

Lab 9

Using Polling with Memory-Mapped Input/Output

Due: Before the start of your lab section on November 10, 15, or 16

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 a number-base conversion tool 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 1 has the functional specification of the system you will develop. Section 2 describes implementation constraints. Section 3 guides you in implementing the first portion of the system, and Section 4 offers suggestions in implementing the second portion of the system.

1 Number-Base Conversion Tool Specification

1. The tool shall have two modes, *demonstration mode* and *conversion mode*.
2. When the **left switch** is toggled to the left, the tool is in demonstration mode. When the tool 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 2a.
 - (c) Initially, no digits on the **7-segment display module** shall have any of their segments illuminated.
 - (d) Whenever the user presses a button on the **matrix keypad**, the corresponding hexadecimal digit shall be displayed in the least-significant digit position of the **7-segment display module**, replacing any digit previously displayed.
3. When the **left switch** is toggled to the right, the tool is in conversion mode. When the tool is in conversion mode:
 - (a) The tool shall have two sub-modes, *decimal* and *hexadecimal*.
 - (b) When the **right switch** is toggled to the left, the tool is in decimal sub-mode. When the **right switch** is toggled to the right, the tool is in hexadecimal sub-mode.
 - (c) Initially, no digits on the **7-segment display module** shall have any of their segments illuminated.
 - (d) Whenever the user presses a button on the **matrix keypad**, the corresponding digit 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.
 - i. If the tool is in hexadecimal sub-mode, hexadecimal digits shall be displayed.
 - ii. “0x” shall not be displayed as part of a hexadecimal value.
 - iii. If the tool is in decimal sub-mode, decimal digits shall be displayed. If the user had pressed one of the *A-F* buttons, it shall be ignored.
 - iv. 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.
 - v. A positive sign shall not be displayed as part of a positive decimal value.
 - vi. Leading 0s shall not be displayed. For example, 782 is correctly displayed, but 00000782 is not correctly displayed.

- vii. 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 0xEFFF,FFFF in hexadecimal, less than 0x8000,000 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 `ERROR`.
- (e) Whenever the user toggles the **right switch**, the value previously displayed on the **7-segment display module** shall be re-displayed according to the new sub-mode. For example, if the tool is in decimal sub-mode and is displaying `1234` then after the user changes the tool to hexadecimal sub-mode, `4D2` shall be displayed. Similarly, if the tool is in hexadecimal sub-mode and is displaying `1234` then after the user changes the tool to decimal sub-mode, `4660` shall be displayed.
 - i. If the value is too great to be displayed in the new sub-mode, then `ERROR` shall be displayed.
- (f) Whenever the user presses the **left pushbutton**, the value being displayed shall be negated.
 - i. In decimal sub-mode, the presence or absence of negative sign shall indicate whether a value is negative or not.
 - ii. In hexadecimal sub-mode, 32-bit two's complement shall be used.
- 4. Regardless of mode, whenever the user presses the **right pushbutton**, the **7-segment display module** shall be cleared. If the system is in conversion mode then pressing the **right pushbutton** will also clear the value.
- 5. 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.

2 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 core¹ (unless they are also part of the C standard), nor AVR-specific functions, macros, types, or constants of avr-libc².

¹<https://www.arduino.cc/reference/en/>

²<https://www.nongnu.org/avr-libc/user-manual/index.html>

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: ATmega328 I/O port registers. Cropped from ATmega328P Data Sheet, §30

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.

(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.)

3 Implementing Demonstration Mode

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

3.1 Simple Input/Output

The ATmega328 microcontroller⁷ 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, and Table 1 shows how to configure the specific pins.

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 we wish to use D10 as an output pin, then we would set register DDRB’s bit 2 to a 1 in our `setup()` function, and then our `loop()` function (or other functions it

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/>

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

DDR xn	PORT xn	Direction	High/Low Value
0	0	Input, Hi-Z	Takes on value from input device
0	1	Input, Pull-Up	High unless input device pulls value low
1	0/1	Output	Value assigned by program

Table 1: Configuring I/O port pins. Adapted from ATmega382P Data Sheet, Table 13-1.

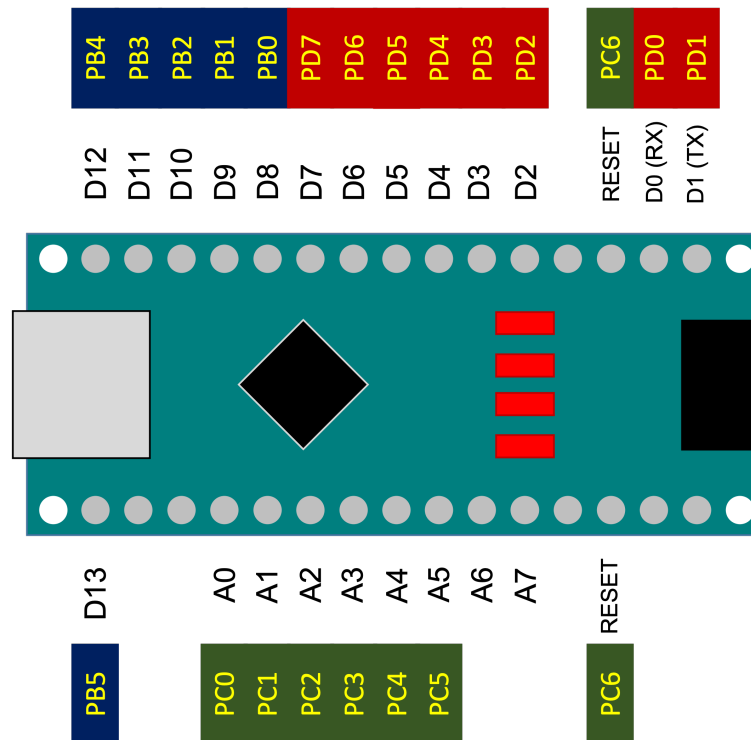


Figure 3: Mapping of Arduino Nano pins to ATmega328 input/output ports. Diagram by Bohn

calls) would set register PORTB's bit 2 to a 0 or 1 as needed, based on which logic level is needed by the output device.

On the other hand, if we wanted to use D10 as an input pin, then we would set register DDRB's bit 2 to a 0 in our **setup()** function. If we wanted the logic value to be whichever logic value is set by the input device then we would place the pin in a "high-impedance" state by setting PORTB's bit 2 to 0. (Note that if the pin is in high-impedance input mode and a logic value is not being set by an input device, the pin's value will "float" and will change at the whims of stray electrons.) If we wanted the logic level to normally be high but can be pulled low when an input device grounds the pin, then we would set PORTB's bit 2 to 1, and an internal pull-up resistor will prevent a short circuit. You would use high-impedance input mode when the input device is able to set both high and low values, such as with a dual-throw switch or an active input device; you would use pull-up input mode when the input device is able to set only one value, such as with a single-throw switch or

button.

3.1.1 Accessing I/O Registers

Open *PollingLab.ino* in the Arduino IDE. There is a global variable, `gpio`, that you will use to access the input/output registers for the external pins. If you look in *cowpi.h*, you will see a constant pointer, `IObase` that holds the memory address that is the start of ATmega328's memory-mapped input/output register bank. On line 10, you will see a comment noting that the external pins' input/output registers start 3 bytes above `IObase`.

- On line 43 of *PollingLab.ino*, find this line in **setup**: `// gpio = ...`
- Remove the comment mark and ellipses, and assign to `gpio` the address 3 bytes above `IObase`.

You can now use `gpio` as a 3-element array to access each of the three I/O ports. Notice in *cowpi.h* there are three constants defined to allow you index this array without having to remember which I/O port corresponds to which set of external pins. Each element of the `gpio` array is a **struct** `gpio_registers` with three fields: `input`, `output`, and `direction`, corresponding to the `PINx`, `PORTx`, and `DDRx` registers, respectively.

- Recall that the left switch is connected to pin A4 and the right switch is connected to pin A5.
- In the `setup_simple_io()` function in *PollingLab.ino*, use the `gpio` array to set pins A4 and A5 to **high-impedance input**. Refer to Figures 2 and/or 3 to determine which bits need to be set. Use Table 1 to determine which values those bits need to be set to. *Take care not to change bits that you don't need to change.*
- Recall that the left pushbutton is connected to pin D8 and the right pushbutton is connected to pin D9.
- In `setup_simple_io()`, use the `gpio` array to set pins D8 and D9 to **pulled-up input**.
- Recall that the external LED is connected to pin D12 through a current-limiting resistor.
- In `setup_simple_io()`, use the `gpio` array to set pins D12 to **output**.

3.1.2 Testing Your Configuration

The starter code includes a `test_simple_io()` 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). You can use this function to check whether you wrote `setup_simple_io()` correctly.

You may notice the code comparing `now` with the time a button or switch was last used and acting only if at least 500ms 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 even though 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 hold the button down and have the software recognize that the user has been holding it down longer.

3.1.3 Replacing Disallowed Function Calls

The `test_simple_io()` function uses `digitalRead()` and `digitalWrite()`, two functions that are part of the Arduino core that you cannot use in this lab. After you’ve satisfied yourself that you correctly configured pins A4, A5, D8, D9, and D12, you need to remove these functions.

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

3.2 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.
- In the `setup_keypad()` function, use the `gpio` array to set pins D5, D6, D7, and D8 to **output**. Refer to Figures 2 and/or 3 to determine which bits need to be set. Use Table 1 to determine which values those bits need to be set to. *Take care not to change bits that you don't need to change.*
- In the `setup_keypad()` function, use the `gpio` array to set the initial output values of D5, D6, D7, and D8 to 0.
- Recall that `column1` is connected to A0, `column2` pin to A1, `column3` to A2, and `columnA` to A3.
- In `setup_keypad()`, use the `gpio` array to set pins A0, A1, A2, and A3 to **pulled-up input**.



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

3.2.1 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 Conversion Tool is to use a lookup table. Starting on line 24 of *PollingLab.ino* you will find a 2-dimensional array that will serve as the lookup table.

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. If no key has been pressed, then column pins A0- A3 will all have the value 1 because

you configured the pins to pull their logic values high when nothing forces them low. When a button is pressed, the button's column will be electrically connected to the button's row. Since you configured row pins D4-D7 to initially hold 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 starting on line 52 in *PollingLab.ino*. Un-comment this **if** block.
- Replace the `...` on line 52 with a conditional expression that evaluates to **true** whenever at least one of the column pins holds the value 0.

Now locate `get_key_pressed()`. Notice that it has timing code similar to what you saw in `test_simple_io()` that will treat a keypress that lasts less than 500ms as a single keypress.

- 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 D6, D7, and D8 to output a 1 but D5 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 `key_pressed`.
 - 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 `keys_pressed` when you shouldn't have done so.
- After the **for** loop terminates, 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.

3.2.2 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.

3.3 7-Segment Display Module

The 7-Segment Display Module consists of a MAX7219⁸ 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 ATmega328 microcontroller used by the Arduino Nano implements the protocol in hardware (see chapter 18 of the ATmega328 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 ATmega328 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 starter code takes care of configuring the MAX7219's control registers. Your task will be to write the code that sends data to the display module.

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>

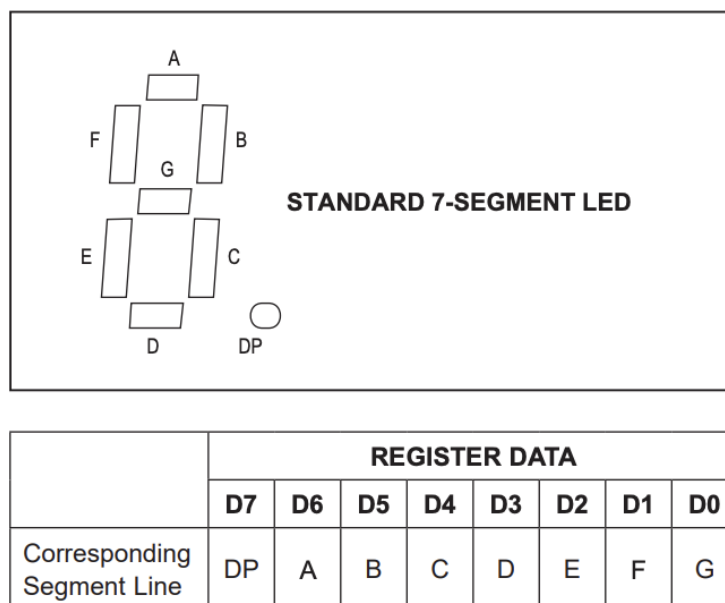


Figure 5: Mapping of bits to 7-segment LEDs. Copied from MAX7219 Data Sheet, Table 6

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

3.3.1 Configuring the Arduino Nano for SPI

There is a global variable in *PollingLab.ino*, `spi`, that you will use to access the SPI registers for the external pins. If you look in *cowpi.h*, you will see on line 21, you will see a comment noting that the SPI registers start 0x2C bytes above `IObase`.

- On line 44 of *PollingLab.ino*, find this line in **setup**: `// spi = ...`
- Remove the comment mark and ellipses, and assign to `spi` the address 0x2C bytes above `IObase`.

The **struct** `spi_registers` has three fields: `control`, `status`, and `data`, corresponding to the `SPCR`, `SPSR`, and `SPDR` registers, respectively (see Table 2). We need to enable SPI by setting the `SPE` bit to 1, set the Arduino Nano as the controller by setting the `Controller` (née `MSTR`) bit to 1, and set the SPI clock at 1MHz by setting the `SPR1` and `SPR0` bits to 01. All remaining bits in `SPCR` should be 0.

- Replace the first line in **setup_display_module()** that says
`// Set COPI, SCK, and CS to output`
 with code that uses the `gpio` array to set pins D11, D13, and D10 to output.



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.

- Replace the second line in `setup_display_module()` that says *// Enable SPI, Controller, set clock rate fck/16* with code that uses the `spi` struct to set the bits in the SPI Control Register to the desired values.

3.3.2 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 35 of *PollingLab.ino* you will find a 1-dimensional array that will serve as the lookup table. The element `seven_segments[0]` will contain the bit pattern for the numeral `0`, `seven_segments[1]` will contain the bit pattern for the numeral `1`, and so on through `seven_segments[15]` which will contain the bit pattern for the hex numeral `F`. For reference, Figure 6 shows the desired segments that 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. This is demonstrated in `setup_display_module()`. You do not need to include this in the lookup table.)

In `display_data()`:

- Use the `gpio` array to set D10 to 0, instructing the display module to receive data.
- The MAX7219 expects the address to be sent before the value. Use the `spi` struct 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 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 `gpio` array to set D10 to 1, instructing the display module to latch the data into its memory.

3.3.3 Testing Your Implementation

On line 57 of *PollingLab.ino*, find this line in `loop`:

`// display_data(1, seven_segments[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.

3.4 Completing Demonstration Mode

Most of the code for Demonstration Mode is now complete.

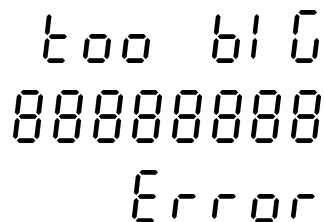
- Either remove the call to `test_simple_io()` in `loop()` or remove the line in `test_simple_io()` that causes the external LED to illuminate, since this is *not* the behavior we want for the external LED.
- Add code to cause the external LED to illuminate when a key on the keypad is pressed and to 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 system enters demonstration mode, all digits on the display module are cleared.
- Add code so that when the user presses the right pushbutton, all digits on the display module are cleared.

4 Implementing Conversion Mode

When the system is not in demonstration mode, it is in conversion mode. Add code so that when the system enters conversion mode (when the left switch is moved to the right position), all digits on the display are cleared. Add code to the “clearing” code (at least to the “clearing” code associated with the right pushbutton) that also clears the value being built.

The best way to tackle conversion mode is to break it into bite-sized subproblems. Start by implementing the code to build the value in accordance with requirement 3d. 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 hexadecimal sub-mode, constantly recalculating the BCD representation of a value every time a key is pressed in decimal



```

      too big
      88888888
      Error
  
```

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

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.
- 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 decimal and hexadecimal sub-modes), but if you need to re-calculate the display array that is fine. Don't forget to update the actual display, too.

Finally, convert between radices. When moving the right switch from the left position to the right position, convert the displayed value from decimal to hexadecimal. When moving the right switch from the right position to the left position, convert the displayed value from hexadecimal to decimal. Obviously, you will need to re-calculate the display array. Don't forget to update the actual display, too.

Turn-in and Grading

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

This assignment is worth 50 points.

Rubric:

_____ +10 Simple input/output functions correctly.

_____ +10 Matrix keypad functions correctly.

- _____ **+10** 7-Segment Display Module functions correctly.
- _____ **+1** System is in demonstration mode only when left switch is in left position and conversion 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 Conversion mode.)
- _____ **+2** All digits on the display module are cleared when the system enters demonstration mode, when the system enters conversion mode, or when the right pushbutton is pressed.
- _____ **+3** Builds a value consistent with requirement **3d** when in decimal sub-mode.
- _____ **+3** Builds a value consistent with requirement **3d** when in hexadecimal sub-mode.
- _____ **+1** Negates value in decimal sub-mode when left pushbutton is pressed.
- _____ **+1** Negates value in hexadecimal sub-mode when left pushbutton is pressed.
- _____ **+3** Converts from decimal to hexadecimal.
- _____ **+3** Converts from hexadecimal to decimal.
- _____ **+2** Conversion mode errors handled correctly.
- _____ **-10** Simple input/output configuration and test function relies on code that violates the constraints in Section **2**.
- _____ **-10** Obtaining values from matrix keypad relies on code that violates the constraints in Section **2**.
- _____ **-10** Sending values to the 7-segment display module relies on code that violates the constraints in Section **2**.
- _____ **-4** Code associated with demonstration mode (other than that covered in the first three penalty items) relies on code that violates the constraints in Section **2**.
- _____ **-16** Code associated with conversion mode (other than that covered in the first three penalty items) relies on code that violates the constraints in Section **2**.
- _____ **Bonus +2** Get assignment checked-off by TA or professor during office hours before it is due. (Cannot get both bonuses.)
- _____ **Bonus +1** Get assignment checked-off by TA at *start* of your scheduled lab immediately after it is due. (Cannot get both bonuses.)

NOTE: This document will be updated with instructions to get it checked-off if you pursue the bonus credit.