Lab 11

Using Interrupt-Driven Input/Output

Due: Week of May 2, Before the start of your lab section*

This is a team-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with your assigned partner(s), the professor, and the TAs. Sharing code with or copying code from a student who is not on your team, or from the internet, is prohibited.

In this assignment, you will write code for your Arduino Nano-based class hardware kit that will use interrupts from external devices and from a timer to drive some of the logic for a four-function calculator.

Figure 1: Interrupts. 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 – the integer calculator specification is required; the fractional calculator specification is extra-credit. Section 2 describes implementation constraints. Section 4 guides you in creating and handling timer interrupts using the AVR-libc `ISR()` macro, and Section 5 guides you in handling external interrupts using

*See Piazza for the due dates of teams with students from different lab sections.

the Arduino core **`attachInterrupt()`** function. Finally, Section 6 has a few thoughts on implementing the calculator functionality.

1 Four-Function Calculator Specification

Integer Calculator

1. The calculator shall be an infix[1] decimal calculator capable of performing addition, subtraction, multiplication, and division.
   - If only integer behavior is implemented, then division shall be integer division; *i.e.*, the fractional portion of quotients values shall be truncated.
2. A decimal point shall not be displayed.
3. Each arithmetic operation shall use two operands, referred in this specification as *operand1* and *operand2*.
   - Because no further history of operands is maintained, the algebraic order of operations is not preserved.
   - The value of *operand1* can only be changed as the result of a calculation or by clearing its value.
   - The value of *operand2* is "built" through keypresses on the matrix keypad.
   - *operand1* and *operand2* are defined in the problem domain. Your solution space might not have direct corollaries (for example, in the problem domain, *operand2* can be undefined, which is not a characteristic of any number types in C).
4. The **7-segment display module** shall display *operand1*'s value except when building *operand2*, except when an error condition is present, and except when the display is disabled.
5. Initially, *operand1* shall hold the value 0, *operand2* shall have no defined value, and no arithmetic operation shall be specified.
6. When building *operand2*:
   (a) When the user starts to build *operand2*, *operand1* shall no longer be displayed. Instead, the value being built shall be displayed.
   (b) Except as otherwise specified in requirement 16, whenever the user presses a numeral 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.

[1] https://en.wikipedia.org/wiki/Calculator_input_methods#Infix_notation

(c) Except as otherwise specified in requirement 16, whenever the user presses the **left pushbutton**, the value being displayed shall be negated.

- If *operand2* has a defined value (is being built) then *operand2* shall be negated.
- If *operand2* has no defined value (*operand1* is displayed) then *operand1* shall be negated.

(d) In no case shall the calculator allow the user to input a value requiring more digits than can be displayed on the **7-segment display module**. If the user attempts to enter a value that requires more digits, then the value shall remain unchanged.

7. If a negative 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.

8. A positive sign shall not be displayed as part of a positive decimal value.

9. Leading 0s shall not be displayed. For example, 782 is correctly displayed, but 00000782 is not correctly displayed. However, when the value to be displayed is 0, then 0 shall be displayed.

10. The `A` button indicates addition. The `B` button indicates subtraction. The `C` button indicates multiplication. The `D` button indicates division. The `#` button directs that the calculation shall be executed without specifying another arithmetic operation.

11. Except as otherwise specified in requirement 16, when the user presses the `#` button, directing that the calculation is to be executed, the previously-specified operation shall be performed, the resulting value shall become *operand1* (and shall be displayed), and there shall no longer be a valid *operand2*. If there was no previously-specified operation, then *operand2* shall become *operand1*, and there shall no longer be a valid *operand2*.

12. Except as otherwise specified in requirement 16, when the user presses an operation button (`A-D`), the previously-specified operation shall be performed, the resulting value shall become *operand1* (and shall be displayed), there shall no longer be a valid *operand2*, and the next operation to be performed shall be that which corresponds to the button pressed. If there was no previously-specified operation, then *operand2* shall become *operand1*, *operand2* shall no longer have a defined value, and the next operation to be performed shall be that which corresponds to the button pressed.

    - If the resulting value (*operand1*) is too great to be displayed, the **7-segment display module** shall display Error.
    - If *operand2* is 0 and the operation is division, the **7-segment display module** shall display Error.

13. The calculator shall not support scientific notation, neither for operand entry nor for displaying a result.

14. Except as otherwise specified in requirement 16, whenever the user presses the **right pushbutton**:
    (a) If *operand2* is being built, then *operand2* shall be reset to no defined value, the next operation (if any) shall be cleared, and *operand1* shall be displayed.
    (b) If *operand2* is not being built, then *operand1* shall be reset to 0 (and 0 shall be displayed), and the next operation (if any) shall be cleared.
15. Display timeout: If no button or key has been pressed for a designated amount of time, the **7-segment display module** shall go blank; however, *operand1*, *operand2* being built (if any), and the next operation (if any) shall be retained. When the **left switch** is in the left position, the designated time is exactly 30 seconds; when the **left switch** is in the right position, the designated time is exactly 5 seconds.
    - Do not place the microcontroller in Idle mode, nor any other Sleep mode.
16. If the display has timed out, then pressing any key or button will cause the display to resume displaying its previous output. The key/button press that takes the display out of its timed-out mode shall not otherwise affect the system's state; specifically, it shall not contribute to *operand2*'s value, it shall not cause an operation to be executed, it shall not specify an operation to be performed, nor shall it clear any operands or specified operations.

Figure 2 depicts a problem-space state diagram of the 4-function integer calculator, not including too-great values, and not including display's time-out and resume-display behavior.

Fractional Calculator

If fractional-value behavior is implemented, then the calculator's specification is amended and extended:

17. Requirement 2 is replaced with:
    A decimal point shall be displayed as necessary:
    (a) The decimal point, when displayed, shall not occupy a full display position in the **7-segment display module**; instead, it shall be occupy the `DP` segment of the same display position as the ones-place digit.
    (b) When *operand1* is displayed, it shall always include a decimal point.
    (c) When *operand2* is displayed, the decimal point shall not be displayed unless the ∗ button has been pressed (see requirement 21).
    (d) If the value being displayed is an integer value and a decimal point is required, then the decimal point shall be displayed after the least-significant digit (for example: 123.). If the value being displayed has a fractional portion, then the decimal point shall be displayed between the ones-place and the tenths-place (for example, $123\frac{1}{4}$ shall be displayed as 123.25).

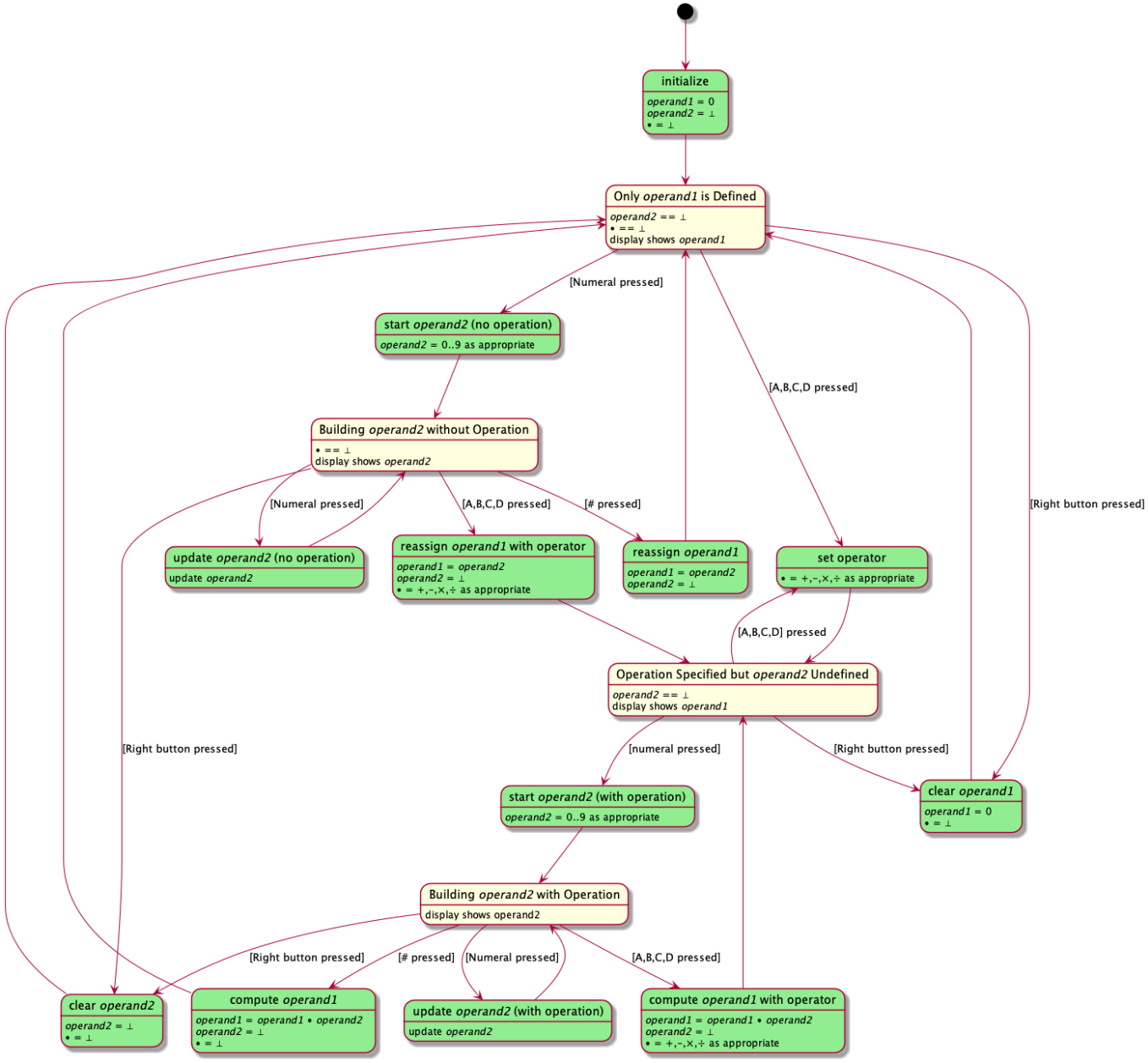

Figure 2: State diagram depicting "normal" behavior for integer calculator. Yellow states depict invariant conditions; green states depict updates to the problem-domain variables *operand1*, *operand2*, and • (operator). Diagram by Bohn

18. The clarification sub-bullet for requirement 1 is replaced with:
Division shall be conventional division; *i.e.*, the fractional portion of quotients, if any, shall be displayed in accordance with the rounding rule of requirement 22.

19. Requirement 9 is amended: If the value to be displayed is between -1.0 and 1.0, exclusive, then a 0 shall be displayed in the ones-place.

20. Trailing 0s in the fractional portion of a value shall not be displayed. For example, 23.18 is correctly displayed, but 23.180000 is not correctly displayed.

21. The * key can be thought of as the decimal point key; that is, any keys pressed before pressing the * key shall be part of the integer portion of the value being built, and any keys pressed after pressing the * key shall be part of the fractional portion of the value being built.

22. If the total number of digits required to display the integer portion and the fractional portion of *operand1* (including the negative sign if needed) exceeds the number of digits in the **7-segment display module**) then the value shall be rounded:

    (a) All display positions shall be used.

    (b) If the most-significant digit of the truncated portion is less than 5, then the value shall be rounded toward zero (*i.e.*, the least-significant displayed digit shall be unchanged). For example, 12345.6782 shall be displayed as 12345.678 after rounding.

    (c) If the most-significant digit of the truncated portion is greater than 5, or if that digit is 5 and there are more digits that follow, then the value shall be rounded away from zero. For example, both 12345.6789 and 12345.67851 shall be displayed as 12345.679 after rounding.

    (d) If the most-significant digit of the truncated portion is 5 and there are no digits that follow ("exactly halfway") then the value shall be rounded in whichever direction makes the least-significant displayed digit even (0, 2, 4, 6, or 8). For example, both 12345.6775 and 12345.6785 shall be displayed as 12345.678 after rounding.

    - If the value is too small to be displayed (the value is between -0.0000005 and 0.00000005, inclusive, which round to 0) then 0 shall be displayed. A negative value that rounds to 0 shall not preserve its negative sign.
    - If the value is too great to be displayed after rounding (the resulting integer portion requires more digits than are available) then the value shall not be rounded; instead, Error shall be displayed.

2 Constraints

You may continue to use the memory-mapped I/O registers, or you can use functions provided by the Arduino core read from and write to pins.

You may re-use code from the previous lab, or you can rewrite the necessary code using functions, macros, types, or constants that are part of the Arduino core;[2] functions, macros, types, or constants of avr-libc;[3] or one of Arduino's standard libraries.[4] You may *not* use a third-party library, not even one that the Arduino Libraries reference page links to (if it isn't among those available in the Arudino IDE's "Sketch" → "Include Library" menu under "Arduino libraries" then you may not use it).

You may *not* poll the matrix keypad nor the pushbuttons to determine if they have been pressed. You must use interrupts to determine if a key or button has been pressed. Once a press has been detected, you may scan the matrix keypad or read the pushbuttons to determine which key or button has been pressed.

While it is possible to configure the SPI hardware to generate an interrupt after the content of the SPI Data Register is transmitted, you may use your **`send_data()`** function that polls the `SPIF` bit.

You may poll the left switch to determine if its position has changed; however, the specification has been written such that your code should only need to occasionally check the switch's position rather than polling it for changes.

While you may use **`millis()`** for debouncing, you may *not* use **`millis()`** nor **`micros()`** to implement the display timeout. You must use an interrupt from the ATmega328's Timer1 or Timer2 as part of implementing the display timeout.

3 Getting Started

Open the Arduino IDE. Create a new sketch by selecting from the menu "File" → "New". Select "File" → "Save As…" and save it as *CalculatorLab*. Using File Explorer (Windows), Finder (MacOS), an equivalent file manager (Linux), or a command-line terminal, copy *cowpi.h* from your `Arduino/PollingLab` directory into your `Arduino/CalculatorLab` directory. In the Arduino IDE, add **`#include`** `"cowpi.h"` to the top of *CalculatorLab.ino*.

Optionally copy the following code from your *PollingLab.ino* into your *CalculatorLab.ino*:

- **`struct`** `gpio_registers *gpio`, **`struct`** `spi_registers *spi`, **`const`** `uint8_t keys[][]`, and **`const`** `uint8_t seven_segments[]`
- **`setup()`** (replacing the **`setup()`** function in *CalculatorLab.ino*), **`setup_simple_io()`**, **`setup_keypad()`**, and **`setup_display_module()`**

[2] https://www.arduino.cc/reference/en/

[3] https://www.nongnu.org/avr-libc/user-manual/index.html

[4] https://www.arduino.cc/reference/en/libraries/ The standard libraries are those under the heading, "Standard Libraries."

- **`get_key_pressed()`** and **`display_data()`**

If you need to rewrite any of the code from the previous lab, you may do so without explicitly using the memory-mapped I/O registers. You may instead use **`pinMode()`**[5,6] to configure the pins, and **`digitalRead()`**[7] and **`digitalWrite()`**[8] to perform input and output on specific pins. You may use the Arduino SPI library[9] to use the ATmega328's SPI hardware to send data to the display module, but you will probably find it easier to use **`shiftOut()`**[10] (see **`SendData()`** in the prelab's *DisplayTest.ino* for an example).

You may want to keep your code from the PollingLab's "conversion mode" handy, but don't copy it into *CalculatorLab.ino* since it uses polling to detect the user pressing keys and buttons.

4 Implementing Timeout

You must use timer interrupts from either Timer1 or Timer2 as part of your implementation of the display timeout. Without using an external clock source, you won't be able to configure an interrupt to occur every 30 seconds, or even every 5 seconds. Since we will not use an external clock source, you will need to use timer interrupts along with other logic.

4.1 Preparation

Design your logic and determine how often you need a timer interrupt to make it work. For example, the Arduino Nano's pseudorealtime clock relies on an interrupt from Timer0 every 1.024µs and advances the millisecond counter after 1,000 of these interrupts have occurred. The pseudorealtime clock uses an overflow-based timer interrupt to arrive at an approximation of milliseconds. The calculator's specification calls for the display module going black exactly 5 seconds or exactly 30 seconds (depending on the switch position) after the last key press or button press. To achieve exactness, you will use a comparison-based timer.

You can then determine the parameters for the timer using this equation:

$$16,000,000\frac{\text{cycles}}{\text{second}} = comparison\_value\frac{\text{beats}}{\text{interrupt}} \times prescaler\frac{\text{cycles}}{\text{beat}} \times interrupt\_frequency\frac{\text{interrupts}}{\text{second}}$$

or, equivalently:

$$comparison\_value\frac{\text{beats}}{\text{interrupt}} \times prescaler\frac{\text{cycles}}{\text{beat}} = 16,000,000\frac{\text{cycles}}{\text{second}} \times interrupt\_period\frac{\text{seconds}}{\text{interrupt}}$$

where:

[5] https://www.arduino.cc/reference/en/language/functions/digital-io/pinmode/
[6] https://www.arduino.cc/en/Tutorial/Foundations/DigitalPins
[7] https://www.arduino.cc/reference/en/language/functions/digital-io/digitalread/
[8] https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/
[9] https://www.arduino.cc/en/Reference/SPI
[10] https://www.arduino.cc/reference/en/language/functions/advanced-io/shiftout/

16,000,000 Hz is the clock frequency (the inverse of the clock period).

comparison_value is a number you will place in one of the timer's `compare` registers for a comparison-based timer interrupt. This can be any possible value of an unsigned 16-bit integer for Timer1, or any possible value of an unsigned 8-bit integer for Timer2.

prescaler is a multiplier applied to the clock period to adjust the time between counter increments ("beats"). Possible values are 1, 8, 64, 256, and 1024 for Timer1, or 1, 8, 32, 64, 128, 256, and 1024 for Timer2.

interrupt_frequency is how often you want a timer interrupt (the inverse of the interrupt period).

interrupt_period is the time between timer interrupts (the inverse of the interrupt frequency).

You may have to iterate on your design until you arrive at one that works with the constraints of that equation's terms for whichever timer you choose to use.

The Waveform Generation Mode you will use is *CTC* (Clear Timer on Compare) with the "TOP" value set by the `OCRnA` (*compareA*) register, where n is the Timer number. Use Figure 3 (for Timer1) or Figure 5 (for Timer2) to select the appropriate `WGM13` `WGM12` `WGM11` `WGM10` or `WGM22` `WGM21` `WGM20` bits.

Based on the prescaler you chose for the above equation, select the appropriate `CS12` `CS11` `CS10` or `CS22` `CS20` `CS20` bits from Figure 4 (for Timer1) or Figure 6) (for Timer2).

4.2 Setup

In the **`setup()`** function or a function called by **`setup()`**:

You may configure the timer using either memory-mapped I/O or using macros provided by AVR-libc.[11] If using memory-mapped I/O:

- For Timer1, create a pointer to a **`struct`** `timer_registers_16bit` and assign to it the address `IObase + 0x60`.
- For Timer2, create a pointer to a **`struct`** `timer_registers_8bit` and assign to it the address `IObase + 0x90`.
- Use the struct's `control` field to set the `WGM` and `CS` bits in the timer's control registers.
  - Use Tables 1 and 2 (for Timer1) or Tables 3 and 4 (for Timer2) to determine where the `WGM` and `CS` bits are located in the control registers.

If using AVR-libc macros:

[11] AVR-libc provides macros named after the I/O registers that allow you to read from and write to these registers as though they were ordinary variables.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, phase correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, phase correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, phase correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, phase and frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, phase and frequency correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, phase correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, phase correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Figure 3: Waveform Generation Mode Bit Description for Timer1. Copied from ATmega382P Data Sheet, Table 15-5

Bits	**31..24**	**23..16**	**15..8**	**7..0**
control	*reserved*	TCCR1C	TCCR1B	TCCR1A
counter			TCNT1	
capture			ICR1	
compareA			OCR1A	
compareB			OCR1B	

Table 1: Relationship of `timer_registers_16bit` fields to Timer1's registers. Adapted from ATmega382P Data Sheet, §15.11.

- For Timer1, make assignments to `TCCR1A` and/or `TCCR1B`.
- For Timer2, make assignments to `TCCR2A` and/or `TCCR2B`.
- Use Table 2 (for Timer1) Table 4 (for Timer2) to determine where the `WGM` and `CS` bits are located in the control registers.

Next assign your computed *comparison_value* the struct's `compareA` field, or to OCR1A (Timer1) or OCR2A (Timer2), as appropriate.

Finally, enable the comparison-based timer interrupt. If you are using memory-mapped I/O:

- Create a pointer to a **`volatile`** `uint8_t` and assign to it the address `IObase + 0x4E`.

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Figure 4: Clock Select Bit Description for Timer1. Copied from ATmega382P Data Sheet, Table 15-6

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

Figure 5: Waveform Generation Mode Bit Description for Timer2. `WGM22`, `WGM21`, and `WGM20` are incorrectly shown here as `WGM2`, `WGM1`, and `WGM0`. `OCR2A` is incorrectly shown here as `OCRA`. Copied from ATmega382P Data Sheet, Table 17-8

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk_{T2S}/(no prescaling)
0	1	0	clk_{T2S}/8 (from prescaler)
0	1	1	clk_{T2S}/32 (from prescaler)
1	0	0	clk_{T2S}/64 (from prescaler)
1	0	1	clk_{T2S}/128 (from prescaler)
1	1	0	clk_{T2S}/256 (from prescaler)
1	1	1	clk_{T2S}/1024 (from prescaler)

Figure 6: Clock Select Bit Description for Timer2. Copied from ATmega382P Data Sheet, Table 17-9

Bit	7	6	5	4	3	2	1	0
TCCR1C	FOC1A	FOC1B	-	-	-	-	-	-
TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
TCCR1A	COM1A1	COM1A0	COM1BA1	COM1B0	-	-	WGM11	WGM10

Table 2: Timer1's control registers. Adapted from ATmega382P Data Sheet, §15.11.

Bits	15..8	7..0
control	TCCR2B	TCCR2A
counter		TCNT2
compareA		OCR2A
compareB		OCR2B

Table 3: Relationship of `timer_registers_8bit` fields to Timer2's registers. Adapted from ATmega382P Data Sheet, §17.11.

Bit	7	6	5	4	3	2	1	0
TCCR2B	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
TCCR2A	COM2A1	COM2A0	COM2BA1	COM2B0	-	-	WGM21	WGM20

Table 4: Timer2's control registers. Adapted from ATmega382P Data Sheet, §17.11.

- Treat the pointer as an array, and use the timer number (1 or 2) for the index when making an assignment.

If you are using AVR-libc macros:

- Make an assignment to `TIMSK1` for Timer1, or to `TIMSK2` for Timer2.

You want to place a 1 in the `OCIEnA` bit (where n is the timer number); use Figure 7 to determine the appropriate bit to set.

4.3 Interrupt Service Routine

Because the Arduino core does not provide a more convenient way to create an interrupt service routine (ISR) for timer interrupts, you will use AVR-libc's **`ISR`**[12] macro.

In *CalculatorLab.ino*, outside of any other function, write this code that looks like a function:

```
ISR(vector) {
    ...
}
```

where *vector* is `TIMER1_COMPA_vect` for Timer1 or `TIMER2_COMPA_vect` for Timer2. Replace "..." with the code that should execute whenever the timer interrupt occurs. You

[12] https://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIMSK2	–	–	–	–	–	OCIE2B	OCIE2A	TOIE2
TIMSK1	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1
TIMSK0	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0

Figure 7: Timer interrupt registers. Cropped from ATmega382P Data Sheet, §30

want to keep your ISR short, no more than a few lines of code. If anything more elaborate needs to happen, code in your **`loop()`** function (or a function called by **`loop()`**) can do that based on changes made from within your ISR.

Initially, you may want to simply place a **`println`** statement in your ISR code to verify that you have the timing correct. Once you are satisfied that you have done so, remove the **`println`** statement and add the code you need for your logic.

NOTE Any global variables used by your ISR should be declared as **`volatile`**.

NOTE If you ever need to "reset" a timer's count back to 0, you can simply write 0 to the structs `counter` field or to `TCNT1`/`TCNT2`.

5 Detecting Key and Button Presses

For external interrupts, the Arduino core has abstracted-away all of the configuration details.[13] Placing a call to

```
attachInterrupt(digitalPinToInterrupt(pin_number), isr_name, mode);
```

will configure all of the necessary registers to call the function **`isr_name()`** whenever the input value on the pin *pin_number* satisfies the *mode*.

5.1 Setup

In the **`setup()`** function or a function called by **`setup()`**, configure pins `D2` and `D3` for **high-impedance input**.

Decide on the names of the function that will be called in response to a keypress on the matrix keypad and of the function that will be called in response to a button press. For the sake of discussion, I will call these functions, **`handle_keypress()`** and **`handle_buttonpress()`**.

Recall that the NAND output for the matrix keypad columns provides input to `D3` and that the NAND output for the pushbuttons provides input to `D2`. Decide on the *mode* for the external interrupts:

LOW to trigger the interrupt whenever the pin is low

RISING to trigger the interrupt whenever the pin goes from low to high

FALLING to trigger the interrupt whenever the pin goes from high to low

CHANGE to trigger the interrupt whenever the pin rises or falls

Because we did not provide a hardware solution to switch bounce, you can expect both the pin input to both rise and fall a few times when a button or key is pressed and again when it is released – but there is no guarantee that bouncing will occur. For this reason we recommend:

[13] https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

- Use the `CHANGE` mode, combined with software debouncing, and use a variable to keep track of whether a key or button has been pressed (toggle this variable whenever an interrupt occurs, and take action based on whether a key or button has been pressed or released).

- The software debouncing will look similar to what you used in PollingLab except that it only needs to be a few milliseconds instead of 500. Since we are not polling the buttons and keypad to detect presses, we do not need to worry about the button or key being held for several dozen milliseconds being interpretted as multiple presses.

  - I very strongly advise against using **`delay()`** for software debouncing:
  - As described in the PollingLab assignment sheet, **`delay()`** will leave your system unresponsive to anything except interrupts.
  - Including **`delay()`** calls in an interrupt handler is particularly ill-advised since you may find yourself in a situation in which you need to disable interrupts in the interrupt handler and then re-enable interrupts before exiting the interrupt handler. (I do not expect this to happen in this assignment, but I've been wrong before.) The **`delay()`** function will never exit, blocking forever, if interrupts are disabled.

If you arrive at a different solution that works, you may use your solution.

In the **`setup()`** function or a function called by **`setup()`**, register your ISR functions for the external interrupts:

```
attachInterrupt(digitalPinToInterrupt(2), handle_buttonpress, CHANGE);
attachInterrupt(digitalPinToInterrupt(3), handle_keypress, CHANGE);
```

(Here I assumed you would use the CHANGE mode. If you use a different mode, replace CHANGE with the mode you chose. Similarly, replace **`handle_keypress`** and **`handle_buttonpress`** with the function names you chose.)

5.2 Handling Button Presses

Your pushbutton handler ("**`void handle_buttonpress()`**") needs to determine which button was pressed – this can be as simple as calling `digitalRead(8)` and `digitalRead(9)`. Don't forget to include software debouncing. If insufficient time (a few milliseconds) has passed since the last time the button handler was invoked, you can simply exit the handler function under the assumption that switch bounce is generating erroneous interrupts.

You might test out your setup with `println` statements, but once you are satisfied with your setup, delete the `println` statements and introduce the code that will actually handle the keypresses.

You want to keep your interrupt handler short. Any time-consuming computations should happen in your **`loop()`** function (or a function called by **`loop()`**). If necessary, set global variable(s) value(s) that code in other functions can use to perform the time-consuming

computations. (Reminder: division, including modulo, is implemented in software on the Arduino Nano, and requires a few-hundred clock cycles to complete.)

Don't forget to reset the 5-second / 30-second countdown so that your code doesn't blank the display too soon.

Any global variables used by your interrupt handler should be declared as **`volatile`**.

5.3 Handling Key Presses

Your keypad handler ("**`void handle_keypress()`**") needs to determine which key was pressed – you can reuse your **`get_key_pressed()`** code from PollingLab with minimal changes. One such change is replacing the code that makes sure the polling treats a keypress that lasts less than 500ms as a single keypress with code that performs software debouncing and possible keeping track of whether a CHANGE interrupt is the result of a key press or a key release. If insufficient time (a few milliseconds) has passed since the last time the keypad handler was invoked, you can simply exit the handler function under the assumption that switch bounce is generating erroneous interrupts.

You might test out your setup with `println` statements, but once you are satisfied with your setup, delete the `println` statements and introduce the code that will actually handle the keypresses.

You want to keep your interrupt handler short. Any time-consuming computations should happen in your **`loop()`** function (or a function called by **`loop()`**). If necessary, set global variable(s) value(s) that code in other functions can use to perform the time-consuming computations. (Reminder: division, including modulo, is implemented in software on the Arduino Nano, and requires a few-hundred clock cycles to complete.) Almost certainly, you will want to set a global variable to indicate which key was pressed and then let **`loop()`** take action based on which key was pressed.

Don't forget to reset the 5-second / 30-second countdown so that your code doesn't blank the display too soon.

Any global variables used by your interrupt handler should be declared as **`volatile`**.

6 Implementing the Calculator

You now have the ability to handle inputs using interrupts. Writing to the display module can be accomplished with the **`display_data()`** function you wrote in PollingLab. The logic for "building" operands is very similar to some code you wrote for PollingLab's conversion mode.

The basic Integer Calculator specification is reasonably straight-forward, and I trust that, given the ability to get input and to display output, any student who met the course prerequisites can implement it. There are a few nuances that you will need to address, such as blanking the display and resuming its display, but I'm confident that you can do that.

The Fractional Calculator, which is extra-credit, is a little more interesting. You might try using **`float`**s, but you will still need to put a little effort into displaying the values

in accordance with the specification. Alternatively, you might try the fixed point notation "hack" used for decimal currency that I described earlier in the semester. This would allow you to use integer types but would need to manually keep track of the decimal point. Either way, don't forget rounding!

Turn-in and Grading

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

This assignment is worth 50 points.

Rubric:

_____ **+9** Timer interrupts configured such that, when combined with other logic, the software is able to determine when exactly 5 seconds or exactly 30 seconds has passed since the last key press or button press.

_____ **+1** Timer ISR is not excessively long.

_____ **+9** Pushbutton presses are detected with an external interrupt, and the handler, at a minimum, determines which button was pressed.

_____ **+1** Pushbutton interrupt handler is not excessively long.

_____ **+9** Matrix keypad presses are detected with an external interrupt, and the handler, at a minimum, determines which button was pressed.

_____ **+1** Keypad interrupt handler is not excessively long.

_____ **+1** Operand is built in accordance with requirements 6, 7, 8, and 9.

_____ **+1** After eight digits (or a negative sign and seven digits), subsequent numeral presses are ignored (except for resetting the display-blanking countdown).

_____ **+2** Addition performs correctly.

_____ **+2** Subtraction performs correctly.

_____ **+2** Multiplication performs correctly.

_____ **+2** Division performs correctly.

_____ **+1** The result of the previous arithmetic operation can be used as the first operand for the next arithmentic operation.

_____ **+1** Too-great results cause Error to be displayed.

_____ **+1** The "clear" button clears the operand being built and any specified operation.

_____ **+1** If no operand is currently being built (the previous arithmetic's result is displayed) then the "clear" button clears the previous arithemtic's result and any specified operation.

_____ **+2** The display goes blank after exactly 5 seconds or exactly 30 seconds of inactivity, depending on the position of the left switch.

_____ **+2** When the display is blank, pressing any button or key causes the display to resume...

_____ **+2** ...and has no other effect, allowing operation to resume where the user had left off before letting the display timeout.

_____ **Bonus +5** Fractional calculator is implemented.

_____ **Bonus +2** Get assignment checked-off by TA or professor during office hours before it is due.

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 if you complete your assignment early and get it checked-off by a TA or the professor during office hours.

Ideally, all team members are present for the check-off; however, only one team member is necessary for the check-off.

1. ( ) Establish that the code you are demonstrating is the code you submitted to to Canvas.
   - Download the file into your PollingLab directory. If necessary, rename it to *CalculatorLab.ino*.
2. ( ) Upload *CalculatorLab.ino* to your Arduino Nano.
3. ( ) If **Integer Calculator** then the display shows:
   0
   If **Fractional Calculator** then the display shows:
   0.
   Make a note of which calculator was implemented.
4. ( ) Place the left switch in the right position.
5. ( ) Enter 123,456; the display shows:
   123456

6. ( ) **If you implemented display timeout:** Wait exactly five seconds. The display goes blank.

7. ( ) **If you implemented display timeout:** Press 7. The display resumes, showing what it previously showed:
   123456

8. ( ) Press 7. The display shows:
   1234567

9. ( ) Press `A` (addition). The display still shows:
   1234567

10. ( ) Enter 890. The display shows:
    890

11. ( ) Press `#`. The display shows:
    1235457 *or* 1235457.

12. ( ) Press the right pushbutton (clear). The display shows:
    0 *or* 0.

13. ( ) Place the left switch in the left position.

14. ( ) Enter -1,234,567, using the left pushbutton for the negative sign. The display shows:
    -1235457

15. ( ) Press 8. The display still shows:
    -1235457

16. ( ) Press `#`. The display (still) shows:
    -1235457 *or* -1234567.

17. ( ) Enter 231. The display shows:
    231

18. ( ) Press `B` (subtraction) and enter 321. The display shows:
    321

19. ( ) Press `#`. The display shows:
    -90 *or* -90.

20. ( ) Press the right pushbutton (clear). The display shows:
    0 *or* 0.

21. ( ) Press `C` (multiplication) and enter 12. The display shows:
    12

22. ( ) Press `#`. The display shows:
    0 *or* 0.

23. ( ) Enter 42. The display shows:
    42

24. ( ) Press `D` (division) and enter 6. The display shows:
    6

25. ( ) Press `#`. The display shows:
    7 *or* 7.

26. ( ) Press `C` (multiplication) and enter 73. The display shows:
    73

27. ( ) Press `#`. The display shows:
    511 *or* 511.

28. ( ) Press `C` (multiplication) and enter 200. The display shows:
    200

29. ( ) Press the right pushbutton (clear). The display shows:
    511 *or* 511.

30. ( ) Press `C` (multiplication) and enter 200,000. The display shows:
    200000

31. ( ) Press `#`. The display shows:
    Error

32. ( ) **If you implemented display timeout:**

    [ ] Show the code in which you configured Timer1 or Timer2

    [ ] Explain the calculation that you used to arrive at your comparison value

    $$comparison\_value = 16,000,000 \times interrupt\_period \div prescaler$$

    [ ] Show and explain the Timer ISR

    [ ] Show and explain any other code used to implement the display timeou

    [ ] Show and explain the Timer ISR.

    If it is not clear to the TA how the display-timeout code works, then the TA will:

    [ ] Prepare a stopwatch or another timepiece with a "seconds" display (or a "seconds" hand)

    [ ] If necessary, re-activate the display by pressing a button or key

[ ] Press a button or key on the system at the exact same time as starting the stopwatch (or at the exact moment that the "seconds" display/hand hits *00*)

[ ] Note whether the display clears at exactly 30 seconds and not at 30.7 seconds, 29.7 seconds, nor any other that would suggest that **`millis()`** is used in the display timeout implementation

33. ( ) Show and explain the code used to register the external interrupt handlers

34. ( ) Show and explain the pushbutton interrupt handler

35. ( ) Show and explain the keypad interrupt handler

36. ( ) Show the start of *CalculatorLab.ino* to show the TA that the only **`#include`**d header files are *cowpi.h* and possibly the header file(s) from one or more of the Arduino standard libraries.

If you implemented **Integer Calculator**, then this concludes the demonstration of your system's functionality. These steps are written to detect constraint violations during the check-off; however, the TAs will later examine your code for violations of the assignment's constraints.

- If we find that your display timeout uses Timer0 or uses code that uses Timer0 (such as **`millis()`**, **`micros()`**, or **`delay()`**) then you will lose any points that you received for the display timeout functionality.
- If we find that your code polls the pushbuttons to detect button presses then you will lose any points that you received for implementing the pushbutton handler.
- If we find that your code polls the matrix keypad to detect key presses then you will lose any points that you received for implementing the keypad handler.
- If we find that you used a third-party library then you will lose any points that you received for functionality that made use of that library.

If you implemented **Fractional Calculator**:

37. ( ) Enter 3.5, using the `*` key for the decimal point. The display shows:


38. ( ) Press `B` (subtraction) and enter 2.8. The display shows:


39. ( ) Press `#`. The display shows:


40. ( ) Press `A` (addition) and enter 10.32. The display shows:

41. ( ) Press `#`. The display shows:
    1102

42. ( ) Press `C` (multiplication) and enter 23.9. The display shows:
    239

43. ( ) Press `#`. The display shows:
    263378

44. ( ) Press `A` (addition) and enter 100,000. The display shows:
    100000

45. ( ) Press `#`. The display shows:
    10026338

46. ( ) Enter 22.9. The display shows:
    229

47. ( ) Press `D` (division) and enter 5.4. The display shows:
    54

48. ( ) Press `#`. The display shows:
    42407407

49. ( ) Enter 25.5. The display shows:
    255

50. ( ) Press `D` (division) and enter 4. The display shows:
    4

51. ( ) Press `#`. The display shows:
    6375

52. ( ) Press `D` (division) and enter 100. The display shows:
    100

53. ( ) Press `#`. The display shows:
    06375

54. ( ) Press `D` (division) and enter 10,000,000. The display shows:
    10000000

55. ( ) Press `#`. The display shows:
    0

This concludes the demonstration of your system's functionality. These steps are written to detect constraint violations during the check-off; however, the TAs will later examine your code for violations of the assignment's constraints and deduct any unearned points accordingly.