Computer Science 230
Computer Architecture and Assembly Language
Fall 2023

Assignment 3

Due: Sunday, November 19th, 11:55 pm by Brightspace submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on Arduino boards in ECS 249. If you have installed Microchip Studio on your own computer, then you are welcome to do some of the programming work on your machine. However, please plan to spend a significant amount of time in the lab.

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Baharloo).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.

Objectives of this assignment

- Use multiple AVR 16-bit timers.
- Write interrupt handlers for two timers.
- Output a representation of program state onto the Arduino mega2560 board's 2x16 LCD display.
- Practice with writing code the implements (and using functions that depend upon) the parameter-passing mechanism based on stack frames.

Interrupts, LCD panel

In this assignment you will use two features of the Arduino boards that have been introduced recently in lab: interrupts and the use of the 2x16 LCD display. The handlers required for the assignment will not be particularly complex, but they must work precisely. (Their execution will be triggered by several of the board's timers.) The LCD panel will finally allow you to create board behavior that is richer than simply turning LEDs on and off; the labs earlier this semester introduced you to the LCD panel and to interrupts.

A big challenge of this assignment, however, is that you are now moving into an area of programming where our debugger is of very little help. When handlers are not correctly coded or configured, the result is a mute board. Hence it is crucial you not only complete this assignment in the order of the four parts listed, but also build upon each of the completed parts by coding with successive projects folders. The idea here is that even if you struggled with part D, your success with parts A, B and C will be in those completed project folders for those parts (i.e., the code for those earlier parts can still run successfully).

A brief demonstration illustrating the behavior for each of the parts can be seen in this video:

`https://youtu.be/EpKo95vsFmU`

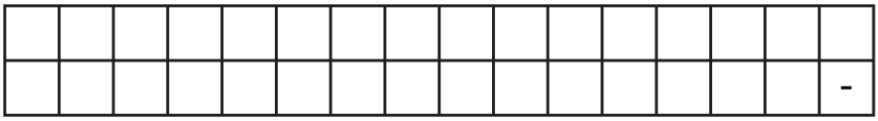
As stated earlier, this assignment is in four parts, ordered from easier to more difficult, with each later part building upon the work of the previous parts.

A. Write code to show on the LCD panel whether or not a button is currently being pressed.
B. Write code to show on the LCD panel which of four buttons (left, up, down, right) is either being pressed, or if there is no button being pressed, which button was last pressed.
C. Write code to permit the use of the "up" and "down" buttons to set a hexadecimal digit at the top-left of the LCD display.
D. Write code to extend your work in Part C such that using the "right" and "left" buttons will permit the "up"/"down" buttons to work in a different part of the display's top row.

The ZIP file distributed with this assignment contains four directories, each consisting of a Microchip Studio project. Directory `a3part-A/` is meant for part A, `a3part-B/` is for part B, etc. In directory is an A#3 starter file in which you are to write your solution for that particular part of the assignment. (Note every directory starts out with exactly the same file contents.) The idea is that as your work progresses from part to part, you can copy working code from one Microchip Studio project to the next. In that way, if a later part is not finished or does not work, it will not interfere with your working solution to an earlier part.

Part A: Button press

After completing this part, your Arduino board's LCD display will appear have the following display when no button is being pressed:



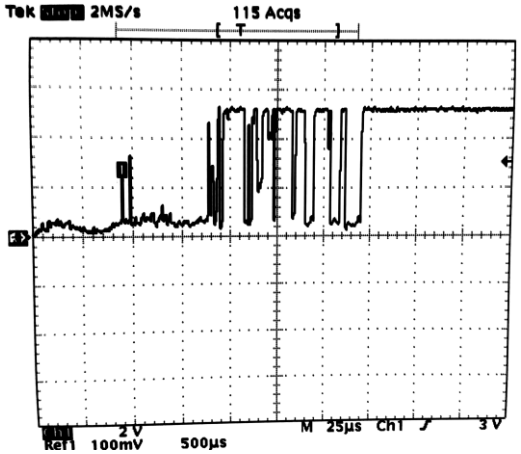
and the following display when any button is being pressed:

and this behavior continues as long as the Arduino runs your program.

This semester in lab 4 you examined the way an Analog-to-Digital Converter (ADC) is used to read buttons on the Arduino board. The ADC obtains a value from 0 to 1023 from the buttons – if the value is greater than 900, then no button on the LCD shield is being pushed. Code from lab 4 helped you explore how you might write code to detect which button is pressed; polling loops were an important part of your solutions.

Interrupts could help us here to detect ADC events, but not necessarily in the way we expect. It would be tempting to have a solution where a button press itself results causes an interrupt. However, in practice this is rarely done because of an electrical phenomenon known *bouncing*.

Consider the figure below (found at `http://bit.ly/2IdyYRg`):



This is a screengrab from an *oscilloscope* (if you've never heard of these, visit `http://bit.ly/1RXdkgw` for a description) showing the electrical behavior of a

button being pressed. The signal does not clearly go from voltage low to voltage high, but goes up and down quickly before settling to a high voltage; we say that the voltage *bounces* (i.e., like a rubber ball being bounced by a schoolchild on a playground). The diagram shows that in the space of two milliseconds, a button being pressed actually appears to the computer as if the button were rapidly pressed a dozen or so times. Therefore, if we used an interrupt to detect when a button caused the voltage to go high, then we would have too many interrupts for a single button press. So, in practice designers implement *debouncing,* usually in hardware. Our Arduino boards already perform some debouncing on the buttons, but still not enough for our purposes.

What we will use instead is a timer to sample the value of the ADC at specific intervals (in our case, `timer1` set at 10 milliseconds). Every time the interrupt handler for `timer1` is executed, its code must determine whether or not a button is currently being pressed. If this is the case, then a value of 1 is stored in `BUTTON_IS_PRESSED` (i.e., see the `.dseg` at the end of the assembler file); otherwise a value of 0 is stored in this location.

So far this description indicates the need for one timer (i.e., one with a duration of 0.01 seconds). But how then is the LCD panel updated to reflect the button state (i.e. either a "-" or "*" character at the lower-left corner)? Unfortunately, the interrupt handler for `timer1` itself **must never call the LCD routines**! In fact, no interrupt handler is able to call LCD code. So, if the handler is not permitted[1] to call `lcd_gotoxy` and `lcd_putchar`, how do we update the display?

The answer: Use **another** timer in main program but make use of it via a polling technique. This additional timer (`timer3` in your program) goes off every 100 milliseconds. When the polling loop detects `timer3` has reached its `TOP` value (i.e., 100 ms has occurred), then the code for updating the LCD display can be executed. The latter code will determine whether "-" or "*" or should appear in the area of the LCD to indicate the button-press state. *It is perfectly legitimate to re-write repeatedly the same characters in the same locations on the LCD!*

Reminder: `timer3` does not use an interrupt handler!

[1] My use of the phrase "is not permitted" is a bit strong. In fact, the assembler will not forbid us from calling the LCD routines from an interrupt handler. However, once the program calls such routines from a handler, the whole program will simply stop working. In general, it is a *Very Bad Idea* for an interrupt handler to call other routines, especially because we cannot always be guaranteed those routines themselves do not depend upon interrupts. The one exception here is the code I've provided to you in a function named `compare_words`.

Part B: "What button was pressed?"

(This part will be based on your solution to part A. That is, the code within your `a3part-B.asm` will use code you have copied-and-pasted from `a3part-A.asm`).

In lab 4 you may have done a little bit more with buttons – in fact, you may have even had an opportunity to write code to determine precisely which button is pressed. The ADC obtains a value from 0 to 1023 from that represent button states – if the value is greater than 900, then definitely no button is being pushed. (All of the comments earlier in part A regarding "noise" and "bouncing" are especially important here.) The ranges for all buttons on our LCD/button shields are maddingly inexact and may require a bit of tweaking and tuning, but they can be described roughly as follows:

- from 0 to around 50: "right" button[2]
- from around 50 to around 176: "up" button
- from around 176 to around 352: "down" button
- from around 352 to around 555: "left" button
- from around 555 to around 800: "select" button
- above 900: *no button is pressed*

For this assignment we will use "left", "down", "up", and "right" buttons (i.e. we'll ignore "select"). The LCD will display along the bottom row either the current button being pressed, or if no button is being pressed, the last button to be pressed. Button-letter-to-LCD mappings are to be as shown below:

Although the diagram above shows all button letters, in practice only one letter will be shown at any one time. (At the start of the program, no letter needs appear as no button will have yet be pressed; this will be the only situation where no letter appears in the lower-left area of the display.)

Therefore, in order to complete this part of the assignment you will need to modify both the handler for `timer1` and also the LCD-update loop that is in your code driven by `timer3`. The button pressed must be stored in `LAST_BUTTON_PRESSED` (in the `.dseg` area at the bottom of the assembly file); that is, `timer1` will "communicate" with the `timer3` loop via sharing this region of memory.

And do not be too disturbed by the "bouncing" you see when you write your code. Sometimes a button press will result in some other letter showing up on the display. This can be avoided either by shorter presses or longer presses. You may be able to reduce the occurrence of this by tweaking the range values given above for buttons

[2] There is a typo on many of the boards in the lab that spells this as "RIGTH".

and the ACD reading. Regardless, if such "bouncing" is the exception and not the norm, then you are on the right track with your work. Perfection is hard to achieve here and is not expected.

Part C: Setting a hexadecimal digit on the LCD top row

(This part will be based on code you have written for parts A and B. That is, the code within your `a3part3-C.asm` will use code you wrote for `a3part2-B.asm`).

Now that you can determine which of the directional letters has been pressed, you will use this for setting a hexadecimal digit that appears at the upper-left hand corner of the LCD display:

To do this, the "up" and "down" buttons will be used – "up" to move from a smaller hex digit to a larger, and "down" to move from larger hex digit to a smaller. At the bottom of the assembly file you will see:

```
AVAILABLE_CHARSET: .db "0123456789abcdef_", 0
```

which you must use to move "up" (i.e. from left-to-right in the sequence) and "down" (i.e. from right-to-left in the sequence"). Notice too that there is a special character at the end that is not a digit ("_") and that the sequence is actually a string as it is null terminated. **Your code must not hard-code the length of the sequence string** – that is, it must be possible to place a difference sequence in the assembly code (i.e. some other sequence of letters / numbers / characters), such that this different set of characters can be selected when the program is re-assembled and re-run.

The rate at which the letter can be changed needs to be controlled, however, and that is accomplished by using `timer4` which raises an interrupt twice a second (i.e. every 500 milliseconds). That is, this handler will use other state maintained by the other handlers / loops to update these three memory areas in `.dseg`:

- `TOP_LINE_CONTENT` which is 16 bytes in length and is to be read by the `timer3` loop (i.e. calls to `lcd_putchar` for the top row, amongst other calls). This must be initialized with space characters.
- `CURRENT_CHARSET_INDEX` which is 16 bytes in length and where its first location will hold the index/offset into `AVAILABLE_CHARSET` that resulted from the last up/down button press. For example, if the digit 'd' appears in the top-left corner of the display, then the byte at `CURRENT_CHARSET_INDEX` will hold the value 13 as a result of your implementation.

- `CURRENT_CHAR_INDEX` can be ignored for this Part C, but its implied value here is zero (0).

Remember that the `timer3` loop is the only code permitted to call LCD routines.

Part D: Setting other hexadecimal digits on the LCD top row

(This part will be based on the code you have written for the previous parts. That is, the code within your `a3part-D.asm` will use code you wrote for `a3part-C.asm`).

The last part of this assignment is to add behavior to the "right" and "left" buttons so that other hex digits on the LCD's top row may be set.

c	a	f	e	8	b	0									
		U													-

This will involve adding extra functionality to the `timer4` handler, and hopefully will require only minimal modifications to your `timer3` polling loop. The memory regions introduced in Part C must be used for these purposes (i.e. there are 16 chars on the top row of the LCD, there are 16 byte locations in both `TOP_LINE_CONTENT` and there are 16 byte locations `CURRENT_CHARSET_INDEX`).

You may find that pressing the "right" and "left" buttons result in top-row spots being missed. In the same spirit as what was mentioned above regarding the effects of "bouncing" for Part B, as long as this behavior is the exception and not the norm, you're on the right track with your programming.

Summing up …

By the time you are completed work for all parts of this assignment, you will have – in essence[3] – accomplished the following:

- For `timer1`: The **handler** will examine the ADC for button-presses **every 10 milliseconds**, writing correct values into the `BUTTON_IS_PRESSED` and `LAST_BUTTON_PRESSED` memory areas.
- For `timer4`: The **handler** will examine the values in `BUTTON_IS_PRESSED` and `LAST_BUTTON_PRESSED` **every 0.5 seconds** in order read and write memory areas `CURRENT_CHARSET_INDEX`, `CURRENT_CHAR_INDEX`, and possibly `TOP_LINE_CONTENT`.
- For `timer3`: The **polling loop** will examine the values in the five memory areas (listed in the two bullet points above) every **100 milliseconds**. The

[3] 😟

code in this loop will make the appropriate calls to `lcd_gotoxy` and `lcd_putchar` to ensure the LCD display represents that state of the program. You may simply "redraw" the entire LCD display every 100 milliseconds; and that frequency, there is very little (if any) flicker.

What you must submit

- Your four completed parts: `a3part-A.asm`, `a3part-B.asm`, `a3part-C.asm` and `a3part-D.asm`. **Do not change the name of these files!** Do not submit the LCD files. **Submit only the `.asm` files listed above!**
- Your work must use the provided skeleton files. Any other kinds of solutions will not be accepted.

Evaluation

- 6 marks: Solution for part A
- 5 marks: solution for part B
- 5 marks: solution for part C
- 4 marks: solution for part D

Therefore, the total mark for this assignment is 20.

Some of the evaluations above will also take into account whether or not submitted code is properly formatted (i.e., indenting and commenting are suitably used), and the file correctly named.

In contrast to previous assignments, Assignment #3 will be assessed through a live code demonstration. In this process, each student will have the opportunity to meet with a member of the teaching team. These instructors will review the code submitted by the student and engage in a discussion about their work during the lab session scheduled for the week following the Assignment #3 deadline.