Lab 8

Using Polling with Memory-Mapped Input/Output

Due: Week of April 11, Before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

In this assignment, you will write functions to use the input and output devices on your Arduino Nano-based class hardware kit. You will then use those functions to implement the ability to build a number, similar to entering a value on a calculator, using polling.

Figure 1: Polling. Image by 20th Century Fox Television

The instructions are written assuming you will edit the code in the Arduino IDE and run it on your Arduino Nano-based class hardware kit, constructed according to the pre-lab instructions. If you wish, you may edit the code in a different environment; however, our ability to provide support for problems with other IDEs is limited.

Please familiarize yourself with the entire assignment before beginning. Section 2 has the functional specification of the system you will develop. Section 3 describes implementation constraints. Section 4 guides you in implementing the first portion of the system, and Section 5 offers suggestions in implementing the second portion of the system.

Learning Objectives

After successful completion of this assignment, students will be able to:

- Use memory-mapped I/O to obtain inputs from peripheral devices
- Use memory-mapped I/O to send outputs to peripheral devices
- Poll an input to determine when it has taken on a value of interest
- Scan a matrix keypad
- Use the Serial-Parallel Interface (SPI) protocol
- Display a value or a message on a collection of 7-segment displays

Continuing Forward

We will use the hardware kit for the remaining labs. In the labs after this one, you will not be required to use the memory-mapped I/O registers (but you may!); however, we will assume that you know how to scan a matrix keypad, and that you can build and display a value from keypad inputs.

During Lab Time

During your lab period, the TAs will provide a refresher on bitmasks, both to read inputs and to use the read-modify-write pattern for outputs. They will also demonstrate creating the bit patterns that you will use for the 7-segment display module by guiding students through a discussion and discovery of the bit pattern for the numeral 2. During the remaining time, the TAs will be available to answer questions.

No Spaghetti Code Allowed

In the interest of keeping your code readable, you may *not* use any **`goto`** statements, nor may you use any **`continue`** statements, nor may you use any **`break`** statements to exit from a loop, nor may you have any functions **`return`** from within a loop.

1 Scenario

Archie walks up to you, along with Herb Bee from Eclectic Electronics. Herb is holding a tangled mess of electronics. Archie explains, "Herb here has developed a prototype of a device that he thinks will be useful for our physical security needs, as well as a few other applications around here. He calls it the *Cow Pi*."

You look at the device in Herb's hands and see the Arduino Nano central to the circuit. "Isn't *-Pi* typically used as a suffix for circuits that use a Raspberry Pi instead of an Arduino?"

Herb replies, "Typically, yes, but *Cowduino* isn't very punny, is it?"

Archie chimes in, "Maybe with the right emphasis: *Cow-DOO-ino*."

"That's kind of subtle, don't you think? How will people know to put the emPHAsis on that sylLAble?"

"I think we're getting off topic here," you point out. "How can I help?"

"Oh, right," Herb says, "We'd like you to kick its proverbial tires. Let's start off with something simple, like a number builder tool."

2 Number Builder Specification

1. The tool shall have two modes, *demonstration mode* and *builder mode*. The **left switch** controls the mode.

2. The tool shall have two number bases, *decimal* and *hexadecimal*. The **right switch** controls the number base.

3. When the **right switch** is toggled to the left, the number builder will use the decimal number base. When the user presses a button on the **matrix keypad** with a decimal numeral (*0-9*) then the number builder shall take the appropriate action as specified in requirement 6c or 7a. Any other buttons on the keypad shall be ignored.

4. When the **right switch** is toggled to the right, the number builder will use the hexadecimal number base. When the user presses a button on the **matrix keypad** then the number builder shall take the appropriate action as specified in requirement 6c or 7a. Buttons with decimal numerals (*0-9*) or alphabetic letters (*A-D*) shall be interpreted as having the corresponding hexadecimal numeral; the button with the octothorp (#) shall be interpreted as having the hexadecimal numeral *E*; and the button with the asterisk (*) shall be interpreted as having the hexadecimal numeral *F*.

5. Initially, no digits on the **7-segment display module** shall have any of their segments illuminated.

6. When the **left switch** is toggled to the left, the number builder is in demonstration mode. When the number builder is in demonstration mode:

   (a) Whenever the user presses a button on the **matrix keypad**, the **external LED** shall illuminate for approximately one-half of a second.

   (b) The **external LED** shall not illuminate except as specified in requirement 6a.

   (c) Whenever the user presses a button on the **matrix keypad**, the corresponding numeral shall be displayed in the least-significant digit position of the **7-segment**

display module, replacing any numeral previously displayed. The numeral displayed shall follow the interpretations specified in requirements 3 and 4.

(d) If the user toggles the **right switch** (*i.e.*, changes the number base) while in demonstration mode, then the numeral being displayed shall continue to be displayed even if it is an invalid numeral in the new number base. The system shall, however, respect the new number base for future keypresses.

(e) While the system is in demonstration mode, here are no requirements nor restrictions with respect to printing to the Serial Monitor.

(f) Whenever the user presses the **right pushbutton** while the system is in demonstration mode, the **7-segment display module** shall be cleared: no segments shall be illuminated.

7. When the **left switch** is toggled to the right, the number builder is in builder mode. When the number builder is in builder mode:

(a) Whenever the user presses a button on the **matrix keypad**, the corresponding numeral shall be displayed in the least-significant position of the **7-segment display module**, and any digits already displayed shall increase in significance by one order of magnitude. For example, if 234 is displayed and the user presses 5 then 2345 shall be displayed. The numeral displayed shall follow the interpretations specified in requirements 3 and 4.

i. There shall be no noticeable lag in updating the display.

ii. The new value shall be printed to the Serial Monitor in decimal or hexadecimal, depending on the system's current number base.

(b) Whenever the user presses the **left pushbutton**, the value being displayed shall be negated.

i. In the decimal number base, the presence or absence of negative sign shall indicate whether or not a value is negative

ii. In the hexadecimal number base, 32-bit two's complement shall be used.

(c) "0x" shall not be displayed as part of a hexadecimal value.

(d) A positive sign shall not be displayed as part of a positive value.

(e) If a negative decimal value is displayed, the negative sign shall be displayed immediately to the left of the most-significant digit being displayed. For example, -456 is correctly displayed, but - 456 is not correctly displayed.

(f) Leading 0s may be displayed but are discouraged. For example, 782 is preferred, but 00000782 is allowed.

(g) In no case shall the tool allow the user to input a value too great to be displayed on the **7-segment display module**. If the user attempts to enter a value greater than 0x7FFF,FFFF in hexadecimal, less than 0x8000,0000 in hexadecimal, greater than 99,999,999 in decimal, or less than -9,999,999 in decimal, then the **7-segment display module** shall display too big.

(h) Except as specified in requirement 7(a)ii, nothing shall be printed on the Serial Montior when the system is in building mode.

(i) If the user toggles the **right switch** (*i.e.*, changes the number base) while in builder mode: if the number being built is 0 then the system shall seamlessly transition between number bases; if the number being built is not 0 then the system's behavior is unspecified.

(j) Whenever the user presses the **right pushbutton** while the system is in builder mode, the number being built shall reset to 0, and the **7-segment display module** shall be cleared, *except* that the least-significant digit shall display a 0 (this being the new number being built).

8. If the user changes modes (*i.e.*, toggles the **left switch**) then the following behavior is recommended but not required:

   (a) When changing from demonstration mode to builder mode, the numeral being displayed (if any) may become the most-significant digit of the number to be built.

   (b) When changing from builder mode to demonstration mode, the numerals to the left of the least-significant digit (if any) may continue to be displayed without significance.

9. When the user presses a button for less than approximately one-half of a second, regardless of whether it is one of the **pushbuttons** or a button on the **matrix keypad**, it shall be treated as a single button press.

3 Constraints

All input and output must be accomplished using the memory-mapped I/O registers. You may use any constants, arrays, structs, typedefs, or macros present in the *cowpi.h* header file provided as part of this assignment. You may use any code that you write yourself. You may use any features that are part of the C standard if they are supported by the compiler. Except as noted below, you may *not* use any libraries, functions, macros, types, or constants that are part of the Arduino core[1] (unless they are also part of the C standard), nor AVR-specific functions, macros, types, or constants of avr-libc[2].

You *may* use the **`millis()`**[3] function to measure the passage of time, and you *may* use the **`Serial.begin()`**[4], **`Serial.print()`**[5], and **`Serial.println()`**[6] functions for debugging

[1] https://www.arduino.cc/reference/en/
[2] https://www.nongnu.org/avr-libc/user-manual/index.html
[3] https://www.arduino.cc/reference/en/language/functions/time/millis/
[4] https://www.arduino.cc/reference/en/language/functions/communication/serial/begin/
[5] https://www.arduino.cc/reference/en/language/functions/communication/serial/print/
[6] https://www.arduino.cc/reference/en/language/functions/communication/serial/println/

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

Figure 2: ATmega328P I/O port registers. Each register's I/O address and memory address (in parentheses) are provided. Cropped from ATmega382P Data Sheet, §30

and for implementing requirement 7(a)ii. *Be sure to remove* ***`print()`*** *and* ***`println()`*** *calls that are used only for debugging before turning in your code for grading.*

(A quick note on using **`print()`** and **`println()`** for debugging: if you print too much, the UART buffer may fill, making it difficult to upload a new program to the Arduino Nano. If you have an upload failure in this situation, the simplest fix in this case is to unplug the Arduino Nano, plug it back in, and re-attempt the upload.) (Another quick note on using **`print()`** and **`println()`**: please read these functions' documentation to understand what the functions can – and can't – do.)

4 Implementing Demonstration Mode

You will use "demonstration mode" to configure your Arduino Nano to use the memory-mapped input/output controls and peripheral devices.

4.1 Simple Input/Output

The ATmega328P microcontroller[7] on the Arduino Nano has three input/output ports accessible by external pins. Each port has three registers, the PIN input register, the PORT output register, and the DDR data direction register used to set each pin as input or output. Each pin is individually controlled by a particular bit in the port registers. Figure 2 shows these nine registers and their corresponding addresses. You do not need to configure the pins for input or output; the **`cowpi_setup`** macro provided in *cowpi.h* takes care of all necessary configuration.

Figure 3 shows which bit in which port corresponds to each Arduino Nano pin. For example, pin `D10` is labeled "PB2" indicating that it is part of port B and uses bit 2 in each of port B's registers. If `D10` were an output pin, then we could set the pin's logic level to high or low by assigning a 1 or 0, respectively, to `PORTB`'s bit 2. On the other hand, if `D10` were

[7] http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

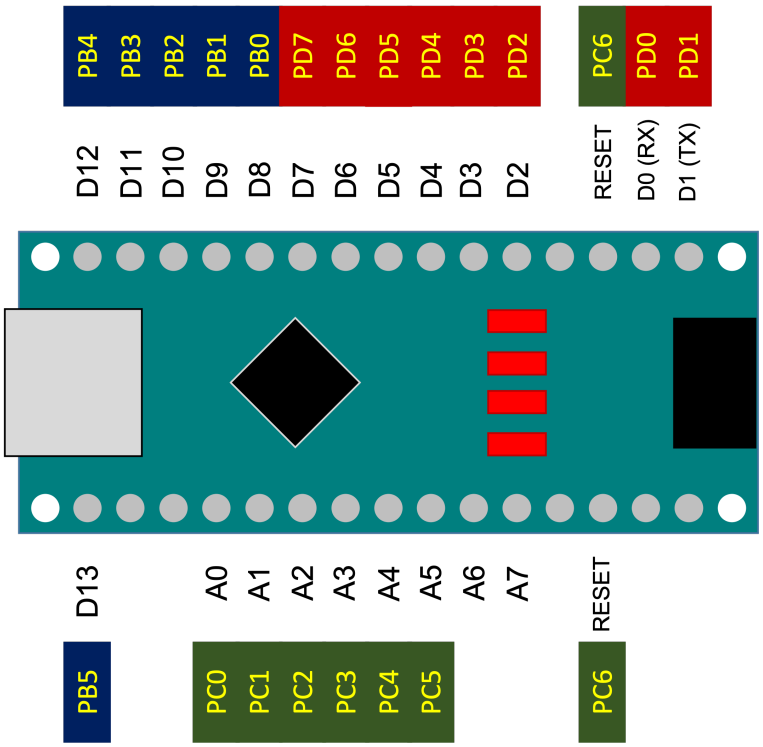


Figure 3: Mapping of Arduino Nano pins to ATmega328P input/output ports. Diagram by Bohn

an input pin, then we could determine the pin's logic level by using a bitmask to examine `PORTB`'s bit 2.

4.1.1 Accessing I/O Registers in Memory Address Space

The header file *cowpi.h* provides data structures to access the memory-mapped I/O registers in a more readable form. For example, the `cowpi_ioPortRegisters` structure eliminates the need to remember which I/O port registers are used for output to peripherals and which are used for input from peripherals.

```
typedef struct {
  volatile uint8_t input;                    // PINx
  volatile uint8_t direction;                // DDRx
  volatile uint8_t output;                   // PORTx
} cowpi_ioPortRegisters;
```

The Arduino Nano's three I/O ports are placed contiguously in the memory address space, which will allow us to create a pointer to the lowest-addressed port and then treat that pointer as an array of I/O ports. Some named constants that we can use to index that array further eliminate the need to remember which port corresponds to each Arduino Nano pin.

```
#define D8_D13   0                    // PINB/DDRB/PORTB / PCMSK0
```

```
#define A0_A5    1                    // PINC/DDRC/PORTC / PCMSK1
#define D0_D7    2                    // PIND/DDRD/PORTD / PCMSK2
```

Finally, there is a constant pointer, `cowpi_IObase` that holds the memory address that is the start of ATmega328P's memory-mapped input/output register bank, and a comment reminding us that the external pins' input/output registers start 3 bytes above `cowpi_IObase` (which is also information we can get from Figure 2).

```
uint8_t * const cowpi_IObase = (uint8_t *)0x20;

/* ================================
 * ARDUINO NANO and UNO (ATMega328P)
 * ================================
 *
 *
 * EXTERNAL PINS
 * Pins `D8_D13` `A0_A5` `D0_D7` cowpi_ioPortRegisters[] at
   ↪ cowpi_IObase + 0x03 (0x23)
```

On line 17 of *PollingLab.ino*, there is a global variable, `ioPorts` that will serve as our array of I/O ports.

- On line 50 of *PollingLab.ino*, find this line in **`setup()`**: `// ioPorts = ...`
- Remove the comment mark and ellipses, and assign to `ioPorts` the address 3 bytes above `IObase`; you will need to cast the resulting address to `(cowpi_ioPortRegisters *)`.

Index the `cowpi_ioPortRegisters` array using the `D8_D13`, `A0_A5`, and `D0_D7` named constants. Each element of the array has an `.input` field to read logic levels from peripherals and a `.output` field to assign logic levels for peripherals (as well as a `.direction` field that you won't use).

As you'll notice in Figure 3, the lower-numbered pins are their registers' less-significant bits (and thus are the less-significant bits of the `cowpi_ioPortRegisters`' fields), and the higher-numbered pins are the more-significant bits. Continuing with our earlier example, if pin `D10` were an input pin, then it would correspond with bit 2 of `ioPorts[D8_D13].input`. If pin `D10` were an output pin, then it would correspond with bit 2 of `ioPorts[D8_D13].output` (having the third-lowest number in the range 13..8, `D10` corresponds with the third-least-significant bit).

To read the logic level of a specific pin or pins, apply a bitmask to the appropriate array element's `.input` field so that the resulting bitfield preserves the bit(s) you're interested in and has 0s in the remaining bits. To set the logic level of a specific pin or pins, apply the read-modify-write pattern to the appropriate array element's `.output` field to set the value of the bit(s) you're interested in and preserve the values of the remaining bit(s).

4.1.2 Reading Polling Code

The starter code includes a **`testSimpleIO()`** function. If you upload *PollingLab.ino* to your Arduino Nano, you will see on the Serial Monitor a message whenever you press a button or toggle a switch. The external LED will also light up whenever both switches are in the right position (if the room is bright, you may have to shade the LED with your hand to see that it has illuminated).

Recall that the left switch is connected to pin `A4` and the right switch is connected to pin `A5`. Recall also that the left pushbutton is connected to pin `D8` and the right pushbutton is connected to pin `D9`. Notice that every time the **`loop()`** function iterates, the pushbuttons' (pins `D8` & `D9`) and switches' (pins `A4` & `A5`) logic values are checked, to determine if they have taken on a value that we're interested in; that is, the code *polls* the input pins.

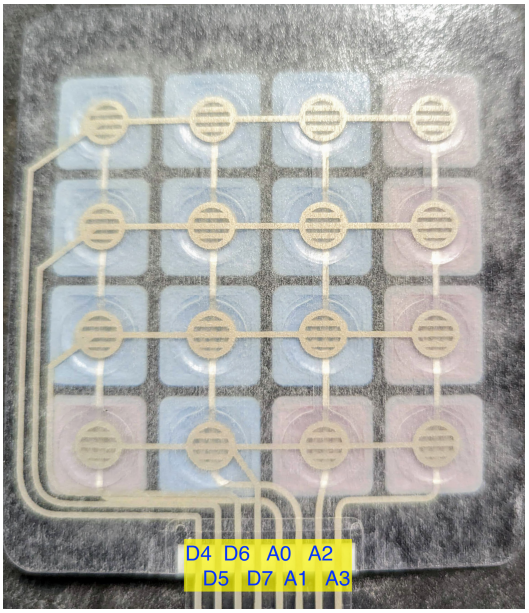
You may notice the code comparing `now` with the time a button or switch was last used and acting only if at least `BUTTON_NO_REPEAT_TIME` (500) or `DEBOUNCE_TIME` (20) milliseconds have passed. This serves two purposes.

Debouncing Mechanical buttons and switches demonstrate a phenomenon called *switch bounce*. This causes voltage to fluctuate for hundreds of microseconds when the contacts close or open. When this fluctuation is in the indeterminate region between the logical low and high thresholds, it can cause the logic level to "bounce" back-and-forth between high and low until settling into the final, correct logic level. This causes the digital circuitry or software to "see" multiple triggering events.

- The traditional way to debounce is to introduce a simple low-pass filter using a resistor and a capacitor. Hardware design can be simplified by solving a hardware problem with software, and so you will often see hobby projects with "debouncing code" such as `delay(2);` that pauses execution between detecting the first change of the button's or switch's position and acting upon it for 2ms, ample time for switch bounce to stabilize. The problem (beyond **`delay()`** being disallowed in this lab) is that 2,000µs is a long time to leave your system completely non-responsive.
- The solution used here allows your system to continue to respond to other external events. For example, you can press both buttons at very nearly the same time (or, unlikely, at the exact same time) and the software will react to both immediately.

Detecting single presses If we did nothing, then when we pressed a button, the code that detects the button press would see the low logic value every time **`loop()`** iterated, causing the message to print repeatedly until we lifted our finger. The solution used here allows a momentary press of a button to be treated as a single event but also allows the user to continue to hold the button down and have the software recognize that the user has been holding it down longer. (In this particular case, ignoring the button for a half-second also provides for software debouncing.)

(a) Front of matrix keypad.



(b) Keypad's underlying contact matrix.

Figure 4: The numeric keypad's header has four row pins and four column pins. Photographs and annotations by Bohn

4.1.3 Replacing Disallowed Function Calls

The **`testSimpleIO()`** function uses **`digitalRead()`** and **`digitalWrite()`**, two functions that are part of the Arudino core that you cannot use in this lab. You need to remove these functions.

- Use the `ioPorts` array to read from `A4`, `A5`, `D8`, and `D9` instead of using **`digitalRead()`**. Confirm that **`testSimpleIO()`** functions the same.
- Now use the `ioPorts` array to write to `D12` instead of using **`digitalWrite()`**. Confirm that **`testSimpleIO()`** functions the same.

4.2 Matrix Keypad

The matrix keypad has sixteen buttons but connects only to eight pins. Instead of reading the button presses directly, we scan the matrix to determine which row and which column the pressed button is in. We do this by setting logic levels on the rows and reading logic levels on the columns.

- Recall that `row1` is connected to `D4`, `row4` pin to `D5`, `row7` to `D6`, and `row*` to `D7`.
- Recall that `column1` is connected to `A0`, `column2` pin to `A1`, `column3` to `A2`, and `columnA` to `A3`.

4.2.1 Scanning the Keypad

There are a few options for obtaining the value corresponding to a key that is pressed on the keypad. The most efficient for a simple application such as the Number Building Tool is to use a lookup table. Starting on line 32 of *PollingLab.ino* you will find a 2-dimensional array that will serve as the lookup table.

```
// Layout of Matrix Keypad
//         1 2 3 A
//         4 5 6 B
//         7 8 9 C
//         * 0 # D
// This array holds the values we want each keypad button to correspond t
const uint8_t keys[4][4] = {
  {},
  {},
  {},
  {}
};
```

The element `keys[0][0]` will correspond to the `1` key; `keys[0][3]` will correspond to the `A` key; `keys[3][0]` will correspond to the `*` key; and `keys[3][3]` will correspond to the `D` key.

We want the numerals `0-9` to produce their respective decimal (and hexadecimal) values. We want `A-D` to produce their respective hexadecimal values. We want `#` to produce the hexadecimal value 0xE, and we want `*` to produce the hexadecimal value 0xF.

- Populate `keys`' nested array initializer so that the lookup table will produce the correct value for each row/column combination.

The first step in reading a value from the keypad is determining whether a key has been pressed. The **`cowpi_setup`** macro configured pins `A0-A3` (connected to the columns) as input pins that pull their logic values high when nothing forces them low. It also configured pins `D4-D7` (connected to the rows) as output pins that initially hold the value 0.

If no key has been pressed, then the column pins `A0-A3` will all have the value 1 because nothing is pulling their logic values low. When a button is pressed, the button's column will be electrically connected to the button's row. Since the row pins `D4-D7` the value 0, this will cause the button's column pin to take on the value 0. Thus, checking whether *any* button has been pressed can be accomplished by polling pins `A0-A3` to determine if each holds the value 1 or if at least one of them holds the value 0.

- Find the **`if`** block starging on line 56 in *PollingLab.ino*. Un-comment this **`if`** block.

```
// if ((...) && (millis() - lastKeypadPress > BUTTON_NO_REPEAT_TIME
//   uint8_t keypress = getKeyPressed();
//   if (keypress < 0x10) {
//     Serial.print("Key pressed: ");
```

```
      //      Serial.println(keypress, HEX);
      //      // displayData(1, sevenSegments[keypress]);
      //    } else {
      //      Serial.println("Error reading keypad.");
      //    }
      // }
```

- Replace the `...` on line 56 with a conditional expression that evaluates to `true` whenever at least one of the column pins holds the value 0.

Now locate **`getKeyPressed()`**.

```
uint8_t getKeyPressed() {
  uint8_t keyPressed = 0xFF;
  unsigned long now = millis();
  if (now - lastKeypadPress > DEBOUNCE_TIME) {
    lastKeypadPress = now;
    // ...
  }
  return keyPressed;
}
```

- Inside the **`if`** block, add a **`for`** loop that iterates over the four rows.
- At the start of the loop body, set each row pin to output 1, except that the pin corresponding to the current iteration's row should output 0. For example, in the first iteration, you want `D5`, `D6`, and `D7` to output a 1 but `D4` to output a 0. You will probably find it easier to do this in two lines, first setting *all* of the row pins to 1 and then setting the current iteration's row pin to 0.
- Poll the column pins. If all column pins are still 1, then the key being pressed is not in the current iteration's row. On the other hand...
- If one of the column pins is 0, then the key being pressed is in the current iteration's row. If this is the case, then determine which column the key is in based on which column pin is has the value 0.
- Knowing the key's row and column, use the lookup table to assign the correct value to the variable `keyPressed`.
  - In case you make a mistake when writing this code, you may wish to add error-handling code to make sure you do not make an assignment to `keysPressed` when you shouldn't have done so.
- After the **`for`** loop termintes, set the row pins back to 0.

This design does *not* handle the case of the user pressing multiple keys simultaneously. Handling multiple simultaneous keypresses is a considerably more-challenging problem.

4.2.2 Testing Your Implementation

If you implemented the keypad code correctly, then after you upload *PollingLab.ino* to your Arduino Nano, you will see on the Serial Monitor a message with the key value whenever you press a key on the keypad. If it does not, examine your code for errors. Adding debugging `Serial.print()`/`Serial.println()` statements may help.

4.3 7-Segment Display Module

The 7-Segment Display Module consists of a MAX7219[8] integrated circuit that acts as a peripheral using the SPI protocol and that drives the eight 7-segment displays.

The *Serial-Parallel Interface* (SPI) is a protocol that allows data to be sent and received serially (over a single wire) that should be used in parallel. This protocol is common enough that the ATmega328P microcontroller used by the Arduino Nano implements the protocol in hardware (see chapter 18 of the ATmega328P datasheet).

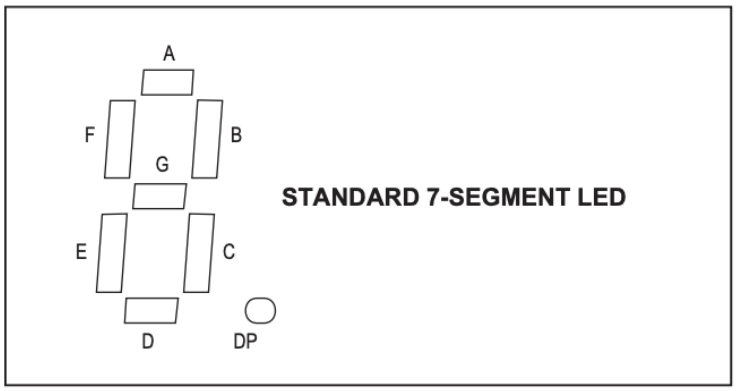
> NOTE: The original SPI specification used the terms *MOSI* and *MISO* to describe its two modes, and alternately used *SS* and *CS* to describe the signal that a device should listen for input. The preferred terminology now is *COPI* (Controller Output, Peripheral Input), *CIPO* (Controller Input, Peripheral Output) to describe the modes in devices that can use either mode (such as the Arduino Nano), *SDO* and *SDI* to describe the modes in devices that can only use one mode (the display module uses SDI), and *CS* (Chip Select) exclusively to describe the signal indicating that a device should listen for input.
>
> We will use the preferred terminology; however, legacy documentation – including the ATmega328P and MAX7219 datasheets – still use the original terminology.

The MAX7219 receives the serial data into a shift register and then latches the shift register's bits in parallel into one of eight memory locations or one of five control registers. The **`cowpi_setup`** macro takes care of configuring the MAX7219's control registers. Your task will be to write the code that sends data to the display module.

Each of the eight memory locations in the MAX7219 each correspond to one of the eight digits. The least-significant digit has the address 1, and the most- significant digit has the address 8. The bit pattern in each of these addresses indicates which LEDs in the 7-segment display should be illuminated (see Figure 5).

Seven-segment displays are so-named because (not including the decimal point) there are seven segments that can be activated/deactivated in combinations to form the ten decimal numerals (and, with a little imagination, most of the letters in the Latin alphabet). In the case of our display module, the segments are LEDs; you might also see segmented displays using LCDs or flip panels.

[8] https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf



	REGISTER DATA							
	D7	D6	D5	D4	D3	D2	D1	D0
Corresponding Segment Line	DP	A	B	C	D	E	F	G

Figure 5: Mapping of bits to 7-segment LEDs. Copied from MAX7219 Data Sheet, Table 6

Bit	7	6	5	4	3	2	1	0
SPDR 0x2E (0x4E)	MSB	...	...	...	...	...	...	LSB
SPSR 0x2D (0x4D)	SPIF	WCOL	not used	not used	not used	not used	not used	SPI2X
SPCR 0x2C (0x4C)	SPIE	SPE	DORD	Controller	CPOL	CPHA	SPR1	SPR0

Table 1: SPI Data, Status, and Control Registers. Adapted from ATmega382P Data Sheet, §18.5.

4.3.1 ATmega328P SPI Registers

The ATmega328P uses three registers for SPI, `SPCR` which is used to configure the SPI hardware, `SPSR` which is used to indicate the status of a data transfer, and `SPDR` which is used to transfer data to and from an SPI peripheral device (see Table 1).

The **`cowpi_setup`** macro enables SPI by setting the `SPE` bit to 1, sets the Arduino Nano as the controller by setting the Controller (née MSTR) bit to 1, and sets the SPI clock at 1MHz by setting the SPR1 and SPR0 bits to 01. All remaining bits in `SPCR` were set to 0.

4.3.2 Accessing SPI Registers in Memory Address Space

The `cowpi_spiRegisters` has three fields: `control`, `status`, and `data`, corresponding to the `SPCR`, `SPSR`, and `SPDR` registers, respectively.

```
typedef struct {
  volatile uint8_t control;                    // SPCR
  volatile uint8_t status;                     // SPSR
  volatile uint8_t data;                       // SPDR
} cowpi_spiRegisters;
```

Figure 6: Segments to be activated for the sixteen numerals. The first row shows the segments to be activated; the second row shows all segments to more clearly show which segment is which.

There is a global variable on line 18 of *PollingLab.ino*, `spi`, that we will use to access the SPI registers. If you look in *cowpi.h*, you will see a comment noting that the SPI registers start 0x2C bytes above `cowpi_IObase`.

```
* PROTOCOLS
* SPI            cowpi_spiRegisters at cowpi_IObase + 0x2C (0x4C)
```

- On line 51 of *PollingLab.ino*, find this line in **`setup`**: *`// spi = ...`*
- Remove the comment mark and ellipses, and assign to `spi` the address 0x2C bytes above `cowpi_IObase`; you will need to cast the resulting address to `(cowpi_ioPortRegisters *)`.

4.3.3 Displaying Values

We will use a lookup table to determine the bit patterns that need to be sent to the display module. Starting on line 43 of *PollingLab.ino* you will find a 1-dimensional array that will serve as the lookup table.

```
// Seven Segment Display mapping between segments and bits
// Bit7 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 Bit0
//  DP   A    B    C    D    E    F    G
// This array holds the bit patterns to display each hexadecimal numeral
const uint8_t sevenSegments[16] = {

};
```

The element `sevenSegments[0]` will contain the bit pattern for the numeral 0, `sevenSegments[1]` will contain the bit pattern for the numeral 1, and so on through `sevenSegments[15]` which will contain the bit pattern for the hex numeral F. For reference, Figure 6 shows the desired segments to be activated for each of the sixteen numerals. (Clearing a digit is achieved by sending 0x00 to that digit's corresponding memory address, deactivating all segments for that digit. You do not need to include this in the lookup table.)

In **`displayData()`**:

- Use the `ioPorts` array to set `D10` to 0, instructing the display module to listen for data from the Arduino Nano.

- The MAX7219 expects the address to be sent before the value. Use the `spi` struct's `.data` field to place `address` in the SPI Data Register.
- You should not place any new data into the SPI Data Register until the previous data has been sent; the `SPIF` bit in the SPI Status Register is a 0 when data is being sent and is a 1 when the data has been sent. Write a **`while`** loop that does nothing except continue to iterate while the `SPIF` bit is a 0 (use the `spi` struct's `.status` field to poll the SPI Status Register).
- Now use the `spi` struct to place `value` in the SPI Data Register.
- Write another **`while`** loop that does nothing except continue to iterate while the `SPIF` bit is a 0.
- Use the `ioPorts` array to set `D10` to 1, instructing the display module to latch the data into its memory.

4.3.4 Testing Your Implementation

On line 61 of *PollingLab.ino*, find this line in **`loop`**:
`// displayData(1, sevenSegments[keypress]);`. If you correctly implemented the keypad code and the display code then any key you press on the matrix keypad will display in the least-significant digit of the seven-segment display module. If it does not, examine your code for errors. Adding debugging `Serial.print()`/`Serial.println()` statements may help.

4.4 Completing Demonstration Mode

Most of the code for Demonstration Mode is now complete.

- Either remove the call to **`testSimpleIO()`** in **`loop()`** or remove the line in **`testSimpleIO()`** that causes the external LED to illuminate, since this is *not* the behavior we want for the external LED.
- Add code to cause the external LED to illuminate when a key on the keypad is pressed and to deluminate 500ms later.
- Add code so that demonstration mode only executes when the left switch is in the left position.
- Add code so that when the right switch is in the right position, only decimal numerals (*0-9*) are displayed.
- Add code so that when the user presses the right pushbutton, all digits on the display module are cleared.

Figure 7: Segments to be activated for error messages. The first row shows the segments to be activated; the second row shows all segments to more clearly show which segment is which.

5 Implementing Building Mode

When the system is not in demonstration mode, it is in building mode.

The best way to tackle building mode is to break it into bite-sized subproblems. Start by implementing the code to build the value in accordance with requirement 7a. Here are some recommendations:

- Start without considering the negation button.
- Keep an array of the bit patterns to be displayed separate from the actual value being built. While it won't matter as much in when using the hexadecimal number base, constantly re-calculating the BCD representation of a value every time a key is pressed in decimal mode may become a performance bottleneck since division (and modulo) is not implemented in hardware *and* even the (8-bit) hardware- implemented arithmetic will require several processor cycles for 32-bit arithmetic on the value.
- When a key is pressed, update both the value being built, the array of display bit patterns, and the actual display. The array can be updated simply by moving existing bit patterns into other positions in the array.
  - Some students in the past recalculated the value being built after each keypress by applying a decimal (or hexadecimal) weighted-sum of the individual digits. This works but requires seven multiplications and seven additions. *Updating* the value being built only requires one multiplication and one addition.
  - Don't forget to print the value being built to the Serial Monitor.
- Conveniently, you can also use the code to update the display array to detect when you need to display too bIG instead of making comparisons to the value being built.

Now that you have that code working, add code to negate the value whenever the left pushbutton is pressed. It is possible to quickly update the display array without using any arithmetic other than negating the value (how this is done differs between the decimal and hexadecimal number bases), but if you need to re-calculate the display array that is fine. Don't forget to update the actual display, too.

Finally, remove any debugging `Serial.print()`/`Serial.println()` statements so that when in builder mode, only the value being built is printed to the Serial Montior.

Turn-in and Grading

When you have completed this assignment, upload *PollingLab.ino* to Canvas.

This assignment is worth 40 points.

Rubric:

_____ **+8** Simple input/output functions correctly.

_____ **+10** Matrix keypad functions correctly.

_____ **+8** 7-Segment Display Module functions correctly.

_____ **+1** System is in demonstration mode only when left switch is in left position and builder mode when the left switch is in the right position.

_____ **+1** External LED illuminates when key is pressed and deluminates 500ms later if in demonstration mode. (This behavior is unspecified when in building mode.)

_____ **+2** When in demonstration mode, the numeral corresponding to the key pressed is displayed in the least-significant digit on the Display Module, in accordance with requirements 3, 4, and 6c.

_____ **+1** When in demonstration mode, the right pushbutton clears the display.

_____ **+2** Builds a value consistent with requirements 3 and 7a when in building mode and using the decimal number base.

_____ **+2** Builds a value consistent with requirements 4 and 7a when in building mode and using the hexadecimal number base.

_____ **+1** Negates value in the decimal number base when left pushbutton is pressed while in building mode.

_____ **+1** Negates value in the hexadecimal number base when left pushbutton is pressed while in building mode.

_____ **+1** Displays correct message when the number being built is too big.

_____ **+1** When in building mode, prints the number being built (and nothing else) to the Serial Monitor. (This behavior is unspecified when in demonstration mode.)

_____ **+1** When in building mode the right pushbutton clears all digits on the Display Module except the least-significant digit, which displays 0.

_____ **Bonus +2** Get assignment checked-off by TA or professor during office hours before it is due. (You cannot get both bonuses.)

_____ **Bonus +1** Get assignment checked-off by TA at *start* of your scheduled lab immediately after it is due. (Your code must be uploaded to Canvas *before* it is due. You cannot get both bonuses.)

Penalties

_____ **-8** Simple input/output configuration and test function relies on code that violates the constraints in Section 3.

_____ **-10** Obtaining values from matrix keypad relies on code that violates the constraints in Section 3.

_____ **-8** Sending values to the 7-segment display module relies on code that violates the constraints in Section 3.

_____ **-5** Code associated with demonstration mode (other than that covered in the first three penalty items) relies on code that violates the constraints in Section 3.

_____ **-8** Code associated with conversion mode (other than that covered in the first three penalty items) relies on code that violates the constraints in Section 3.

_____ **-1** for each **`goto`** statement, **`continue`** statement, **`break`** statement used to exit from a loop, or **`return`** statement that occurs within a loop.

Epilogue

Herb looks over your work. "Hmm, yes. I think this is coming along nicely. Let's run a few more tests."

Archie storms into the room. "We have *got* to do something about security! How's that doodad coming along? Because there's now a half-man/half-fly in the labs going on-and-on about Chaos Theory and how if we just give him a MacBook and a spaceship then he'll be able to get the Lord of Thunder to travel across the 8th Dimension. Is that thing just about ready?"

Herb shakes his head, "No, not quite yet. It should be ready in about a week."

To be continued...

Appendix: Lab Checkoff

You are not required to have your assignment checked-off by a TA or the professor. If you do not do so, then we will perform a functional check ourselves. In the interest of making grading go faster, we are offering a small bonus to get your assignment checked-off at the start of your scheduled lab time immediately after it is due. Because checking off all students during lab would take up most of the lab time, we are offering a slightly larger bonus if you complete your assignment early and get it checked-off by a TA or the professor during office hours.

( ) Establish that the code you are demonstrating is the code you submitted to to Canvas.

- If you are getting checked-off during lab time, show the TA that the file was submitted before it was due.
- Download the file into your PollingLab directory. If necessary, rename it to *PollingLab.ino*.

( ) Upload *PollingLab.ino* to your Arduino Nano.

If you completed demonstration mode, jump ahead to *Demonstration Mode*. Successfully completing demonstration *prima facie* shows that the I/O functions correctly.

If you did not complete demonstration mode:

1. ( ) Using `testSimpleIO()`, show on the Serial Monitor that the left and right button presses are detected.
2. ( ) Using `testSimpleIO()`, show on the Serial Monitor that the left and right switches' position changes are detected.
3. ( ) Show that the external LED illuminates when it is supposed to (in a bright room, you may need to shade the LED with your hand). You can do this either by moving both switches to the right position (using the code in `testSimpleIO()`) or by showing the LED illuminate when you press a key on the matrix keypad (if you started working on demonstration mode).
4. ( ) Show on the Serial Monitor that key presses on the matrix keypad are correctly decoded.
5. ( ) Show on the display module that values decoded from the matrix keypad are displayed as the correct characters.

Demonstration Mode

6. ( ) Place both switches in the left position. The display module is initially blank.

7. ( ) Press a number key on the matrix keypad. The corresponding character appears on the display module, and the external LED illuminates for approximately one-half of a second.

8. ( ) Press a different number key on the matrix keypad. The corresponding character appears on the display module, and the external LED illuminates for approximately one-half of a second.

9. ( ) Press a letter key on the matrix keypad. The display module is unchanged.

10. ( ) Place the right switch in the right position. The display is unchanged.

11. ( ) Press a letter key on the matrix keypad. The corresponding character appears on the display module, and the external LED illuminates for approximately one-half of a second.

12. ( ) Press the right pushbutton. The display module blanks.

13. ( ) Press a key on the matrix keypad. The corresponding character appears on the display module, and the external LED illuminates for approximately one-half of a second.

14. ( ) Press the same key on the matrix keypad. The display module is unchanged (still displays the corresponding character), but the external LED still illuminates for approximately one-half of a second.

Building Mode

15. ( ) Place the left switch in the right position. The display remains unchanged.

16. ( ) Press the right pushbutton. The display shows
    0 (additional leading 0s are allowed)
    and the Serial Monitor shows
    0

17. ( ) Place the right switch in the left position. The display remains unchanged.

18. ( ) Press 2, then 3.
    2 2
    23 23

19. ( ) Press B. The display is unchanged.

20. ( ) Press the left pushbutton, then 1.
    -23 -23
    -231 -231

21. ( ) Press the left pushbutton, then 4.

231	231
2314	2314

22. ( ) Press D. The display is unchanged.

23. ( ) Press the right pushbutton, and place the right switch in the right position.

0	0

24. ( ) Press #, then 7.

E	E
E7	E7

25. ( ) Press the left pushbutton.

FFFFFF19	FFFFFF19

26. ( ) Press 0, then A.

FFFFF190	FFFFF190
FFFF190A	FFFF190A

27. ( ) Press the left pushbutton, then *.

E6F6	E6F6
E6F6F	E6F6F

28. ( ) Press B, then C, then D, then 0.

E6F6Fb	E6F6FB
E6F6Fbc	E6F6FBC
E6F6Fbcd	E6F6FBCD
too big	too big

29. ( ) Press the right pushbutton, and place the right switch in the left position.

0	0

30. ( ) Press 5, then 6, then 7, then 8, then 9, then 0, then 1, then 2, then 3.

5	5
56	56
567	567
5678	5678
56789	56789
567890	567890
5678901	5678901
56789012	56789012
too big	too big

31. ( ) Press the right pushbutton.

0	0

32. ( ) Press 3, then the left pushbutton, then 4, then 5, then 6, then 7, then 8, then 9, then 0.

3	3
-3	-3
-34	-34
-345	-345
-3456	-3456
-34567	-34567
-345678	-345678
-3456789	-3456789
too big	too big

This concludes the demonstration of your system's functionality. The TAs will later examine your code for violations of the assignment's constraints.