circuit I.
- Compare the data and program memory used in Circuit III to that used in Lab 3, Circuit III. (If you did not finish that circuit, contact Dr. P to obtain average values of data and program memory from previous semesters.)

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 16

Carefully read the entirety of Lab 16, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. What memory address locations correspond to the following interrupts?
   (a) `RESET`?

   (b) `INT0`?

   (c) `ADC`?

   (d) `TIMER1_COMPA`?

2. Are interrupt pointers stored in data memory or program memory?

3. In Circuit II, fast PWM will be used to generate a frequency as close as possible to 1 kHz with timer/counter 0 counting to `MAX`.

   (a) What value prescaler is necessary to achieve this objective?

   (b) What is the actual frequency that corresponds to the use of this prescaler? Show all of your work.

4. In Circuit III, what value of `OCR1A` is required for 500 ms interrupts? Express this value in hexadecimal.

5. In Circuit IV, why use the `WDR` instruction before configuring the `WDTCSR` register?

6. Consult Lab 5, Circuit III. What are the two steps that are required to configure the `WDTCSR` register?

Lab 16: Interrupts and WDT in Assembly

In this lab, interrupts will be used in assembly to light an LED to different brightness levels or to blink an LED on and off. The watchdog timer will be used as well. A special emphasis will be on understanding the purpose of the status register of the microcontroller. **AVR Concepts:** `.org` directive, status register `SREG`

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

16.1 Interrupt review

External interrupts were the focus of Lab 5. In addition, software interrupts have been utilized in many labs. These include the ADC interrupt (Lab 4), timed interrupts (Lab 6), SPI interrupts (Lab 9), and USART interrupts (Lab 14). This lab will build off of that knowledge to implement interrupts in assembly language.

Each interrupt vector is given a program address on the microcontroller. At this program address, a pointer to a subroutine is given. That interrupt subroutine will then be accessed if the corresponding interrupt is invoked. Refer to Appendix C for a list of all of the interrupt vectors and program addresses.

If an interrupt is enabled, it is vitally important to have a valid pointer located at the appropriate program address. If there is an invalid pointer, then the program may not go to the correct location in program memory when an interrupt is invoked. (This is similar to enabling interrupts in C and then not writing an `ISR(INTERRUPT_vect)` function.)

Recall that interrupts are never invoked in software, they are automatically called in hardware when an event has occurred. However, the microcontroller needs to be informed when the interrupt is done being serviced. The following code shows how to return from interrupt instruction, using the `RETI` instruction.

```
1 InterruptServiceRoutine:
2  ; do whatever needs to be done in the ISR
3  RETI
```

16.2 `.org` directive

In order to write a pointer to the correct program address, the program counter needs to be told to move to a certain location in program memory. This is accomplished using the `.org` directive. Specifically, the microcontroller needs to be told what to do in case of a `RESET` interrupt, and also what to do in case of any of the interrupts that are specifically enabled in software.

The following lines of assembly code demonstrate how to initialize both the `RESET` and `INT0` interrupts.

```
1  .org 0x0000
2   RJMP setup
3  .org 0x0002
4   RJMP EXT_INT0
5
6  setup:
7   ; setup code goes here
8
9  loop:
10  ; repeating loop code goes here
11  JMP loop
```

```
12
13 ; INT0 vector
14 EXT_INT0:
15  ; code to be executed during an INT0 interrupt
16  RETI
```

16.3 Status register SREG

It is important to have a good understanding of the status register SREG when using assembly, especially with interrupts. The global interrupt flag that exists in SREG has previously been discussed in the context of disabling and enabling interrupts. However, there are many bits in SREG that control the flow of software. For example, when compare or branch instructions are used in assembly, flags in SREG are checked to determine whether or not to branch to a subroutine.

Because an interrupt can be invoked in between a compare instruction and a branch instruction, when a particular value stored in SREG is of vital importance, it is necessary to store the value of SREG at the start of all interrupt service routines, and then restore that data before returning back to the regular code. An example follows.

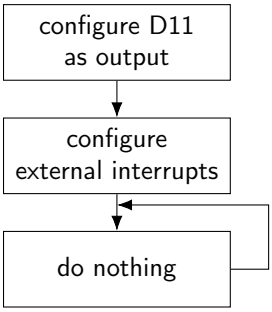
```
1 INTERRUPT_SUBROUTINE:
2  IN r15, SREG ; save the contents of SREG in a GP register
3  ; do whatever needs to be done in the ISR
4  OUT SREG, r15 ; restore the original contents of SREG
5  RETI
```

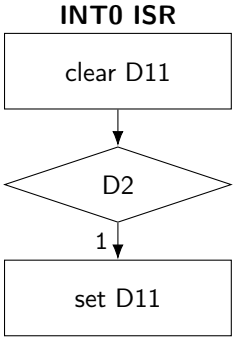
Circuit I: Pushbutton LED toggle with interrupts

When a pushbutton is pressed, an LED will light up. Otherwise, the LED will remain off.

Download the file `lab16_circuit1.asm`. Connect a pushbutton (with 10 kΩ pull-down resistor) to pin D2. Connect an LED (with current-limiting resistor) to pin D11. The LED should light when the pushbutton is pressed.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.





Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

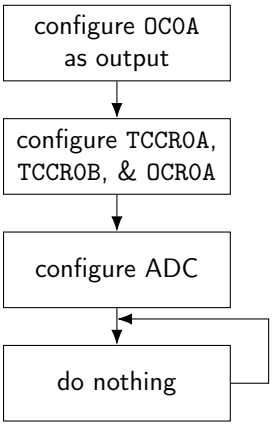
Circuit II: Variable intensity LED

This circuit will have a potentiometer control the brightness of an LED using PWM. As the potentiometer is rotated from one extreme to the next, the LED will go from being completely off to having full brightness.

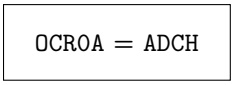
Use fast PWM counting to `MAX` at a frequency as close as possible to 1 kHz. You will use a potentiometer connected to the pin of your choice to control the brightness of an LED. As you rotate the potentiometer from one extreme to the other, the LED will either become brighter or dimmer.

Because you want to set the value of `OCR0A`, which is an 8-bit register, exactly equal to `ADCH`, use the ADC in 8-bit mode.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



ADC ISR



Program Memory: _____ bytes

Data Memory: _____ bytes

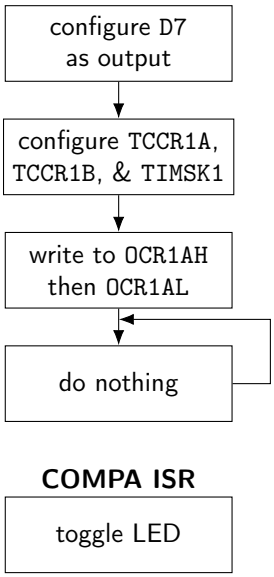
Instructor Stamp: _____

Circuit III: Timed blinking of LED

This circuit will blink an LED on and off at regular intervals.

Use timer/counter 1 to trigger an interrupt every 500 ms to toggle an LED located at pin D7. Do not use PWM for this circuit. It is important to note the order of the flowchart given. Configure the timer/counter 1 registers **before** configuring the output compare register A. When writing to a 16 bit register in assembly, it is imperative to write the high byte before writing to the low byte.

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

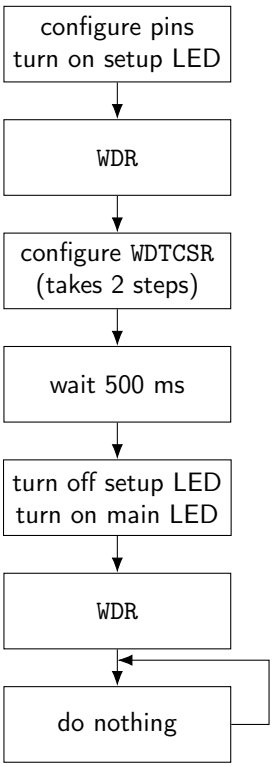
Instructor Stamp: _____

Circuit IV: Watchdog timer

> This circuit will flash a setup LED. Then, a second LED will be lit and the code will do nothing until the WDT is triggered and a reboot occurs.

Set up two different colored LEDs to different digital pins on the Arduino. Write code that sets up the WDT to timeout after 4 seconds. (It is important to write a `WDR` command before configuring the WDT to prevent any possible premature time-outs.) This circuit is going to be a little different from Lab 5 Circuit III. Rather than looping through increasingly large delay times, you will let the watchdog timer time-out by doing nothing. A video demonstration of this circuit is available on Dr. P's YouTube channel: `https://youtu.be/iWqjtBQRcCk`

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 16 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- (there are no special additions to make to this lab report)

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Pre-Lab 17

Carefully read the entirety of Lab 17, then answer the following questions. Attach a separate sheet of paper, if necessary, to show all work and calculations.

1. In Circuit I, register `r16`, which contains the contents of the `PIND` register, is complemented using the `COM` instruction. Why is this?

2. In Circuit I, how is the pin-change interrupt configured to update the LEDs only on rising edges of the pushbutton?

3. In Circuit I, what instruction is used to bitshift the contents of the `PIND` register?

4. In Circuit I, explain the purpose of the `ANDI r16, 0x3F` instruction that is included in the interrupt service routine.

5. In Circuit II, what is the largest possible value of the sum that can exist?

6. In Circuit III, the rolling sum of DIP switch values will be displayed onto LEDs. If the DIP switch uses 8 pins, and one pin is used for a pushbutton, how many pins are left over for LEDs?

7. As a consequence of the above question, what is the highest value that can be stored in the sum in order for the LED display not to overflow?

8. In Circuit IV, what is the highest value that can be stored in the sum?

Lab 17: Greater Than 8-Bit Math in Assembly

In this lab, addition will be performed on numbers that are 8 bits or larger, leading to math that requires more than one 8-bit register to store the result. Each circuit will add incrementally on to the last one, until a rolling sum of data from a DIP switch can be displayed onto a MUX display.

For lab resources and information, go to the following URL or scan the QR code. github.com/DoctorPCOD/DoctorPCOD/tree/main/labs

17.1 Greater than 8-bit operations

When writing C code, it is trivial to use datatypes such as `int` or `long` that store variables that take up more than 8 bits of space. However, the ATmega328P is an 8-bit microcontroller, so (most) all of the registers on the microcontroller are only capable of handling 8 bits of data at once. Therefore, to execute math on data that is greater than 8 bits, it needs to be broken up into pieces. The following examples are for 16-bit operations, but it is possible to scale these up if it is necessary to add data that has more than two bytes.

Each of the following examples in this lab introduction use the following general purpose registers for each of the following pieces of data.

```
; r16 = low byte A
; r17 = high byte A
; r18 = low byte B
; r19 = high byte B
; result is given in registers r16 (LOW) and r17 (HIGH)
```

17.1.1 Addition

Addition must be accomplished between two general purpose registers, as there are no immediate addition instructions. As with most 16-bit arithmetic, it is necessary to add the low bytes first. This is the same as with adding two numbers by hand: start with the least significant digit in case there are carries that cascade to subsequent digits. All of the carries internal to each byte is dealt with automatically by the microcontroller's hardware. However, the carry from the low byte needs to be added into the high byte. This is accomplished by using an add with carry instruction on the high byte.

```
ADD r16, r18   ; r16 <-- r16 + r18
ADC r17, r19   ; r17 <-- r17 + r19 + carry
```

17.1.2 Subtraction

Subtraction can be accomplished both using register addressing and immediate addressing. The register addressing steps are very similar to those used for addition. Recall that in subtraction, borrows are necessary. Just as with subtracting by hand, it is necessary to start with the lowest byte and then continue to the highest byte.

```
SUB r16, r18   ; r16 <-- r16 - r18
SBC r17, r19   ; r17 <-- r17 - r19 - borrow
```

It is also possible to use immediate subtraction instructions. In this case, a constant, immediate value can be subtracted from a GP register. It will be easiest to express this value as a hexadecimal number. The following example demonstrates the subtraction of 2000_{10} (`0x07D0`) from the data in registers `r16` and `r17`.

```
1 SUBI r16, 0xD0  ; r16 <-- r16 - 0xD0
2 SBCI r17, 0X07  ; r17 <-- r17 - 0x07 - borrow
```

17.1.3 Comparisons

Comparison between two 16-bit values can be accomplished using register addressing. This is very similar to the previous operations. After the compare with carry instruction, the branch instruction can be used.

```
1 CP r16, r18
2 CPC r17, r19
```

17.2 Toggling in assembly

At times, it is useful to store the two most recent pieces of data into a circular buffer. Rather than incrementing the array index (as is done in larger circular buffers), the index can be toggled using a bitwise XOR with the number one. This will toggle a variable between 0 and 1. While this may have been straightforward to do in C, it can still be accomplished in assembly, albeit in a very different manner.

The `T` flag in `SREG` can be toggled every time the condition is met to change the value of the circular buffer index. (Note that there is no toggle instruction; one of the objectives of Circuit II in this lab will be to determine how to accomplish this.) If the `T` flag is set, then data will be saved to a particular memory location; otherwise it will be saved to a different memory location.

Circuit I: Display DIP switch value on LEDs with pin-change interrupt

> This circuit will display the binary value of a DIP switch onto 8 LEDs. The LEDs will change their values only when a pushbutton has a rising-edge.

Download the file `lab17_circuit1.asm`. Figure 17.1 is a schematic for this circuit. It is important to note how the DIP switch is connected. There are no external resistors used because internal pull-ups are going to be activated. Use a debounced pushbutton (with pull-down resistor) connected to pin `A5` to trigger the input (this is not shown on the schematic below).

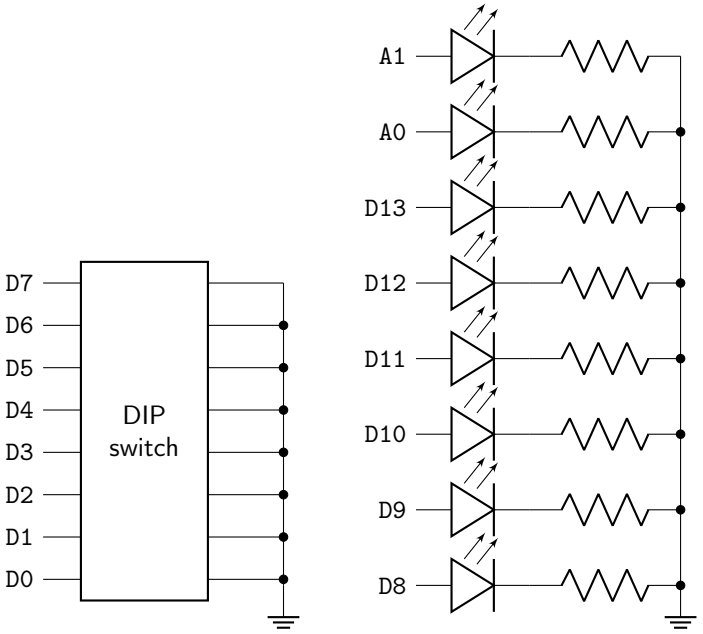


Figure 17.1: Circuit diagram for the DIP switch (left) and LEDs (right).

Do not take this circuit apart when you are done, as you will build on it for subsequent circuits! When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. You will **not** need to submit the software code.

Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

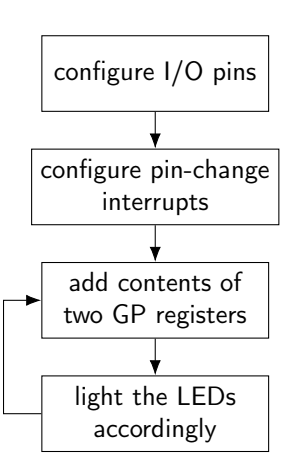
Circuit II: Add two most recent DIP switch values onto LEDs

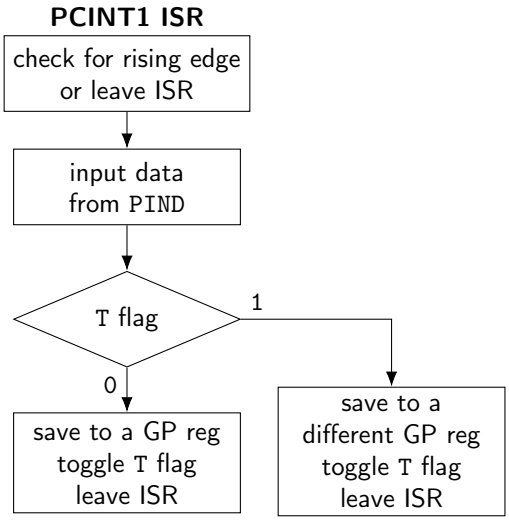
This circuit will display the sum of the two most recent DIP switch values in binary on LEDs. The sum value will update every time a button is pressed.

Build onto the previous circuit. You will need one additional LED to represent the 9-bit output. This circuit will add the most recent (stored when a pushbutton has a rising edge) two DIP switch values together, and display the binary value of the output onto the LEDs. This will **not** be a rolling sum, it will just be the sum of the most recent two DIP switch values.

Use the `T` flag in `SREG` to determine when you should store data to one GP register vs. another. Every time the button is pressed, toggle the `T` flag and if it is set, save the DIP switch value to one GP register and if it is cleared, save the DIP switch value to another GP register, and then add those two GP registers together in the looping section of the code. Note that there are no toggle instructions for the `T` flag so you will have to determine how to change the value from 0–1 or 1–0.

Do not take this circuit apart when you are done, as you will build on it for subsequent circuits! When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.





Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Circuit III: Add rolling DIP switch values onto LEDs

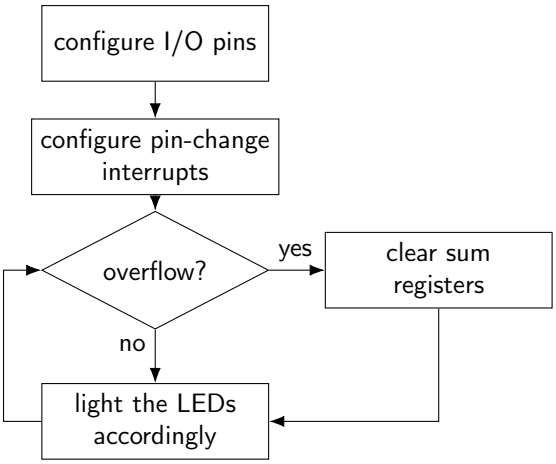
This circuit will display the rolling sum of DIP switch values in binary on LEDs. The sum value will update every time a button is pressed.

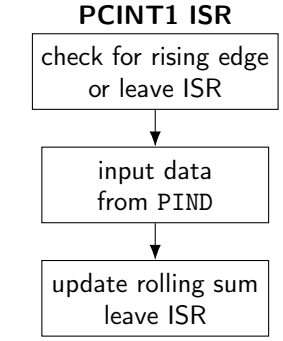
Build onto the previous circuit. You will now use as many LEDs as you can given the number of I/O pins that exist on the Arduino Uno board. What changes in this circuit is that you will create a rolling sum of all of the DIP switch values that are registered when the pushbutton has a rising edge.

Because you are always going to be storing new data into a single GP register, you no longer need to use the `T` flag of `SREG` to determine where to store the data, as no toggling will take place. The difficulty in this circuit is in dealing with numbers that are larger than 8 bits and in determining if the result data has overflowed (cannot be shown using the number of LEDs that you have available on your circuit).

If you would like to manually reset the sum back to zero at any time, simply press the reset button on the Arduino board. (This will save us from having to use another I/O pin on the Arduino and further limit our maximum stored value.)

Do not take this circuit apart when you are done, as you will build on it for subsequent circuits! When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.





Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

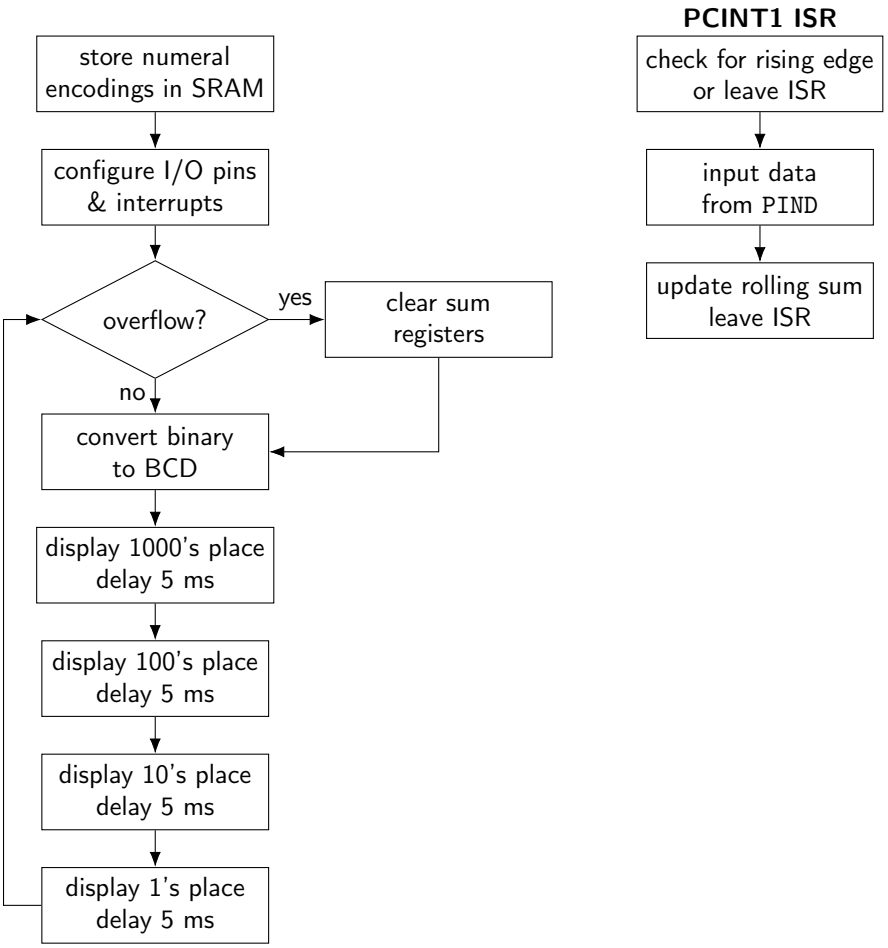
Circuit IV: Add rolling DIP switch values onto MUX display

This circuit will display the rolling sum of DIP switch values in decimal on a MUX 7-segment display. The sum value will update every time a button is pressed.

Build onto the previous circuit. You will now use a MUX display instead of LEDs. Now you can store a much larger rolling sum, as the MUX display is able to show numeral values up to 9,999. To convert your rolling sum to BCD values, use the code provided on GitHub to convert a number into four BCD values corresponding to thousands, hundreds, tens, and ones places. Use the pointer register X to display each numeral value as you did in Lab 15. The difficulty in this circuit is using the limited pins available on ports B and C to configure the seven segments and four cathodes.

If you would like to manually reset the sum back to zero at any time, simply press the reset button on the Arduino board. It is possible that this won't work. In that case, simply re-upload your code to reset the Arduino board. (This issue can be fixed if an external programmer is available.)

When the circuit is functioning properly, demonstrate it to your instructor to receive a stamp. Submit your software code as directed by your instructor.



Program Memory: _____ bytes

Data Memory: _____ bytes

Instructor Stamp: _____

Lab 17 Report

Include the following information in your lab report. Your lab report may be completed in any approved format listed in the syllabus. Take particular care to use accurate technical information (i.e. stay away from ambiguous or imprecise words such as "always", "best", "sort of", "several", "lots of", etc.). **Include headings for each section.**

The Writing, Reading, Speech Assistance Center at COD is a great resource if you are not comfortable with writing. Technical writing is an important aspect of being an engineer, whether or not you believe it to be true at this stage in your career.

Note that the use of ChatGPT or other AI or LLMs in your lab report must abide by the requirements listed in the class syllabus.

Introduction

Give an overview of the objectives of the labs and the important concepts that were covered. Use your own words – do **NOT** just copy the lab introduction.

Procedure and results

Explain how each of the circuits was coded, and what they accomplished. Ensure that your explanation explores both the software and hardware components to the overall functionality. Include the pins that were used on the Arduino, an explanation of all constants and derived terms, any libraries that were used, etc.

Based on the feedback you've received on your code, and anything new you've learned since the lab, what, if any, changes would you make to your code? (If you would not make any changes, please state that explicitly.)

In this lab report, specifically include the following information in the Procedure and results section:

- (there are no special additions to make to this lab report)

Circuit diagrams

Include circuit diagrams using an approved schematic software (hand-drawn schematics will not be accepted). Label each one with the corresponding circuit number(s). If any circuits had identical wiring, there is no need to include two copies.

Challenges

Briefly describe any challenges that you or your lab partner(s) encountered in the lab, and how you overcame them. If the challenges were not resolved, explain how you might prevent similar challenges from occurring in the future. If there were no significant problems, describe how you were able to work well as a team to accomplish that.

Conclusion

Wrap up all of the key concepts from your lab report in a paragraph.

Feedback (optional)

Include an optional section with feedback about the lab. What did you find useful? What was difficult to understand? What would you change? Were there any resources you wish you had to help you with the lab? Does this give you any ideas of things you'd like to learn about going forward?

Appendix A: Register and Fuse Descriptions

ADCH and ADCL – ADC Data Register

The AVR has a 10-bit ADC, so data needs to be split between two registers. Depending on if the data is left-adjusted or right-adjusted (as set in ADLAR), the data will be stored differently in the register.

ADLAR = 0

7	6	5	4	3	2	1	0
–	–	–	–	–	–	ADC9	ADC8
ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

ADLAR = 1

7	6	5	4	3	2	1	0
ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADC1	ADC0	–	–	–	–	–	–

ADCSRA – ADC Control and Status Register A

This register stores information about how the ADC is to be used. It is used in conjunction with ADCSRB.

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- ADEN – ADC enable. This bit must be set for the ADC to work.
- ADSC – ADC start conversion. This bit must be set for the ADC to start processing data.
- ADATE – ADC auto trigger enable. This bit must be set for the ADC to automatically process data.
- ADIF – ADC interrupt flag. This bit is automatically set when the ADC is finished converting and is automatically cleared upon servicing the corresponding interrupt service routine.
- ADIE – ADC interrupt enable. When set, this creates an interrupt condition every time the ADC is finished converting data. This interrupt will be used to obtain the result of the conversion.
- ADPS[2:0] – ADC prescaler select bits. This has to do with how much slower the ADC clock will be compared to the CPU clock.

ADPS2	ADPS1	ADPS0	**Prescaler**
0	0	0	$CLK_{CPU} \div 2$
0	0	1	$CLK_{CPU} \div 2$
0	1	0	$CLK_{CPU} \div 4$
0	1	1	$CLK_{CPU} \div 8$

1	0	0	$CLK_{CPU} \div 16$
1	0	1	$CLK_{CPU} \div 32$
1	1	0	$CLK_{CPU} \div 64$
1	1	1	$CLK_{CPU} \div 128$

ADCSRB – ADC Control and Status Register B

A single register is not large enough to contain all information about how to run the ADC, therefore this register stores additional information about how the ADC is to be used.

7	6	5	4	3	2	1	0
–	ACME	–	–	–	ADTS2	ADTS1	ADTS0

- `ACME` – Analog comparator multiplexer enable.
- `ADTS[2:0]` – ADC auto trigger source. These three bits are given values equal to the auto trigger source that is required, provided in the table below.

ADTS2	ADTS1	ADPT0	**Trigger Source**
0	0	0	Free running mode
0	0	1	Analog comparator
0	1	0	External interrupt request 0
0	1	1	Timer/counter 0 compare match A
1	0	0	Timer/counter 0 overflow
1	0	1	Timer/counter 1 compare match B
1	1	0	Timer/counter 1 overflow
1	1	1	Timer/counter 1 capture event

ADMUX – ADC Multiplexer Selection Register

This register contains information about the reference voltage to be used by the ADC, and which pin is to be used for data input.

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0

- `REFS[1:0]` – Reference selection bits. These bits set the reference voltage for the ADC. The possible voltage reference selections are given in the table. When using either `AVCC` or the internal 1.1 V reference, it is recommended to put an external capacitor between the `AREF` pin and ground if noise immunity is required.

REFS1	REFS0	**Voltage Reference**
0	0	AREF (VREF turned off)
0	1	AVCC
1	0	Reserved (not allowed)
1	1	Internal 1.1 V

- ADLAR – ADC left adjust result. This bit determines how the information for the 10-bit result is saved in the two 16-bit registers. If ADLAR is set, the result will be left-adjusted (i.e. the MSB will be saved in bit 7 of the high data register, and the LSB will be saved in bit 6 of the low data register). If ADLAR is clear, the result will be right-adjusted (i.e. the MSB will be saved in bit 1 of the high data register, and the LSB will be saved in bit 0 of the low data register).
- MUX[3:0] – Analog channel selection bits. The ADC can choose between several input pins from which to convert data. The MUX[3:0] bits act as the 4-bit control bits on the input voltage selection multiplexer. The possible input sources are given in the table below. If data needs to come from several different pins, these bit values will need to be changed correspondingly in the code.

MUX3	MUX2	MUX1	MUX0	**Input Voltage Selection**
0	0	0	0	ADC0
0	0	0	1	ADC1
0	0	1	0	ADC2
0	0	1	1	ADC3
0	1	0	0	ADC4
0	1	0	1	ADC5
0	1	1	0	ADC6
0	1	1	1	ADC7
1	0	0	0	Temperature Sensor
1	0	0	1	Reserved (not used)
1	0	1	0	Reserved (not used)
1	0	1	1	Reserved (not used)
1	1	0	0	Reserved (not used)
1	1	0	1	Reserved (not used)
1	1	1	0	1.1 V
1	1	1	1	0 V

CLKPR – Clock Prescale Register

This register configures the system clock prescaler value. It will affect every synchronous peripheral on the ATmega328P microcontroller. Setting the prescaler is a two step process. First, the `CLKPCE` bit must be set. Then, the register must be set equal to the corresponding value with `CLKPCE` clear and the `CLKPS` bits configured as needed.

7	6	5	4	3	2	1	0
CLKPCE	–	–	–	CLKPS[3:0]			

- `CLKPCE` – Clock prescaler change enable. When written to one, the clock prescaler can be configured within the next four clock cycles.
- `CLKPS[3:0]` – Clock prescaler select bits. These bits define the prescaler value used on the system clock. Any value can be written to these bits regardless of the `CKDIV8` fuse setting. The possible values are given below. Values `1001` – `1111` are reserved.

CLKPS3	CLKPS2	CLKPS1	CLKPS0	**Prescaler**
0	0	0	0	÷ 1
0	0	0	1	÷ 2
0	0	1	0	÷ 4
0	0	1	1	÷ 8
0	1	0	0	÷ 16
0	1	0	1	÷ 32
0	1	1	0	÷ 64
0	1	1	1	÷ 128
1	0	0	0	÷ 256

DDRB – Port B Direction Register

7	6	5	4	3	2	1	0
–	–	DDB[13:8]					

- `DDB[13:8]` – Refers to the data direction of digital pins 13–8. 0 (default) is input, 1 is output.

DDRC – Port C Direction Register

7	6	5	4	3	2	1	0
–	–	DDC[5:0]					

- `DDC[5:0]` – Refers to the data direction of analog pins 5–0. 0 (default) is input, 1 is output.

DDRD – Port D Direction Register

7	6	5	4	3	2	1	0
DDD[7:0]							

- `DDD[7:0]` – Refers to the data direction of digital pins 7–0. 0 (default) is input, 1 is output.

EICRA – External Interrupt Control Register A

This register stores information about how external interrupts should be triggered on pins D2 and D3.

7	6	5	4	3	2	1	0
–	–	–	–	ISC11	ISC10	ISC01	ISC00

- `ISC1[1:0]` – Interrupt sense control 1 bits 1 and 0. These two bits control how the interrupt on external pin `INT1` (D3) will be triggered. The four options are shown in the table below.
- `ISC0[1:0]` – Interrupt sense control 0 bits 1 and 0. These two bits control how the interrupt on external pin `INT0` (D2) will be triggered. The four options are shown in the table below.

Bit 1	Bit 0	Description
0	0	Calls the ISR when the value on the pin is low
0	1	Calls the ISR when the value on the pin changes
1	0	Calls the ISR when the value on the pin changes from 1–0
1	1	Calls the ISR when the value on the pin changes from 0–1

EIMSK – External Interrupt Mask Register

This register contains two bits to enable interrupts on pins D2 and D3.

7	6	5	4	3	2	1	0
–	–	–	–	–	–	INT1	INT0

- `INT1` – If this bit is set, interrupts will be enabled on external pin `INT1` (D3).
- `INT0` – If this bit is set, interrupts will be enabled on external pin `INT0` (D2).

Extended Fuse Byte

This fuse configures the brown-out detection unit.

7	6	5	4	3	2	1	0
–	–	–	–	–	BODLEVEL[2:0]		

- BODLEVEL[2:0] – Brown-out detection trigger level. Programming these bits changes the cutoff for brown-out detection on the microcontroller, according to the following table (which gives values for minimum, maximum, and typical brown-out detection values). Values 000 – 011 are reserved.

BODLEVEL2	BODLEVEL1	BODLEVEL0	V_{MIN} (V)	V_{TYP} (V)	V_{MAX} (V)
1	1	1	brown-out detection disabled		
1	1	0	1.7	1.8	2.0
1	0	1	2.5	2.7	2.9
1	0	0	4.1	4.3	4.5

High Fuse Byte

This fuse configures features that are relevant to programming and debugging the ATmega328P.

7	6	5	4	3	2	1	0
RSTDISBL	DWEN	SPIEN	WDTON	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST

- RSTDISBL – External reset disable. If this bit is programmed (value of 0), pin PC6 can be used as a general purpose I/O pin instead of a reset pin.
- DWEN – debugWIRE enable. If this bit is programmed (value of 0), the debugWIRE system will be enabled.
- SPIEN – Enable serial program and data downloading. When programmed (value of 0), SPI programming is enabled.
- WDTON – Watchdog timer always on. When programmed (value of 0), the watchdog timer is always on.
- EESAVE – EEPROM memory preserved through chip erase. When programmed (value of 0), memory in EEPROM is preserved when the data memory on the chip is rewritten.
- BOOTSZ[1:0] – Select boot size. These bits configure the size of the bootloader memory in flash. When not using a bootloader, configure these to use the least amount of space. These bits should be configured as given in the table below.
- BOOTRST – Select reset vector. If this bit is programmed (value of 0), program execution will start from the bootloader section of code.

BOOTSZ1	BOOTSZ0	**Bootloader Size**	**Application Memory**	**Bootloader Memory**
1	1	256 bytes	0x0000 – 0x3EFF	0x3F00 – 0x3FFF
1	0	512 bytes	0x0000 – 0x3DFF	0x3E00 – 0x3FFF
0	1	1024 bytes	0x0000 – 0x3BFF	0x3C00 – 0x3FFF
0	0	2048 bytes	0x0000 – 0x37FF	0x3800 – 0x3FFF

Low Fuse Byte

This fuse configures the clock source and startup time of the microcontroller.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CKDIV8	CKOUT	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0

- `CKDIV8` – Divide clock by 8. If this bit is programmed (0), the system clock is prescaled by a factor of 8. This can be changed in hardware with the `CLKPR` register.
- `CKOUT` – Clock output. If this bit is programmed (0), the clock is output on pin 0 of port B.
- `SUT[1:0]` – Select start-up time. These two bits control the start-up time of the ATmega328P microcontroller. Clock sources require a sufficiently high value of VCC before it starts oscillating, and it must oscillate a sufficient number of times before the clock can be considered stable. A time-out delay (t_{TOUT}) is used after device reset to ensure a sufficient value of VCC. The value of these bits must be considered carefully based on the clock source (see `CKSEL` bits, below) and whether or not the brown-out detection is used. (Using brown-out detection ensures there is a sufficient value of VCC and reduces the delay necessary at start-up.)
- `CKSEL[3:0]` – Select clock source. These four bits control the clock source used for the ATmega328P microcontroller. The clock options are given in the table below. All clock sources not explicitly labeled as "internal" require an external crystal or ceramic oscillator.

CKSEL[3:0]	**Clock Source**	**Frequency**
1111 – 1110	Low Power Oscillator	8.0 – 16.0 MHz
1101 – 1100		3.0 – 8.0 MHz
1011 – 1010		0.9 – 3.0 MHz
1001 – 1000		0.4 – 0.9 MHz
0111 – 0110	Full Swing Oscillator	0.4 – 20 MHz
0101 – 0100	Low Frequency Crystal Oscillator	32.768 kHz
0011	Internal 128 kHz RC Oscillator	128 kHz
0010	Calibrated Internal RC Oscillator	7.3 – 8.1 MHz
0001	Reserved	
0000	External Clock	0 – 20 MHz

The following values of `CKSEL0` and `SUT[1:0]` are used with both low power and full swing ceramic resonators and crystal oscillators. These clock sources must be connected between pins `XTAL2` and `XTAL1` with appropriate capacitance values as given by the ATmega328P datasheet.

`CKSEL0`	`SUT[1:0]`	**Oscillator Source**	**Conditions**	**Start-Up Time**	**t_{TOUT}**
0	00	Ceramic resonator	Fast start-up	258 CK	14 CK + 4.1 ms
0	01		Slow start-up		14 CK + 65 ms
0	10		BOD enabled	1K CK	14 CK
0	11		Fast start-up		14 CK + 4.1 ms
1	00		Slow start-up		14 CK + 65 ms
1	01	Crystal oscillator	BOD enabled	16K CK	14 CK
1	10		Fast start-up		14 CK + 4.1ms
1	11		Slow start-up		14 CK + 65 ms

The following values of `CKSEL0` and `SUT[1:0]` are used with 32.768 kHz watch crystal oscillators.

`CKSEL0`	`SUT[1:0]`	**Conditions**	**Start-Up Time**	**t_{TOUT}**
0	00	Fast start-up or BOD enabled	1K CK	4 CK
0	01	Slow start-up		4 CK + 4.1 ms
0	10	Stable frequency at start-up		4 CK + 65 ms
0	11	Reserved		
1	00	Fast start-up or BOD enabled	32K CK	4 CK
1	01	Slow start-up		4 CK + 4.1 ms
1	10	Stable frequency at start-up		4 CK + 65 ms
1	11	Reserved		

The following values of `SUT[1:0]` are used with both of the internal RC oscillators as well as with the external clock.

`SUT[1:0]`	**Conditions**	**Start-Up Time**	**t_{TOUT}**
00	BOD enabled	6 CK	14 CK
01	Fast start-up		14 CK + 4.1 ms
10	Slow start-up		14 CK + 65 ms
11	Reserved		

MCUCR – MCU Control Register

This register enables and disables configuration details on the microcontroller.

7	6	5	4	3	2	1	0
–	BODS	BODSE	PUD	–	–	IVSEL	IVCE

- BODS – BOD sleep. If the brown-out detection (BOD) is activated in the extended fuse byte, then this bit must be set to disable the brown-out detection unit during sleep mode (if desired). To disable BOD in sleep mode, first set both BODS and BODSE. In the next instruction, set BODS and clear BODSE.
- BODSE – BOD sleep enable. As explained above, this bit can be used to disable the BOD during sleep modes.
- PUD – Pull-up disable. When clear, pull-ups can be enabled. When set, pull-ups on all I/O ports will be disabled.
- IVSEL – Interrupt vector select. When clear, interrupt vectors are placed at the start of flash memory. When set, interrupt vectors are moved to the beginning of the boot loader section of flash memory. To perform a change in vector location, first set the IVCE bit. In the next instruction, write the desired value to IVSEL while writing a zero to IVCE. (Note that if the vector location is changed, the address data in Appendix C will no longer be valid.)
- IVCE – Interrupt vector change enable. This bit must be set to enable a change to IVSEL.

OCR0A – Timer/Counter 0 Output Compare Register A

This register contains an 8-bit value that is continuously compared with the value in TCNT0. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0A pin.

7	6	5	4	3	2	1	0
OCR0A[7:0]							

OCR0B – Timer/Counter 0 Output Compare Register B

This register contains an 8-bit value that is continuously compared with the value in TCNT0. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0B pin.

7	6	5	4	3	2	1	0
OCR0B[7:0]							

OCR1AH and OCR1AL – Timer/Counter 1 Output Compare Register A

These two registers contain a 16-bit value that is continuously compared with the value in TCNT1H and TCNT1L. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC1A pin.

7	6	5	4	3	2	1	0
OCR1A[15:8]							
OCR1A[7:0]							

OCR1BH and OCR1BL – Timer/Counter 1 Output Compare Register B

These two registers contain a 16-bit value that is continuously compared with the value in TCNT1H and TCNT1L. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC1B pin.

7	6	5	4	3	2	1	0
OCR1B[15:8]							
OCR0B[7:0]							

OCR2A – Timer/Counter 2 Output Compare Register A

This register contains an 8-bit value that is continuously compared with the value in TCNT2. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC2A pin.

7	6	5	4	3	2	1	0
OCR2A[7:0]							

OCR2B – Timer/Counter 2 Output Compare Register B

This register contains an 8-bit value that is continuously compared with the value in TCNT2. A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC2B pin.

7	6	5	4	3	2	1	0
OCR2B[7:0]							

PCICR – Pin Change Interrupt Control Register

This register contains three bits which dictate whether or not interrupts are enabled on each of the three I/O ports.

7	6	5	4	3	2	1	0
–	–	–	–	–	PCIE2	PCIE1	PCIE0

- PCIE2 – If set, interrupts will be enabled on I/O pins in PORTD.
- PCIE1 – If set, interrupts will be enabled on I/O pins in PORTC.
- PCIE0 – If set, interrupts will be enabled on I/O pins in PORTB.

PCMSK0 – Pin Change Mask Register 0

This register allows pin-change interrupts to be enabled on individual pins in PORTB when their respective bit locations are set.

7	6	5	4	3	2	1	0
PCINT[7:0]							

PCMSK1 – Pin Change Mask Register 1

This register allows pin-change interrupts to be enabled on individual pins in PORTC when their respective bit locations are set.

7	6	5	4	3	2	1	0
–	PCINT[14:8]						

PCMSK2 – Pin Change Mask Register 2

This register allows pin-change interrupts to be enabled on individual pins in PORTD when their respective bit locations are set.

7	6	5	4	3	2	1	0
PCINT[23:16]							

PINB – Port B Input Pins Address

7	6	5	4	3	2	1	0
–	–	PB[13:8]					

- `PB[13:8]` – Stores the value on digital input pins 13–8.

PINC – Port C Input Pins Address

7	6	5	4	3	2	1	0
–	–	PC[5:0]					

- `PC[5:0]` – Stores the value on analog input pins 5–0. (Digital mode only. ADC must be used to access analog values.)

PIND – Port D Input Pins Address

7	6	5	4	3	2	1	0
PD[7:0]							

- `PD[7:0]` – Stores the value on digital input pins 7–0.

PORTB – Port B Data Register

7	6	5	4	3	2	1	0
–	–	PORTB[13:8]					

- `PORTB[13:8]` – Used to set or clear digital output pins 13–8. Setting these pins in input mode activates the internal pull-up resistor.

PORTC – Port C Data Register

7	6	5	4	3	2	1	0
–	–	PORTC[5:0]					

- `PORTC[5:0]` – Used to set or clear analog output pins 5–0 (used digitally). Setting these pins in input mode activates the internal pull-up resistor.

PORTD – Port D Data Register

7	6	5	4	3	2	1	0
PORTD[7:0]							

- `PORTD[7:0]` – Used to set or clear digital output pins 7–0. Setting these pins in input mode activates the internal pull-up resistor.

PRR – Power Reduction Register

This register enables and disables peripheral devices on the microcontroller. When peripherals are disabled, less power is consumed, but functionality is reduced.

7	6	5	4	3	2	1	0
PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC

- `PRTWI` – Power reduction TWI. Writing a one to this bit shuts down the TWI (two wire interface) communication system by stopping the clock to the module.
- `PRTIM2` – Power reduction timer/counter 2. Writing a one to this bit shuts down the timer/-counter 2 module in synchronous mode.
- `PRTIM0` – Power reduction timer/counter 0. Writing a one to this bit shuts down the timer/-counter 0 module.
- `PRTIM1` – Power reduction timer/counter 1. Writing a one to this bit shuts down the timer/-counter 1 module.
- `PRSPI` – Power reduction SPI. Writing a one to this bit shuts down the SPI (serial peripheral interface) communication system by stopping the clock to the module.
- `PRUSART0` – Power reduction USART0. Writing a one to this bit shuts down the USART (universal synchronous/asynchronous receiver/transmitter) communication system by stopping the clock to the module.
- `PRADC` – Power reduction ADC. Writing a one to this bit shuts down the ADC (analog to digital converter). The ADC must be disabled before shut down.

SMCR – Sleep Mode Control Register

This register enables sleep modes on the microcontroller.

7	6	5	4	3	2	1	0
–	–	–	–	SM2	SM1	SM0	SE

- `SM[2:0]` – Sleep mode select. These bits select between the six available sleep modes. These modes are given in the table below.

SM2	SM1	SM0	**Sleep Mode**
0	0	0	Idle
0	0	1	ADC noise reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved (not allowed)
1	0	1	Reserved (not allowed)
1	1	0	Standby

1	1	1	Extended standby

- `SE` – Sleep enable. This bit must be set to enable the microcontroller to enter the sleep mode. When clear, the microcontroller will not enter a sleep mode.

SPCR – SPI Control Register

This register configures serial peripheral interface (SPI) communication on the microcontroller.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`SPIE`	`SPE`	`DORD`	`MSTR`	`CPOL`	`CPHA`	`SPR1`	`SPR0`

- `SPIE` – SPI interrupt enable. When set, allows interrupts to occur upon successful data transfer.
- `SPE` – SPI enable. When set, enables SPI communication.
- `DORD` – Data order. When clear, MSB is sent first. When set, LSB is sent first.
- `MSTR` – Primary/secondary select. When clear, the device is the secondary. When set, the device is the primary.
- `CPOL` – Clock polarity. When clear, the clock will be low when idle. When set, the clock will be high when idle.
- `CPHA` – Clock phase. When clear, data will be sampled on the leading clock edge. When set, data will be sampled on the trailing clock edge.
- `SPR[1:0]` – SPI clock rate. These two bits, when used with `SPI2X` in the `SPSR` register, controls the frequency of `SCK`. These rates are given in the table below.

`SPI2X`	`SPR1`	`SPR0`	**SCK Frequency**
0	0	0	$f_{OSC} \div 4$
0	0	1	$f_{OSC} \div 16$
0	1	0	$f_{OSC} \div 64$
0	1	1	$f_{OSC} \div 128$
1	0	0	$f_{OSC} \div 2$
1	0	1	$f_{OSC} \div 8$
1	1	0	$f_{OSC} \div 32$
1	1	1	$f_{OSC} \div 64$

SPDR – SPI Data Register

This register is a read/write register used for data transfer. Writing to the register initiates data transmission. Reading the register causes the shift register receive buffer to be read.

7	6	5	4	3	2	1	0
MSB							LSB

SPSR – SPI Status Register

This register configures serial peripheral interface (SPI) communication on the microcontroller.

7	6	5	4	3	2	1	0
SPIF	WCOL	–	–	–	–	–	SPI2X

- `SPIF` – SPI interrupt flag. This bit is set when serial transfer is complete.
- `WCOL` – Write collision flag. The WCOL bit is set if the SPI data register (`SPDR`) is written during a data transfer.
- `SPI2X` – SPI double speed bit. Doubles the bit rate. The data transfer frequencies are given in the table accompanying the `SPCR` register.

SREG – AVR Status Register

This register contains information pertaining to the result of the most recently executed arithmetic instruction.

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

- `I` – Global interrupt enable. Must be set for interrupts to be enabled. If cleared, interrupts are disabled.
- `T` – Bit copy storage. Can be used as source or destination for a single bit. Used by the `BLD` and `BST` instructions.
- `H` – Half carry flag. Is set when the half carry (carry out from the least-significant nibble) is 1.
- `S` – Sign bit. $S = N \oplus V$
- `V` – Two's complement overflow flag. Is set when there is an overflow.
- `N` – Negative flag. Is set when the result of the arithmetic instruction is negative.
- `Z` – Zero flag. Is set when the result of the arithmetic instruction is zero.
- `C` – Carry flag. Is set when there is a carry out from the most-significant bit.

TCCR0A – Timer/Counter 0 Control Register A

This register configures the mode of operation of TCNT0.

7	6	5	4	3	2	1	0
COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00

- COM0A[1:0] – Compare match output A mode. These bits control the behavior of the output compare pin OC0A (pin D6). These compare output modes are given in the tables below.

Compare Output Mode, non-PWM Mode

If waveform generation mode is configured as non-PWM, OC0A will function as follows.

COM0A1	COM0A0	**Description**
0	0	Normal port operation, OC0A disconnected
0	1	Toggle OC0A on compare match
1	0	Clear OC0A on compare match
1	1	Set OC0A on compare match

Compare Output Mode, Fast PWM Mode

If waveform generation mode is configured as fast PWM, OC0A will function as follows.

COM0A1	COM0A0	**Description**
0	0	Normal port operation, OC0A disconnected
0	1	WGM02 =0: Normal port operation, OC0A disconnected WGM02 =1: Toggle OC0A on compare match
1	0	Clear OC0A on compare match, set OC0A at BOTTOM (non-inverting mode).
1	1	Set OC0A on compare match, clear OC0A at BOTTOM (inverting mode).

Compare Output Mode, Phase Correct PWM Mode

If waveform generation mode is configured as phase correct PWM, OC0A will function as follows.

COM0A1	COM0A0	**Description**
0	0	Normal port operation, OC0A disconnected
0	1	WGM02 =0: Normal port operation, OC0A disconnected WGM02 =1: Toggle OC0A on compare match
1	0	Clear OC0A on compare match when up-counting, set OC0A on compare match when down-counting (non-inverting mode).
1	1	Set OC0A on compare match when up-counting, clear OC0A on compare match when down-counting (inverting mode).

- `COM0B[1:0]` – Compare match output B mode. These bits control the behavior of the output compare pin `OC0B` (pin D5). The output compare modes are given in the tables below.

Compare Output Mode, non-PWM Mode

If waveform generation mode is configured as non-PWM, `OC0B` will function as follows.

`COM0B1`	`COM0B0`	**Description**
0	0	Normal port operation, `OC0B` disconnected
0	1	Toggle `OC0B` on compare match
1	0	Clear `OC0B` on compare match
1	1	Set `OC0B` on compare match

Compare Output Mode, Fast PWM Mode

If waveform generation mode is configured as fast PWM, `OC0B` will function as follows.

`COM0B1`	`COM0B0`	**Description**
0	0	Normal port operation, `OC0B` disconnected
0	1	Reserved (not allowed)
1	0	Clear `OC0B` on compare match, set `OC0B` at BOTTOM (non-inverting mode).
1	1	Set `OC0B` on compare match, clear `OC0B` at BOTTOM (inverting mode).

Compare Output Mode, Phase Correct PWM Mode

If waveform generation mode is configured as phase correct PWM, `OC0B` will function as follows.

`COM0B1`	`COM0B0`	**Description**
0	0	Normal port operation, `OC0B` disconnected
0	1	Reserved (not allowed)
1	0	Clear `OC0B` on compare match when up-counting, set `OC0B` on compare match when down-counting (non-inverting mode).
1	1	Set `OC0B` on compare match when up-counting, clear `OC0B` on compare match when down-counting (inverting mode).

- `WGM0[1:0]` – Waveform generation mode. Combined with the `WGM02` bit found in the `TCCR0B` register, these bits control the counting sequence of the counter. The options are given in the table below.

`WGM02`	`WGM01`	`WGM00`	**Mode of Operation**	`TOP Value`
0	0	0	Normal	`0xFF`
0	0	1	PWM, Phase Correct	`0xFF`
0	1	0	CTC	`OCR0A`

0	1	1	Fast PWM	0xFF
1	0	0	Reserved (not allowed)	–
1	0	1	PWM, Phase Correct	OCR0A
1	1	0	Reserved (not allowed)	–
1	1	1	Fast PWM	OCR0A

TCCR0B – Timer/Counter 0 Control Register B

This register sets one of the three waveform generation mode bits for TCNT0 and selects the clock source and prescaler.

7	6	5	4	3	2	1	0
FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00

- FOC0x – Force output compare x. (x can be either A or B.) These bits must be cleared when using a PWM mode. When set, an immediate compare match is forced on the waveform generation unit.
- WGM02 – Waveform generation mode. See the description in register TCCR0A.
- CS0[2:0] – Clock select. These three bits select the clock source to be used, described below.

CS02	CS01	CS00	**Description**
0	0	0	No clock source (TCNT0 stopped)
0	0	1	$CLK_{CPU} \div 1$
0	1	0	$CLK_{CPU} \div 8$
0	1	1	$CLK_{CPU} \div 64$
1	0	0	$CLK_{CPU} \div 256$
1	0	1	$CLK_{CPU} \div 1024$
1	1	0	External clock on T0 pin. Clock on falling edge.
1	1	1	External clock on T0 pin. Clock on rising edge.

TCCR1A – Timer/Counter 1 Control Register A

This register configures the modality of TCNT1 and sets two of the four waveform generation mode bits.

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10

- COM1A[1:0] – Compare match output A mode. These bits control the behavior of the output

compare pin OC1A (pin D9). The compare output modes are given in the table below.

Compare Output Mode, non-PWM Mode

If waveform generation mode is configured as non-PWM, OC1A will function as follows.

COM1A1	COM1A0	**Description**
0	0	Normal port operation, OC1A disconnected
0	1	Toggle OC1A on compare match
1	0	Clear OC1A on compare match
1	1	Set OC1A on compare match

Compare Output Mode, Fast PWM Mode

If waveform generation mode is configured as fast PWM, OC1A will function as follows.

COM1A1	COM1A0	**Description**
0	0	Normal port operation, OC1A disconnected
0	1	WGM1[3:0] = 14 or 15: Toggle OC1A on compare match All other WGM settings: Normal port operation, OC1A disconnected
1	0	Clear OC1A on compare match, set OC1A at BOTTOM (non-inverting mode).
1	1	Set OC1A on compare match, clear OC1A at BOTTOM (inverting mode).

Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM

If waveform generation mode is configured as phase correct or phase and frequency correct PWM, OC1A will function as follows.

COM1A1	COM1A0	**Description**
0	0	Normal port operation, OC1A disconnected
0	1	WGM1[3:0] = 9 or 11: Toggle OC1A on compare match All other WGM settings: Normal port operation, OC1A disconnected
1	0	Clear OC1A on compare match when up-counting, set OC1A on compare match when down-counting (non-inverting mode).
1	1	Set OC1A on compare match when up-counting, clear OC1A on compare match when down-counting (inverting mode).

- COM1B[1:0] – Compare match output B mode. These bits control the behavior of the output compare pin OC1B (pin D10). The compare output modes are given in the table below.

Compare Output Mode, non-PWM Mode

If waveform generation mode is configured as non-PWM, OC1B will function as follows.

COM1B1	COM1B0	**Description**
0	0	Normal port operation, OC1B disconnected
0	1	Toggle OC1B on compare match
1	0	Clear OC1B on compare match
1	1	Set OC1B on compare match

Compare Output Mode, Fast PWM Mode

If waveform generation mode is configured as fast PWM, OC1B will function as follows.

0	0	Normal port operation, OC1x disconnected
0	1	Normal port operation, OC1B disconnected
1	0	Clear OC1B on compare match, set OC1B at BOTTOM (non-inverting mode).
1	1	Set OC1B on compare match, clear OC1B at BOTTOM (inverting mode).

Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM

If waveform generation mode is configured as phase correct or phase and frequency correct PWM, OC1B will function as follows.

0	0	Normal port operation, OC1B disconnected
0	1	Normal port operation, OC1B disconnected
1	0	Clear OC1B on compare match when up-counting, set OC1B on compare match when down-counting (non-inverting mode).
1	1	Set OC1B on compare match when up-counting, clear OC1B on compare match when down-counting (inverting mode).

- WGM1[1:0] – Waveform generation mode. Combined with the WGM1[3:2] bits found in the TCCR1B register, these bits control the counting sequence of the counter. The options are given in the table below.

WGM13	WGM12	WGM11	WGM10	**Mode of Operation**	TOP Value
0	0	0	0	Normal	0xFFFF
0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF
0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF
0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF
0	1	0	0	CTC	OCR1A
0	1	0	1	Fast PWM, 8-bit	0x00FF
0	1	1	0	Fast PWM, 9-bit	0x01FF
0	1	1	1	Fast PWM, 10-bit	0x03FF

1	0	0	0	PWM, Phase and Frequency Correct	ICR1
1	0	0	1	PWM, Phase and Frequency Correct	OCR1A
1	0	1	0	PWM, Phase Correct	ICR1
1	0	1	1	PWM, Phase Correct	OCR1A
1	1	0	0	CTC	ICR1
1	1	0	1	Reserved (not allowed)	–
1	1	1	0	Fast PWM	ICR1
1	1	1	1	Fast PWM	OCR1A

TCCR1B – Timer/Counter 1 Control Register B

This register sets two of the four waveform generation mode bits for TCNT1 and selects the clock source and prescaler.

7	6	5	4	3	2	1	0
ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10

- `ICNC1` – Input capture noise canceler. When set the noise canceler is activated. This filters the captured input values.
- `ICES1` – Input capture edge select. A capture event is triggered on a falling edge when clear, and on a rising edge when set.
- `WGM1[3:2]` – Waveform generation mode. See the description in register `TCCR1A`.
- `CS1[2:0]` – Clock select. These three bits select the clock source, described below.

CS12	CS11	CS10	**Description**
0	0	0	No clock source (TCNT1 stopped)
0	0	1	$\text{CLK}_{\text{CPU}} \div 1$
0	1	0	$\text{CLK}_{\text{CPU}} \div 8$
0	1	1	$\text{CLK}_{\text{CPU}} \div 64$
1	0	0	$\text{CLK}_{\text{CPU}} \div 256$
1	0	1	$\text{CLK}_{\text{CPU}} \div 1024$
1	1	0	External clock on `T1` pin. Clock on falling edge.
1	1	1	External clock on `T1` pin. Clock on rising edge.

TCCR1C – Timer/Counter 1 Control Register C

7	6	5	4	3	2	1	0
FOC1A	FOC1B	–	–	–	–	–	–

- `FOC1A` – In a non-PWM mode, when writing a logical one to this bit, an immediate compare match is forced on the waveform generation unit.
- `FOC1B` – In a non-PWM mode, when writing a logical one to this bit, an immediate compare match is forced on the waveform generation unit.

TCCR2A – Timer/Counter 2 Control Register A

This register configures the modality of TCNT2 and sets two of the three waveform generation mode bits.

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20

- `COM2x[1:0]` – Compare match output `x` mode. (`x` can be either A or B.) These bits control the behavior of the output compare pin `OC2x`. (Pin `OC2A` is D11, pin `OC2B` is D3.) The compare output modes are given in the table below.

Compare Output Mode, non-PWM Mode

If waveform generation mode is configured as non-PWM, `OC2x` will function as follows.

COM2x1	COM2x0	**Description**
0	0	Normal port operation, OC2x disconnected
0	1	Toggle OC2x on compare match
1	0	Clear OC2x on compare match
1	1	Set OC2x on compare match

Compare Output Mode, Fast PWM Mode

If waveform generation mode is configured as fast PWM, `OC2x` will function as follows.

COM2x1	COM2x0	**Description**
0	0	Normal port operation, OC2x disconnected
0	1	Reserved (not allowed)
1	0	Clear OC2x on compare match, set OC2x at BOTTOM (non-inverting mode).
1	1	Set OC2x on compare match, clear OC2x at BOTTOM (inverting mode).

Compare Output Mode, Phase Correct PWM Mode

If waveform generation mode is configured as phase correct PWM, OC2x will function as follows.

COM2x1	COM2x0	**Description**
0	0	Normal port operation, OC2x disconnected
0	1	Reserved (not allowed)
1	0	Clear OC2x on compare match when up-counting, set OC2x on compare match when down-counting (non-inverting mode).
1	1	Set OC2x on compare match when up-counting, clear OC2x on compare match when down-counting (inverting mode).

- WGM2[1:0] – Waveform generation mode. Combined with the WGM22 bit found in the TCCR2B register, these bits control the counting sequence of the counter. These options are given in the table below.

WGM22	WGM21	WGM20	**Mode of Operation**	TOP Value
0	0	0	Normal	0xFF
0	0	1	PWM, Phase Correct	0xFF
0	1	0	CTC	OCR0A
0	1	1	Fast PWM	0xFF
1	0	0	Reserved (not allowed)	–
1	0	1	PWM, Phase Correct	OCR0A
1	1	0	Reserved (not allowed)	–
1	1	1	Fast PWM	OCR0A

TCCR2B – Timer/Counter 2 Control Register B

This register sets one of the three waveform generation mode bits for TCNT2 and selects the clock source and prescaler.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20

- FOC2x – Force output compare x. (x can be either A or B.) These bits must be cleared when using a PWM mode. When set, an immediate compare match is forced on the waveform generation unit.
- WGM22 – Waveform generation mode. See the description in register TCCR2A.
- CS2[2:0] – Clock select. These three bits select the clock source to be used by TCNT2, with options described below. $\text{CLK}_{\text{T2S}} = \text{CLK}_{\text{CPU}}$ if used synchronously.

CS22	CS21	CS20	**Description**
0	0	0	No clock source (TCNT2 stopped)
0	0	1	$CLK_{T2S} \div 1$
0	1	0	$CLK_{T2S} \div 8$
0	1	1	$CLK_{T2S} \div 32$
1	0	0	$CLK_{T2S} \div 64$
1	0	1	$CLK_{T2S} \div 128$
1	1	0	$CLK_{T2S} \div 256$
1	1	1	$CLK_{T2S} \div 1024$

TCNT0 – Timer/Counter 0 Register

This register contains the current value of TCNT0.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TCNT0[7:0]							

TCNT1H and TCNT1L – Timer/Counter 1 Register

These registers contain the current value of TCNT1. Two registers are required to store the value because TCNT1 is a 16-bit counter.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TCNT1[15:8]							
TCNT1[7:0]							

TCNT2 – Timer/Counter 2 Register

This register contains the current value of TCNT2.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TCNT2[7:0]							

TIMSK0 – Timer/Counter 0 Interrupt Mask Register

This register configures TCNT0 interrupts.

7	6	5	4	3	2	1	0
–	–	–	–	–	OCIE0B	OCIE0A	TOIE0

- `OCIE0B` – TCNT0 compare match B interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT0 compare match B.
- `OCIE0A` – TCNT0 compare match A interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT0 compare match A.
- `TOIE0` – TCNT0 overflow interrupt enable. When set, and if the I-bit in `SREG` is set, an interrupt will be executed any time the timer/counter overflows.

TIMSK1 – Timer/Counter 1 Interrupt Mask Register

This register configures TCNT1 interrupts.

7	6	5	4	3	2	1	0
–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1

- `ICIE1` – TCNT1 input capture interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be triggered when an input is captured.
- `OCIE1B` – TCNT1 compare match B interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT1 compare match B.
- `OCIE1A` – TCNT1 compare match A interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT1 compare match A.
- `TOIE1` – TCNT1 overflow interrupt enable. When set, and if the I-bit in `SREG` is set, an interrupt will be executed any time the timer/counter overflows.

TIMSK2 – Timer/Counter 2 Interrupt Mask Register

This register configures TCNT2 interrupts.

7	6	5	4	3	2	1	0
–	–	–	–	–	OCIE2B	OCIE2A	TOIE2

- `OCIE2B` – TCNT2 compare match B interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT2 compare match B.
- `OCIE2A` – TCNT2 compare match A interrupt enable. When set, and if the I-bit in `SREG` is set, interrupts will be enabled on TCNT2 compare match A.
- `TOIE2` – TCNT2 overflow interrupt enable. When set, and if the I-bit in `SREG` is set, an interrupt will be executed any time the timer/counter overflows.

UBRR0H and UBRR0L – USART Baud Rate Registers

These registers contain the baud rate to be used in the USART module. Two registers are required to store the value because it is a 12-bit value.

7	6	5	4	3	2	1	0
–	–	–	–	UBRR0[11:8]			
UBRR0[7:0]							

UCSR0A – USART Control and Status Register A

This register stores information about how the USART is to be used. It is used in conjunction with UCSR0B and UCSR0C.

7	6	5	4	3	2	1	0
RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0

- RXC0 – USART receive complete. (Read-only) This bit is set when there is unread data in the data receive buffer.
- TXC0 – USART transmit complete. This bit is set when all data has been shifted out of the data transmit register UDR0.
- UDRE0 – USART data register empty. (Read only) This bit is set when the data transmit buffer UDR0 is empty and is ready to receive new data.
- FE0 – Frame error. (Read only) This bit is set if a STOP bit is not detected at the appropriate STOP time.
- DOR0 – Data OverRun. (Read only) This bit is set if there is an overrun error, which occurs when new data comes in before the last set of data was processed.
- UPE0 – USART Parity Error. (Read only) This bit is set if parity checking was enabled and there is a parity error.
- U2X0 – Double the USART Transmission Speed. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication. This bit should be 0 when using USART in synchronous mode.
- MPCM0 – Multi-processor Communication Mode. When set, this bit enables multi-processor communication mode. The multi-processor communication mode enables several secondary MCUs to receive data from a primary MCU.

USART in SPI mode

This register has different settings when using the USART in SPI mode.

7	6	5	4	3	2	1	0
RXC0	TXC0	UDRE0	–	–	–	–	–

- RXC0 – USART receive complete. (Read-only) This bit is set when there is unread data in the data receive buffer.

- `TXC0` – USART transmit complete. This bit is set when all data has been shifted out of the data transmit register `UDR0`.
- `UDRE0` – USART data register empty. (Read only) This bit is set when the data transmit buffer `UDR0` is empty and is ready to receive new data.

UCSR0B – USART Control and Status Register B

This register stores information about how the USART is to be used. It is used in conjunction with `UCSR0A` and `UCSR0C`.

7	6	5	4	3	2	1	0
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

- `RXCIE0` – RX Complete Interrupt Enable. Setting this bit enables the USART Rx complete interrupt.
- `TXCIE0` – TX Complete Interrupt Enable. Setting this bit enables the USART Tx complete interrupt.
- `UDRIE0` – USART Data Register Empty Interrupt Enable. Setting this bit enables the USART data register empty interrupt.
- `RXEN0` – Receiver Enable. Setting this bit enables the USART to receive data.
- `TXEN0` – Transmitter Enable. Setting this bit enables the USART to transmit data.
- `UCSZ02` – Character Size. Combined with `UCSZ0[1:0]` bits found in UCSR0C, these bits control the size of each character transmitted and received through the USART. These options are given in the table below.

UCSZ02	UCSZ01	UCSZ00	**Character Size**
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

- `RXB80` – Receive Data Bit 8. (Read only) When operating with 9 data bits, this bit contains the ninth bit of data received. Must be read before reading the low bits from `UDR0`.
- `TXB80` – Transmit Data Bit 8. When operating with 9 data bits, this bit contains the ninth bit of data to be transmitted. Must be written before writing the low bits to `UDR0`.

USART in SPI mode

This register has different settings when using the USART in SPI mode.

7	6	5	4	3	2	1	0
RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	–	–	–

- RXCIE0 – RX Complete Interrupt Enable. Setting this bit enables the USART Rx complete interrupt.
- TXCIE0 – TX Complete Interrupt Enable. Setting this bit enables the USART Tx complete interrupt.
- UDRIE0 – USART Data Register Empty Interrupt Enable. Setting this bit enables the USART data register empty interrupt.
- RXEN0 – Receiver Enable. Setting this bit enables the USART to receive data.
- TXEN0 – Transmitter Enable. Setting this bit enables the USART to transmit data.

UCSR0C – USART Control and Status Register C

This register stores information about how the USART is to be used. It is used in conjunction with UCSR0A and UCSR0B.

7	6	5	4	3	2	1	0
UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

- UMSEL0[1:0] – USART Mode Select. These bits control the modality of the USART. These options are given in the table below.

UMSEL01	UMSEL00	**Mode**
0	0	Asynchronous USART
0	1	Synchronous USART
1	0	Reserved
1	1	Primary SPI

- UPM0[1:0] – Parity Mode. These two bits control the parity generation (using the transmitter) and parity checker (using the receiver). These options are given in the table below.

UPM01	UPM00	**Parity Mode**
0	0	Parity disabled
0	1	Reserved
1	0	Enabled, even parity
1	1	Enabled, odd parity

- USBS0 – Stop Bit Select. This bit selects the number of stop bits to be inserted by the transmitter. If this bit is cleared (0), one stop bit will be inserted. If this bit is set (1), two stop bits will be inserted.
- UCSZ0[1:0] – Character Size. These two bits are used in conjunction with UCSZ02 which is located in UCSR0B.

- UCPOL0 – Clock Polarity. This bit is only used in synchronous mode. Write this bit to zero when asynchronous mode is used. In synchronous mode, the two options are given in the table below.

UCPOL0	**Transmitted Data Updates On...**	**Received Data Updates On...**
0	Rising edge of XCK0	Falling edge of XCK0
1	Falling edge of XCK0	Rising edge of XCK0

USART in SPI mode

This register has different settings when using the USART in SPI mode.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
UMSEL01	UMSEL00	–	–	–	UDORD0	UCPHA0	UCPOL0

- UMSEL0[1:0] – USART Mode Select. These bits control the functionality of the USART module. In SPI mode, both of these bits must be set. All options are given in the table below.

UMSEL01	UMSEL00	**Mode**
0	0	Asynchronous USART
0	1	Synchronous USART
1	0	Reserved
1	1	Primary SPI

- UDORD0 – Data Order. When set, the LSB will be transmitted first. When cleared, the MSB will be transmitted first.
- UCPHA0 – Clock Phase. Together with the clock polarity bit, this sets the SPI data mode and timing.
- UCPOL0 – Clock Polarity. This bit together with the clock phase bit sets the SPI data mode and timing. Options for these two bits are given in the table below.

UCPHA0	UCPOL0	**SPI Mode**	**Leading Edge**	**Trailing Edge**
0	0	0	Sample	Setup
0	1	1	Setup	Sample
1	0	2	Sample	Setup
1	1	3	Setup	Sample

UDR0 – USART I/O Data Register

This register contains either information that was received from the USART in receive mode, or contains information to be transmitted out by the USART in transmit mode.

The transmit buffer can only be written when the UDRE0 flag in the UCSR0A register is set. Data written to UDR0 when the UDRE0 flag is not set, will be ignored by the USART transmitter. When data is written to the transmit buffer, and the transmitter is enabled, the transmitter will load the data into the transmit shift register when the shift register is empty. Then the data will be serially transmitted on the TxD0 pin.

7	6	5	4	3	2	1	0
UDR0[7:0]							

WDTCSR – Watchdog Timer Control Register

This register contains information for enabling and configuring the watchdog timer (WDT). Configuring the WDT is a two-step process:

1. Using a bitwise OR operation, simultaneously set the WDT change enable bit (WDCE) and the WDT system reset enable bit (WDE).
2. In the command immediately following the above, using an assignment operation, set WDE and write the desired values of WDP[3:0] to the register. (Assign a zero to all other bits in the register.)

7	6	5	4	3	2	1	0
WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0

- WDIF – WDT interrupt flag. This bit is set when a WDT interrupt is triggered if the WDT interrupt mode is enabled.
- WDIE – WDT interrupt enable. When set, this bit allows WDT interrupts.
- WDCE – WDT change enable. This bit must be set to configure the WDT.
- WDE – WDT system reset enable. When set, this bit allows WDT resets.
- WDP[3:0] – WDT prescaler bits. By changing these bits, the clock frequency of 128 kHz is divided by a value, causing timeouts to occur at different intervals as defined in the table below.

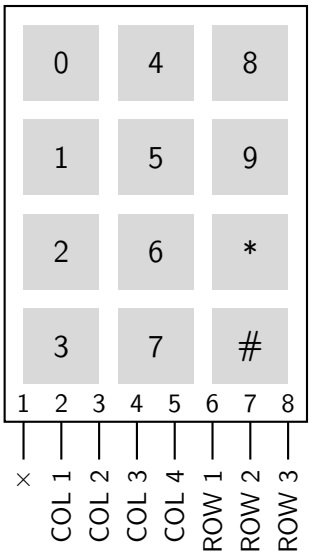
WDP3	WDP2	WDP1	WDP0	**Prescaler**	**Time-Out Interval**
0	0	0	0	2,048	16 ms
0	0	0	1	4,096	32 ms
0	0	1	0	8,192	64 ms
0	0	1	1	16,384	0.125 s
0	1	0	0	32,768	0.25 s
0	1	0	1	65,536	0.5 s

0	1	1	0	131,072	1.0 s
0	1	1	1	262,144	2.0 s
1	0	0	0	524,288	4.0 s
1	0	0	1	1,048,576	8.0 s

Appendix B: Pinout Diagrams

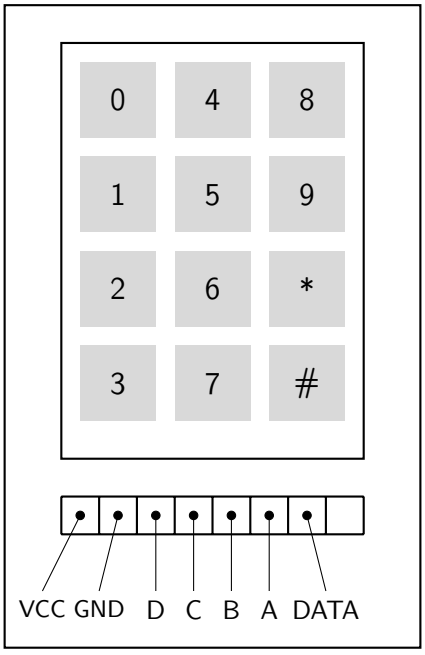
Digital Input Devices

12-Character Keypad

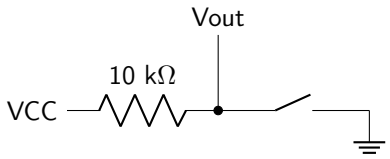


12-Character Keypad PCB

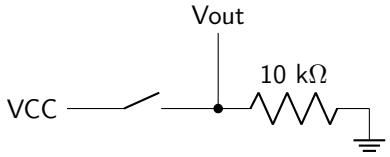
The following PCB has been wired up to accept the 12-character keypad data in asynchronous mode and output a BCD value `DCBA`.



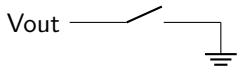
DIP Switch, External Pull-Up Resistor (Active LOW)



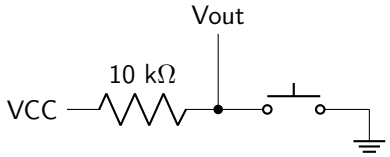
DIP Switch, External Pull-Down Resistor (Active HIGH)



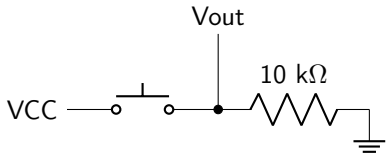
DIP Switch, Internal Pull-Up Resistor (Active LOW)



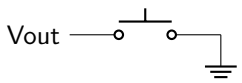
Pushbutton, External Pull-Up Resistor (Active LOW)



Pushbutton, External Pull-Down Resistor (Active HIGH)

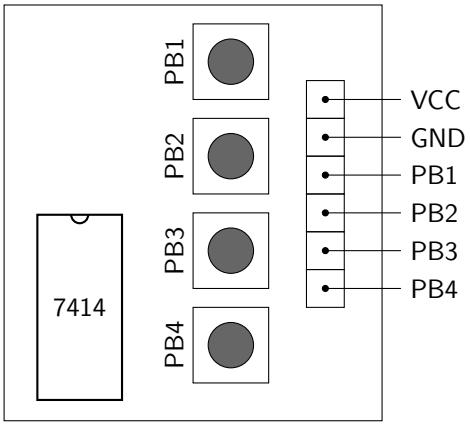


Pushbutton, Internal Pull-Up Resistor (Active LOW)

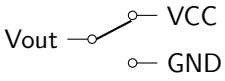


Debounced Pushbutton PCB

The following PCB has been wired up with four debounced pushbuttons.

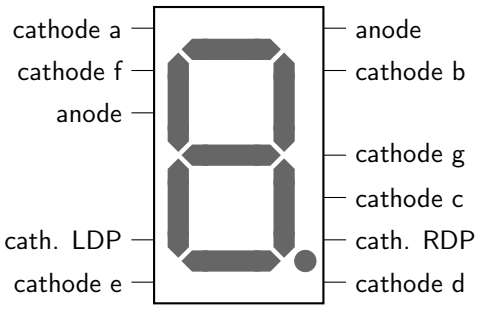


Toggle Switch

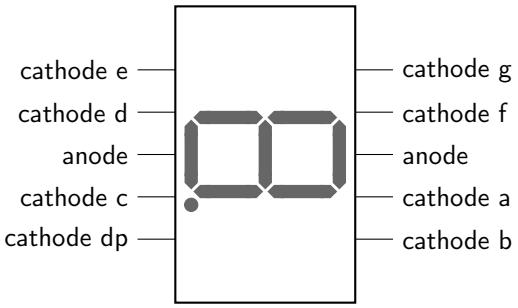


Displays

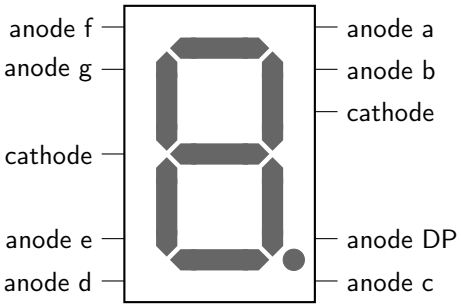
Common Anode 7-Segment Display



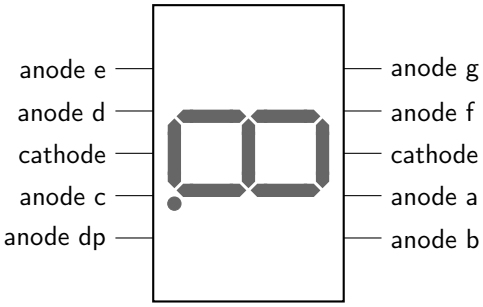
Sideways Common Anode 7-Segment Display



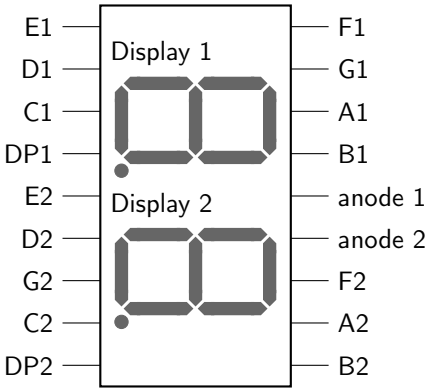
Common Cathode 7-Segment Display



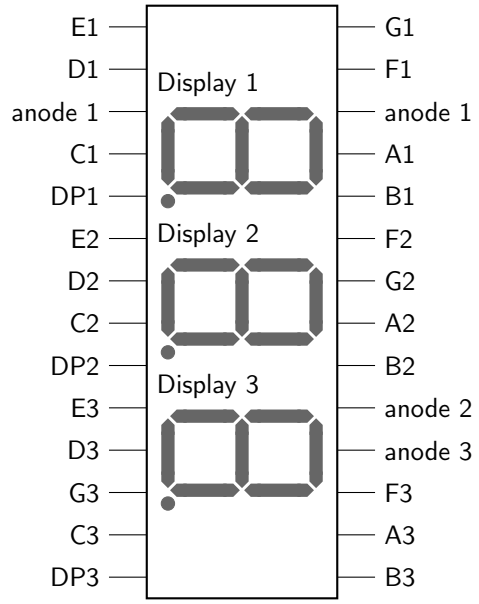
Sideways Common Cathode 7-Segment Display



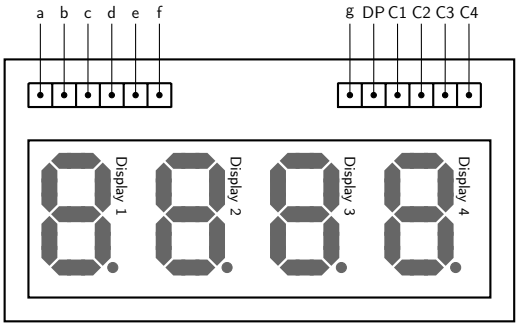
Dual Common Anode 7-Segment Display



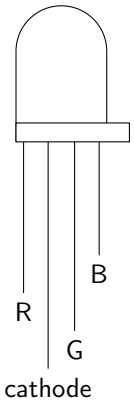
Triple Common Anode 7-Segment Display



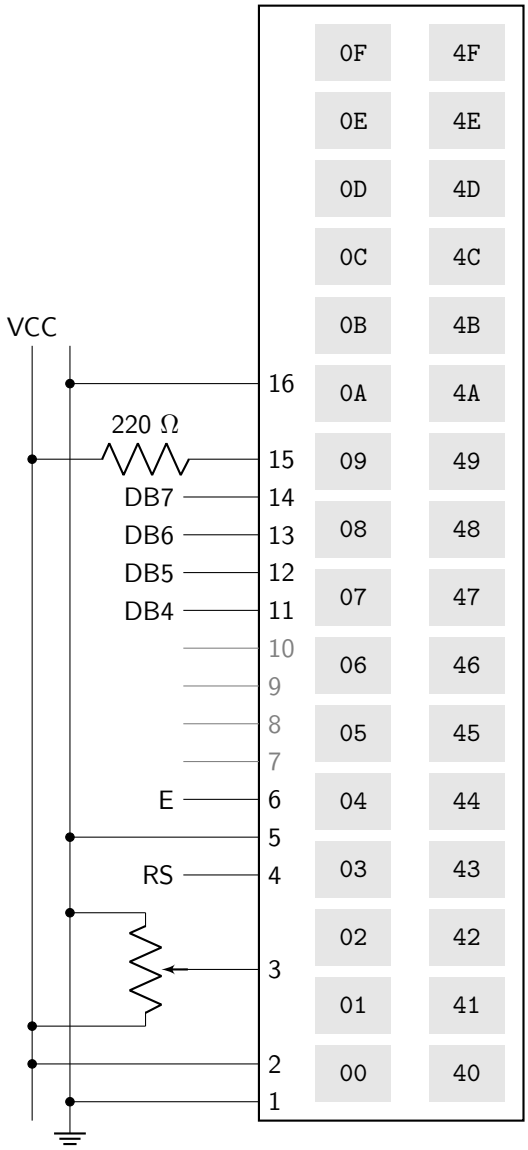
Multiplexed Quad 7-Segment Display PCB



RGB LED

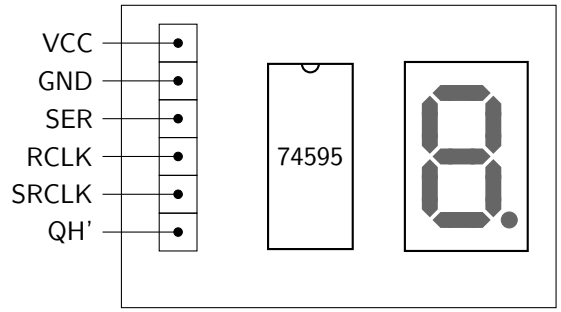


16x2 Character LCD Screen



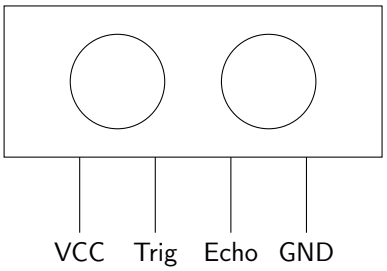
- RS – Register Select
- E – Enable
- DB4 – Bit 4
- DB5 – Bit 5
- DB6 – Bit 6
- DB7 – Bit 7

Serial Input 7-Segment Display PCB

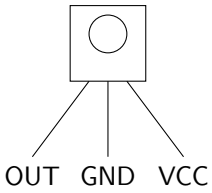


Sensors

HC-SR04: Ultrasonic Detector

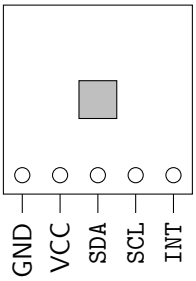


IR Receiver (38 kHz)



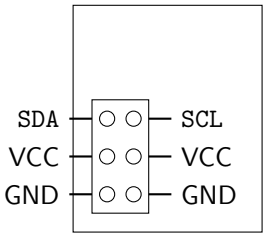
MAG-3110: 3-Axis Magnetometer

I^2C secondary address: `0x0E`

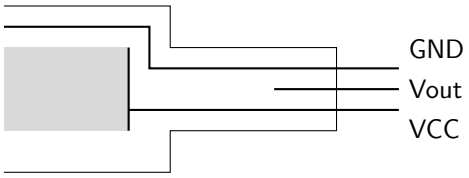


Redbot 3-Axis Accelerometer

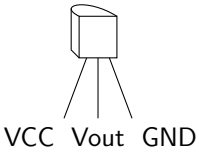
I^2C secondary address: `0x1D`



Soft Pot

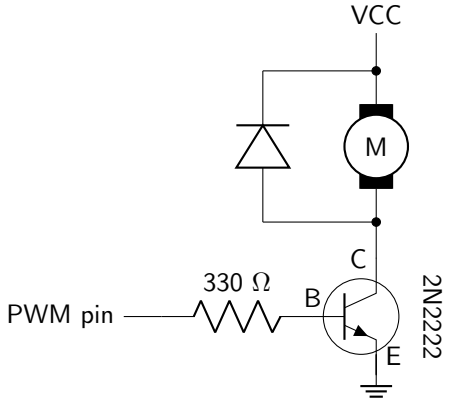


TMP36 Temperature Sensor



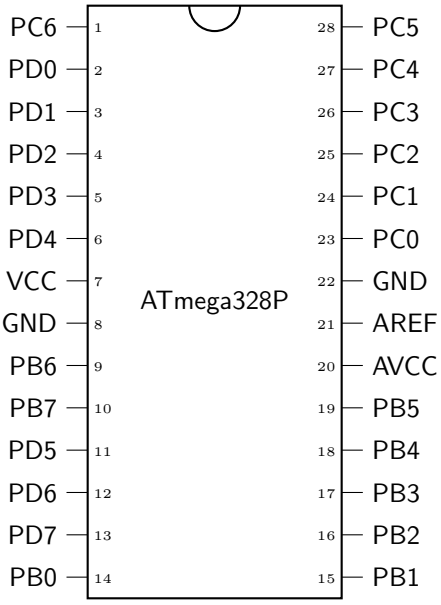
Motors

DC Motor (Single Direction)

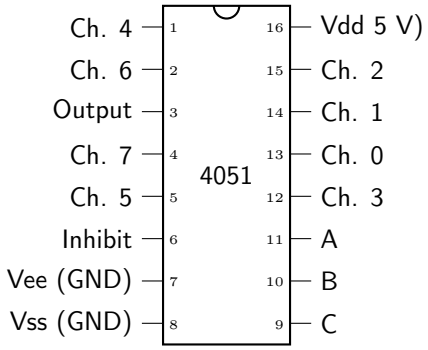


Analog and Digital ICs

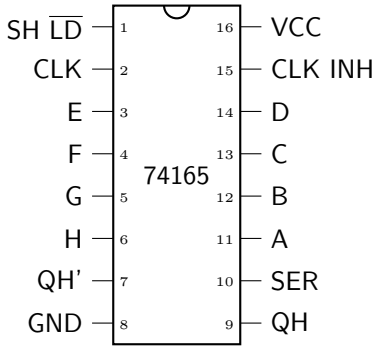
ATmega328P Microcontroller DIP-28



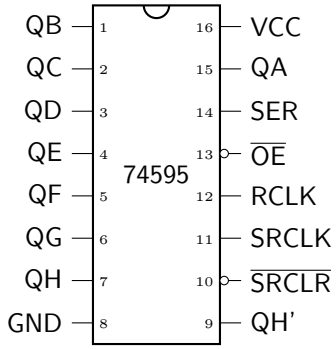
4051 8-bit Analog MUX



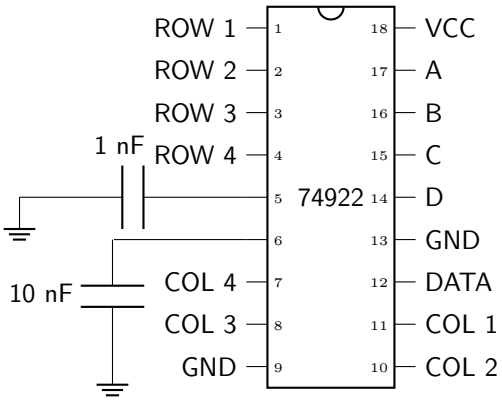
74165 8-bit PISO Shift Register



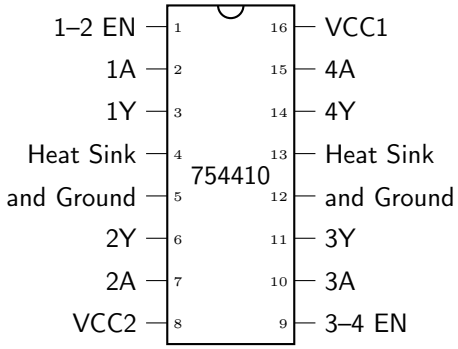
74595 8-bit SIPO Shift Register



74922: Keypad Encoder (Asynchronous Data Entry Mode)

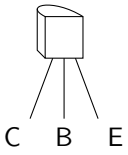


754410 Quadruple Half H Driver



Transistors

2N2222A NPN Transistor



Appendix C: Interrupt Vector Table

Interrupt Vector Table

Vector No.	Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset, watchdog system reset
2	0x0002	INT0	External interrupt request 0
3	0x0004	INT1	External interrupt request 1
4	0x0006	PCINT0	Pin change interrupt request 0
5	0x0008	PCINT1	Pin change interrupt request 1
6	0x000A	PCINT2	Pin change interrupt request 2
7	0x000C	WDT	Watchdog time-out interrupt
8	0x000E	TIMER2 COMPA	Timer/counter 2 compare match A
9	0x0010	TIMER2 COMPB	Timer/counter 2 compare match B
10	0x0012	TIMER2 OVF	Timer/counter 2 overflow
11	0x0014	TIMER1 CAPT	Timer/counter 1 capture event
12	0x0016	TIMER1 COMPA	Timer/counter 1 compare match A
13	0x0018	TIMER1 COMPB	Timer/counter 1 compare match B
14	0x001A	TIMER1 OVF	Timer/counter 1 overflow
15	0x001C	TIMER0 COMPA	Timer/counter 0 compare match A
16	0x001E	TIMER0 COMPB	Timer/counter 0 compare match B
17	0x0020	TIMER0 OVF	Timer/counter 0 overflow
18	0x0022	SPI, STC	SPI serial transfer complete
19	0x0024	USART, RX	USART Rx complete
20	0x0026	USART, UDRE	USART data register empty
21	0x0028	USART, TX	USART Tx complete
22	0x002A	ADC	ADC conversion complete
23	0x002C	EE READY	EEPROM ready
24	0x002E	ANALOG COMP	Analog comparator
25	0x0030	TWI	2-wire serial interface
26	0x0032	SPM READY	Store program memory ready

Appendix D: Alternate Port Functions

Alternate Functions of Port B

Port Pin	Alternate Functions
PB7	XTAL2 (Chip Clock Oscillator pin 2)
	TOSC2 (Timer Oscillator pin 2)
	PCINT7 (Pin Change Interrupt 7)
PB6	XTAL1 (Chip Clock Oscillator pin 1 or External clock input)
	TOSC1 (Timer Oscillator pin 1)
	PCINT6 (Pin Change Interrupt 6)
PB5	SCK (SPI Bus Primary clock Input)
	PCINT5 (Pin Change Interrupt 5)
PB4	MISO (SPI Bus Primary Input/Secondary Output)
	PCINT4 (Pin Change Interrupt 4)
PB3	MOSI (SPI Bus Primary Output/Secondary Input)
	OC2A (Timer/Counter2 Output Compare Match A Output)
	PCINT3 (Pin Change Interrupt 3)
PB2	$\overline{\text{SS}}$ (SPI Bus Secondary select)
	OC1B (Timer/Counter1 Output Compare Match B Output)
	PCINT2 (Pin Change Interrupt 2)
PB1	OC1A (Timer/Counter1 Output Compare Match A Output)
	PCINT1 (Pin Change Interrupt 1)
PB0	ICP1 (Timer/Counter1 Input Capture Input)
	CLKO (Divided System Clock Output)
	PCINT0 (Pin Change Interrupt 0)

Alternate Functions of Port C

Port Pin	Alternate Functions
PC6	$\overline{\text{RESET}}$ (Reset pin)
	PCINT14 (Pin Change Interrupt 14)
PC5	ADC5 (ADC Input Channel 5)

	SCL (2-wire Serial Bus Clock Line)
	PCINT13 (Pin Change Interrupt 13)
PC4	ADC4 (ADC Input Channel 4)
	SDA (2-wire Serial Bus Data Input/Output Line)
	PCINT12 (Pin Change Interrupt 12)
PC3	ADC3 (ADC Input Channel 3)
	PCINT11 (Pin Change Interrupt 11)
PC2	ADC2 (ADC Input Channel 2)
	PCINT10 (Pin Change Interrupt 10)
PC1	ADC1 (ADC Input Channel 1)
	PCINT9 (Pin Change Interrupt 9)
PC0	ADC0 (ADC Input Channel 0)
	PCINT8 (Pin Change Interrupt 8)

Alternate Functions of Port D

Port Pin	**Alternate Functions**
PD7	AIN1 (Analog Comparator Negative Input)
	PCINT23 (Pin Change Interrupt 23)
PD6	AIN0 (Analog Comparator Positive Input)
	OC0A (Timer/Counter0 Output Compare Match A Output)
	PCINT22 (Pin Change Interrupt 22)
PD5	T1 (Timer/Counter1 External Counter Input)
	OC0B (Timer/Counter0 Output Compare Match B Output)
	PCINT21 (Pin Change Interrupt 21)
PD4	XCK (USART External Clock Input/Output)
	T0 (Timer/Counter0 External Counter Input)
	PCINT20 (Pin Change Interrupt 20)
PD3	INT1 (External Interrupt 1 Input)
	OC2B (Timer/Counter2 Output Compare Match B Output)
	PCINT19 (Pin Change Interrupt 19)
PD2	INT0 (External Interrupt 0 Input)

	PCINT18 (Pin Change Interrupt 18)
PD1	TXD (USART Output Pin)
	PCINT17 (Pin Change Interrupt 17)
PD0	RXD (USART Input Pin)
	PCINT16 (Pin Change Interrupt 16)

Appendix E: C Datatypes

Integer Types

Data Type	Length	Range of Values
`char`	8 bits	-128 to 127
`unsigned char`	8 bits	0 to 255
`int`	16 bits	$-32,768$ to 32,767
`unsigned int`	16 bits	0 to 65,535
`long`	32 bits	$-2,147,483,648$ to 2,147,483,647
`unsigned long`	32 bits	0 to 4,294,967,295

Examples

```
// signed char
char a = -67; // decimal
char a = 0b10111101; // binary
char a = 0xBD; // hexadecimal

// unsigned char
unsigned char x = 248; // decimal
unsigned char x = 0b11111000; // binary
unsigned char x = 0xF8; // hexadecimal

// unsigned long
unsigned long a = 1000910518; // decimal
unsigned long a = 0b00111011101010001010111010110110; // binary
unsigned long a = 0x3BA8AEB6; // hexadecimal

// signed long
long b = -3102; // decimal
long b = 0b11111111111111111111001111100010; // binary
long b = 0xFFFFF3E2; // hexadecimal

// alphanumeric characters
char a = 'X'; // alphanumeric
char a = 88; // decimal
char a = 0b01011000; // binary
char a = 0x58; // hexadecimal

// character arrays ("strings")
char a[] = "Hello!"; // alphanumeric
char a[7] = "Hello!"; // alphanumeric
char a[] = {'H','e','l','l','o','!','\0'}; // alphanumeric
char a[] = {72, 101, 108, 108, 111, 33}; // decimal
char a[] = {0b01001000, 0b01100101, 0b01101100, 0b01101100, 0b01101111,0
    b00100001}; // binary
char a[] = {0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x21} // hexadecimal
```

Floating-Point Types

Data Type	Length	Range of Values
`float`	32 bits	-3.4×10^{38} to 3.4×10^{38}
`double`	32 bits	-3.4×10^{38} to 3.4×10^{38}

The two types used on the Arduino Uno with the Arduino IDE, `float` and `double`, are both single-precision type floating-point numbers. These numbers have a single sign bit, 8 exponent bits, and 23 fractional bits. The number they represent is described by equation 17.1, where s is the sign bit, f is the fractional number, and e is the exponent.

$$(-1^s) \times (1.f) \times \left(2^{e-127}\right) \tag{17.1}$$

Appendix F: C Operators

Assignment Operator

Operator	Description
=	Equals

Bitshift Operators

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
>>	Bitshift Right
<<	Bitshift Left

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Comparison Operators

Operator	Description
==	Equal To
!=	Not Equal To
>	Greater Than
>=	Greater Than or Equal To

<	Less Than
<=	Less Than or Equal To

Boolean Operators

Operator	Description
&&	AND
||	OR
!	NOT

Compound Operators

Operator	Description
++	Increment
-	Decrement
+=	Compound Addition
-=	Compound Subtraction
*=	Compound Multiplication
/=	Compound Division
%=	Compound Modulo
&=	Compound Bitwise AND
|=	Compound Bitwise OR
^=	Compound Bitwise XOR
>>=	Compound Bitshift Right
<<=	Compound Bitshift Left

Appendix G: Register Summary

The following list denotes all of the registers stored in the SRAM of the ATmega328P with their corresponding address. Missing addresses denote reserved (unused) register spaces.

General Purpose Registers

0x00	r0	0x08	r8	0x10	r16	0x18	r24
0x01	r1	0x09	r9	0x11	r17	0x19	r25
0x02	r2	0x0A	r10	0x12	r18	0x1A	r26 (X low)
0x03	r3	0x0B	r11	0x13	r19	0x1B	r27 (X high)
0x04	r4	0x0C	r12	0x14	r20	0x1C	r28 (Y low)
0x05	r5	0x0D	r13	0x15	r21	0x1D	r29 (Y high)
0x06	r6	0x0E	r14	0x16	r22	0x1E	r30 (Z low)
0x07	r7	0x0F	r15	0x17	r23	0x1F	r31 (Z high)

I/O Registers

Note that these registers can and should be addressed using I/O direct addressing instructions such as `IN` and `OUT`.

0x23	PINB	0x36	TIFR1	0x43	GTCCR	0x4E	SPDR
0x24	DDRB	0x37	TIFR2	0x44	TCCR0A	0x50	ACSR
0x25	PORTB	0x3B	PCIFR	0x45	TCCR0B	0x53	SMCR
0x26	PINC	0x3C	EIFR	0x46	TCNT0	0x54	MCUSR
0x27	DDRC	0x3D	EIMSK	0x47	OCR0A	0x55	MCUCR
0x28	PORTC	0x3E	GPIOR0	0x48	OCR0B	0x57	SPMCSR
0x29	PIND	0x3F	EECR	0x4A	GPIOR1	0x5D	SPL
0x2A	DDRD	0x40	EEDR	0x4B	GPIOR2	0x5E	SPH
0x2B	PORTD	0x41	EEARL	0x4C	SPCR	0x5F	SREG
0x35	TIFR0	0x42	EEARH	0x4D	SPSR		

Extended I/O Registers

Note that these registers must be addressed using data direct and indirect addressing instructions such as `STS` and `LDS` (direct) and `ST` and `LD` (indirect).

0x60	WDTCSR	0x78	ADCL	0x86	ICR1L	0xB8	TWBR
0x61	CLKPR	0x79	ADCH	0x87	ICR1H	0xB9	TWSR
0x64	PRR	0x7A	ADCSRA	0x88	OCR1AL	0xBA	TWAR
0x66	OSCCAL	0x7B	ADCSRB	0x89	OCR1AH	0xBB	TWDR
0x68	PCICR	0x7C	ADMUX	0x8A	OCR1BL	0xBC	TWCR
0x69	EICRA	0x7E	DIDR0	0x8B	OCR1BH	0xBD	TWAMR
0x6B	PCMSK0	0x7F	DIDR1	0xB0	TCCR2A	0xC0	UCSR0A
0x6C	PCMSK1	0x80	TCCR1A	0xB1	TCCR2B	0xC1	UCSR0B
0x6D	PCMSK2	0x81	TCCR1B	0xB2	TCNT2	0xC2	UCRS0C
0x6E	TIMSK0	0x82	TCCR1C	0xB3	OCR2A	0xC4	UBRR0L
0x6F	TIMSK1	0x84	TCNT1L	0xB4	OCR2B	0xC5	UBRR0H
0x70	TIMSK2	0x85	TCNT1H	0xB6	ASSR	0xC6	UDR0

Appendix H: Instruction Set Summary

The following is a summary of the AVR instruction set. More details are available in the AVR Instruction Set Manual.

Operand Descriptions

Operand	Description
Rd	Destination (GP register)
Rr	Source (GP register)
K	Constant data (immediate)
k	Constant address
b	Bit number in register
s	Bit number in SREG
X,Y,Z	Pointer register
A	I/O register
q	Displacement

Arithmetic and Logic Instructions

Mnemonic	Operand(s)	Description	Operation
ADD	Rd, Rr	Add without carry	Rd ← Rd + Rr
ADC	Rd, Rr	Add with carry	Rd ← Rd + Rr + C
ADIW	Rd, K	Add immediate to word	Rd ← Rd + 1:Rd + K
SUB	Rd, Rr	Subtract without carry	Rd ← Rd - Rr
SUBI	Rd, K	Subtract immediate	Rd ← Rd - K
SBC	Rd, Rr	Subtract with carry	Rd ← Rd - Rr - C
SBCI	Rd, K	Subtract immediate with carry	Rd ← Rd - K - C
SBIW	Rd, K	Subtract immediate from word	Rd + 1:Rd ← Rd + 1:Rd - K
AND	Rd, Rr	Logical AND	Rd ← Rd · Rr
ANDI	Rd, K	Logical AND with immediate	Rd ← Rd · K
OR	Rd, Rr	Logical OR	Rd ← Rd ∨ Rr
ORI	Rd, K	Logical OR with immediate	Rd ← Rd ∨ K
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr

COM	Rd	One's complement	Rd ← 0xFF - Rd
NEG	Rd	Two's complement	Rd ← 0x00 - Rd
SBR	Rd, K	Set bit(s) in register	Rd ← Rd ∨ K
CBR	Rd, K	Clear bit(s) in register	Rd ← Rd · (0xFF - K)
INC	Rd	Increment	Rd ← Rd + 1
DEC	Rd	Decrement	Rd ← Rd - 1
TST	Rd	Test for zero or minus	Rd ← Rd · Rd
CLR	Rd	Clear register	Rd ← Rd ⊕ Rd
SER	Rd	Set register	Rd ← 0xFF
MUL	Rd, Rr	Multiply unsigned	R1:R0 ← Rd × Rr
MULS	Rd, Rr	Multiply signed	R1:R0 ← Rd × Rr
MULSU	Rd, Rr	Multiply signed with unsigned	R1:R0 ← Rd × Rr
FMUL	Rd, Rr	Fractional multiply unsigned	R1:R0 ← (Rd × Rr) « 1
FMULS	Rd, Rr	Fractional multiply signed	R1:R0 ← (Rd × Rr) « 1
FMULSU	Rd, Rr	Fractional multiply signed with unsigned	R1:R0 ← (Rd × Rr) « 1

Branch Instructions

Mnemonic	Operand(s)	Description	Operation
RJMP	k	Relative jump	PC ← PC + k + 1
IJMP		Indirect jump to (Z)	PC ← Z
JMP	k	Jump	PC ← k
RCALL	k	Relative subroutine call	PC ← PC + k + 1
ICALL		Indirect call to (Z)	PC ← Z
CALL	k	Direct subroutine call	PC ← k
RET		Subroutine return	PC ← STACK
RETI		Interrupt return	PC ← STACK
CPSE	Rd, Rr	Compare, skip if equal	if (Rd = Rr) PC ← PC + 2 or 3
CP	Rd, Rr	Compare	Rd - Rr
CPC	Rd, Rr	Compare with carry	Rd - Rr - C

CPI	Rd, K	Compare with immediate	Rd - K
SBRC	Rr, b	Skip if bit in register cleared	if (Rr(b) = 0) PC ← PC + 2 or 3
SBRS	Rr, b	Skip if bit in register set	if (Rr(b) = 1) PC ← PC + 2 or 3
SBIC	A, b	Skip if bit in I/O register cleared	if (I/O(A, b) = 0) PC ← PC + 2 or 3
SBIS	A, b	Skip if bit in I/O register set	if (I/O(A, b) = 1) PC ← PC + 2 or 3
BRBS	s, k	Branch if status flag set	if (SREG(s) = 1) PC ← PC + k + 1
BRBC	s, k	Branch if status flag cleared	if (SREG(s) = 0) PC ← PC + k + 1
BREQ	k	Branch if equal	if (Z = 1) PC ← PC + k + 1
BRNE	k	Branch if not equal	if (Z = 0) PC ← PC + k + 1
BRCS	k	Branch if carry set	if (C = 1) PC ← PC + k + 1
BRCC	k	Branch if carry cleared	if (C = 0) PC ← PC + k + 1
BRSH	k	Branch if same or higher	if (C = 0) PC ← PC + k + 1
BRLO	k	Branch if lower	if (C = 1) PC ← PC + k + 1
BRMI	k	Branch if minus	if (N = 1) PC ← PC + k + 1
BRPL	k	Branch if plus	if (N = 0) PC ← PC + k + 1
BRGE	k	Branch if greater or equal, signed	if (N ⊕ V = 0) PC ← PC + k + 1
BRLT	k	Branch if less than, signed	if (N ⊕ V = 1) PC ← PC + k + 1
BRHS	k	Branch if half carry flag set	if (H = 1) PC ← PC + k + 1
BRHC	k	Branch if half carry flag cleared	if (H = 0) PC ← PC + k + 1
BRTS	k	Branch if T flag set	if (T = 1) PC ← PC + k + 1
BRTC	k	Branch if T flag cleared	if (T = 0) PC ← PC + k + 1
BRVS	k	Branch if overflow flag is set	if (V = 1) PC ← PC + k + 1
BRVC	k	Branch if overflow flag is cleared	if (V = 0) PC ← PC + k + 1
BRIE	k	Branch if interrupt enabled	if (I = 1) PC ← PC + k + 1
BRID	k	Branch if interrupt disabled	if (I = 0) PC ← PC + k + 1

Bit and Bit-Test Instructions

Mnemonic	Operand(s)	Description	Operation
SBI	A, b	Set bit in I/O register	I/O(A, b) ← 1
CBI	A, b	Clear bit in I/O register	I/O(A, b) ← 0
LSL	Rd	Logical shift left	Rd(n+1) ← Rd(n), Rd(0) ← 0
LSR	Rd	Logical shift right	Rd(n) ← Rd(n+1), Rd(7) ← 0
ROL	Rd	Rotate left through carry	Rd(0) ← C, Rd(n+1) ← Rd(n), C ← Rd(7)
ROR	Rd	Rotate right through carry	Rd(7) ← C, Rd(n) ← Rd(n+1), C ← Rd(0)
ASR	Rd	Arithmetic shift right	Rd(n) ← Rd(n+1), n = 0..6
SWAP	Rd	Swap nibbles	Rd(3..0) ← Rd(7..4), Rd(7..4) ← Rd(3..0)
BSET	s	Flag set	SREG(s) ← 1
BCLR	s	Flag clear	SREG(s) ← 0
BST	Rr, b	Bit store from register to T	T ← Rr(b)
BLD	Rd, b	Bit load from T to register	Rd(b) ← T
SEC		Set carry	C ← 1
CLC		Clear carry	C ← 0
SEN		Set negative flag	N ← 1
CLN		Clear negative flag	N ← 0
SEZ		Set zero flag	Z ← 1
CLZ		Clear zero flag	Z ← 0
SEI		Global interrupt enable	I ← 1
CLI		Global interrupt disable	I ← 0
SES		Set signed test flag	S ← 1
CLS		Clear signed test flag	S ← 0
SEV		Set two's complement overflow	V ← 1
CLV		Clear two's complement overflow	V ← 0
SET		Set T in SREG	T ← 1

CLT	Clear T in SREG	T ← 0
SEH	Set half carry flag in SREG	H ← 1
CLH	Clear half carry flag in SREG	H ← 0

Data Transfer Instructions

Mnemonic	Operand(s)	Description	Operation
MOV	Rd, Rr	Move between registers	Rd ← Rr
MOVW	Rd, Rr	Copy register pair	Rd + 1:Rd ← Rr + 1:Rr
LDI	Rd, K	Load immediate	Rd ← K
LD	Rd, X	Load indirect	Rd ← (X)
LD	Rd, X+	Load indirect and post-increment	Rd ← (X), X ← X + 1
LD	Rd, -X	Load indirect and pre-decrement	X ← X - 1, Rd ← (X)
LD	Rd, Y	Load indirect	Rd ← (Y)
LD	Rd, Y+	Load indirect and post-increment	Rd ← (Y), Y ← Y + 1
LD	Rd, -Y	Load indirect and pre-decrement	Y ← Y - 1, Rd ← (Y)
LDD	Rd, Y + q	Load indirect with displacement	Rd ← (Y + q)
LD	Rd, Z	Load indirect	Rd ← (Z)
LD	Rd, Z+	Load indirect and post-increment	Rd ← (Z), Z ← Z + 1
LD	Rd, -Z	Load indirect and pre-decrement	Z ← Z - 1, Rd ← (Z)
LDD	Rd, Z + q	Load indirect with displacement	Rd ← (Z + q)
LDS	Rd, k	Load direct from SRAM	Rd ← (k)
ST	X, Rr	Store indirect	(X) ← Rr
ST	X+, Rr	Store indirect and post-increment	(X) ← Rr, X ← X + 1
ST	-X, Rr	Store indirect and pre-decrement	X ← X - 1, (X) ← Rr
ST	Y, Rr	Store indirect	(Y) ← Rr
ST	Y+, Rr	Store indirect and post-increment	(Y) ← Rr, Y ← Y + 1
ST	-Y, Rr	Store indirect and pre-decrement	Y ← Y - 1, (Y) ← Rr
STD	Y + q, Rr	Store indirect with displacement	(Y + q) ← Rr
ST	Z, Rr	Store indirect	(Z) ← Rr
ST	Z+, Rr	Store indirect and post-increment	(Z) ← Rr, Z ← Z + 1

ST	-Z, Rr	Store indirect and pre-decrement	Z ← Z - 1, (Z) ← Rr
STD	Z + q, Rr	Store indirect with displacement	(Z + q) ← Rr
STS	k, Rr	Store direct to SRAM	(k) ← Rr
LPM		Load program memory	R0 ← (Z)
LPM	Rd, Z	Load program memory	Rd ← (Z)
LPM	Rd, Z+	Load program memory and post-increment	Rd ← (Z), Z ← Z + 1
SPM		Store program memory	(Z) ← R1:R0
IN	Rd, A	In from I/O location	Rd ← I/O(A)
OUT	A, Rr	Out to I/O location	I/O(A) ← Rr
PUSH	Rr	Push register on stack	STACK ← Rr
POP	Rd	Pop register from stack	Rd ← STACK

MCU Control Instructions

Mnemonic	Operand(s)	Description	Operation
NOP		No operation	
SLEEP		Sleep	
WDR		Watchdog reset	
BREAK		Break (for on-chip debug only)	