Computer Science 230
Computer Architecture and Assembly Language
Summer 2022

Assignment 3

Due: Monday, July 18th, 11:55 pm by Brightspace submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the simulator within Microchip Studio 7 as installed on workstations in ECS 249. If you have installed Microchip Studio on your own then you are welcome to do much of the programming work on your machine. (The IDE is available at no charge from `https://bit.ly/3llENk7` but comes only in the Microsoft Windows flavour.) **You must allow enough time** to ensure your solutions work on the lab machines. If your submission fails to work on a lab machine, the fault is very rarely that the lab workstations. "It worked on my machine!" will be treated as the equivalent of "The dog ate my homework!"

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 (Zastre).** 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 program state onto the Arduino mega2560 board's LCD display.
- Practice more 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 may be new to you. Interrupts and interrupt handling will be introduced and discussed in lectures and labs. The handlers required for the assignment will not be long or 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 during the week of March 12th will introduce you to the LCD panel.

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 4, your success with parts 1, 2 and 3 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/ydZmuO5DNWg`

The assignment is in four parts, ordered from easier to more difficult, each part building upon the work of previous parts.

1) Write code to display a "heartbeat" on the LCD panel.
2) Write code for detecting if any button is pressed, maintaining a count of such presses, and outputting that count on the LCD panel as a button is pressed.
3) Write the code for detecting that a button is held down, and showing this by outputting a visible message on the LCD panel during that button press.
4) Write code for detecting if a button press is short or long (i.e., equivalent to a Morse code dot or dash) and displaying the dots and dashes on the LCD panel

(Note: For parts 2, 3, and 4 any button on the shield can be pressed **except** for the "RST" button.)

The ZIP file distributed with this assignment contains four directories, each consisting of an Microchip Studio project. Directory `a3part1/` is meant for part 1, `a3part2/` is for part 2, etc. In each directory are copies of code needed for using the LCD display. Also in each 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 you progress 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 1: Heartbeat

After completing this part, your Arduino board's LCD display will appear to turn the symbols "<>" on and off. For half a second the board will look like this:

and for the next half second the board will look like this:

and this behavior will repeat itself as long as the Arduino runs your program. (The "<" character is written in the cell at row 0 and column 14, while ">" is written in the cell at row 0 and column 15.)

This description clearly indicates the need for one timer (i.e., one with a duration of 0.5 seconds), and for this we will use `timer1`. When the timer's interrupt handler is executed, it must somehow toggle the state of the heartbeat (should the "<>" be showing? should it not?). This state can be kept in the `PULSE` data memory location (i.e., this is defined towards the end of the `a3part1.asm` file provided to you) and it is up to you how to represent the state.

However, the interrupt handler for `timer1` itself **must never call the LCD routines**! In fact, interrupt handlers must have a minimal amount of 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 use it via the polling technique. This additional timer (`timer3` in your program) runs for 50 milliseconds. When the polling loop detects `timer3` has reached its TOP value (i.e., 50 ms has occurred), then the code for updating the LCD display can be executed. The latter code will determine whether "<" or ">" or a space should appear in the area of the LCD display for the heartbeat based on its examination of the value stored in `PULSE`. Reminder: `timer3` does not use an interrupt handler!

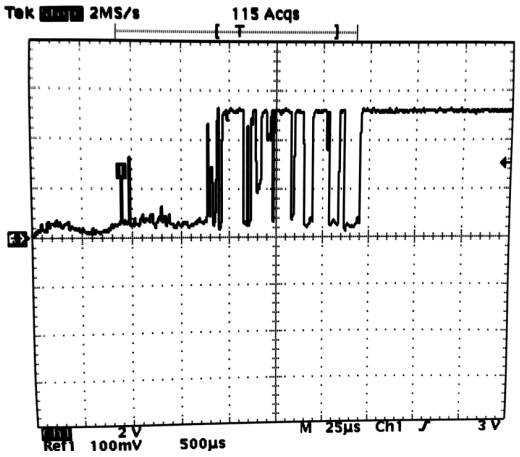
[1] My use of the phrase "is not permitted" is a bit strong. In fact, the assembler will not forbid us from calling the LCD routines from an interrupt handler. However, once the program calls such routines from a handler, the whole program will simply stop working. In general it is a *Very Bad Idea*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

Part 2: Counting button presses

(This part will be based on your solution to the part 1. That is, the code within your `a3part2.asm` will use code you wrote for `a3part1.asm`). 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 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 can help us here but not necessarily in the way we expect. It would be tempting to have a solution where a button press itself results in an interrupt. However, in practice this very 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). If we just used an interrupt to detect when a button caused the voltage to go high, then we would have 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.

What we will use is a timer to sample the value of the ADC at specific intervals (in our case, `timer4` is set at 10 milliseconds, and this will be our interval). Every time the interrupt handler for `timer4` is executed, its code must store the previous button-press value in `BUTTON_PREVIOUS` (i.e., see the `.dseg` at the end of the assembler file), and then determine the current button value (pressed? or not pressed?) and store that into `BUTTON_CURRENT`. And before the handler is finished, the it will examine `BUTTON_PREVIOUS` and `BUTTON_CURRENT` to determine if the button press has just started; if it so, `BUTTON_COUNT` will be incremented by one.

And what about our LDC display? When no button has yet been pressed, the LCD display will look something like that below (note the position on the display at which the count **must** appear):

(The heartbeat is grayed-out in the diagram indicate that it isn't relevant to this particular discussion, but it will continue to be active when this part 2 code is completed.) When a button has been pressed, the display will look like:

The same loop used to update the heartbeat should be extended to update the count on the display (i.e., the loop which updates the LCD display every 50ms can itself be modified). One problem you will need to solve, however, is how to convert the 16-bit `BUTTON_COUNT` into its decimal representation as characters. I have suggested an approach in a file named `hex-to-decimal.pdf` that you will find with the provided ZIP file for this assignment. If you decide to use this approach in your solution, then please make sure to cite this in a comment. You can use the `DISPLAY_TEXT` location in data memory as a buffer in which to store – and from which to load – text characters.

Part 3: Displaying a button press

(This part will be based on code you have written for parts 1 and 2. That is, the code within your `a3part3.asm` will use code you wrote for `a3part2.asm`).

In the previous part of the assignment you wrote a handler that reads an ADC value (for the buttons) every 10 milliseconds. As part of its work, that handler assigns suitable values to `BUTTON_PREVIOUS` and `BUTTON_CURRENT`.

In this third part of the assignment, you **do not** need to write a new interrupt handler or modify an existing handler. Rather, you will extend the LCD update loop that already exists in your code such that:

- when `BUTTON_CURRENT` indicates a button is currently pressed, the lower-left portion of the display will indicate this by six asterisks
- when `BUTTON_CURRENT` indicates no button is pressed, the lower-right portion of the display will indicate this by keeping the lower-left portion of the display blank.

So if a button is pressed, the display will look something like:

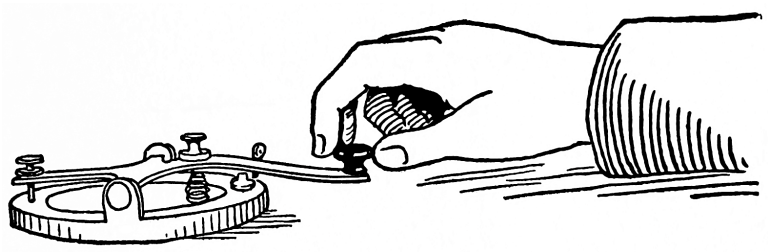
														<	>
*	*	*	*	*	*						0	0	0	3	9

and when no button is pressed, the asterisks will have disappeared. Throughout all this the heartbeat must continue to operate, and the button-press count must be updated. (In the figure above, the heartbeat and button count are greyed out to indicate their values are not important in this part of the assignment discussion.)

Part 4: Detecting dots and dashes

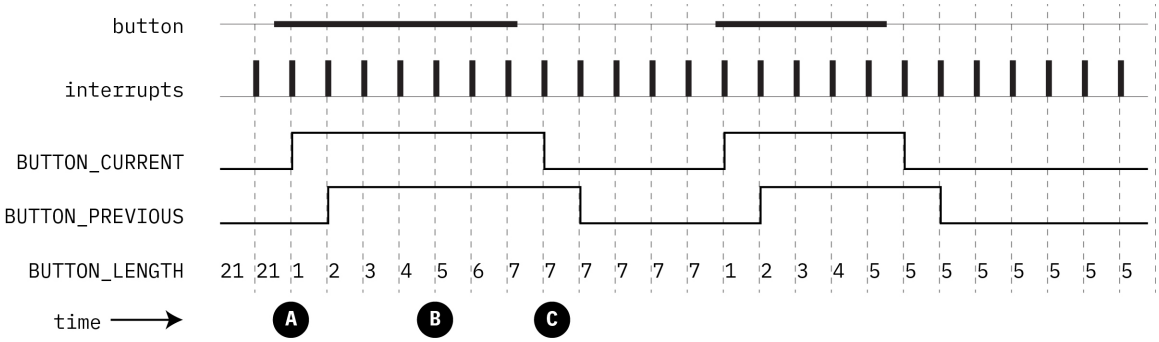
(This part will be based on the code you have written for the previous parts. That is, the code within your `a3part4.asm` will use code you wrote for `a3part3.asm`).

In assignment #2 you learned about Morse code and the way letters can be encoded in the form of short signals and long signals (i.e., dots and dashes). In the 19th century these dashes were electrical signals generated by the sender and heard as sounds by the receiver. The sender used a telegraph key like that shown below (from `https://bit.ly/3OY1CY6`):



A dot was literally a short tap on the key, while a dash was a longer press on the key. For our assignment, a dot corresponds to a short button press, while a dash is a longer button press. In practice "short" means "less than or equal to 200 milliseconds" while "longer" is "more than 200 milliseconds". You can use work completed in part 2 to help detect the length of a button press.

Consider the figure below:



The top-most row represents the state of a physical button. (Any button may be pressed except for the RST button.) Where the line is thin, no button is pressed; where the line is thick, one of the buttons is pressed. The 10-millisecond timer interrupts are shown on the second row, where a thick vertical line indicates the running of the interrupt handler. The third and fourth rows (for `BUTTON_CURRENT` and `BUTTON_PREVIOUS`) represent values stored in these memory locations; these could be 0 and 1 (i.e., 0 is the line when it is low, 1 is the line when it is high). The last row represents the number of interrupts for which a button has been pressed (`BUTTON_LENGTH`). Three points in time are also shown:

- At point A, the pushed button has just been recognized (i.e., in the previous interrupt, the button was not pushed). Therefore we can reset `BUTTON_LENGTH` to 1.
- At point B, the button is still being pressed, and `BUTTON_LENGTH` continues to increase.
- At point C, a button release has just been recognized (i.e, in the previous interrupt, a button was indeed pushed down, but now no button is being pushed). At this point we can not only stop incrementing `BUTTON_LENGTH`, but we can also determine whether or not the button push was long enough to have been a dot or a dash. Since this button press is 70 milliseconds long, it is short enough to be a dot.

In part 4 we will therefore want to modify and extend the handler we first wrote in part 2. This handler is the best place to not only maintain `BUTTON_LENGTH` but also decide on dots and dashes and record the result as characters (i.e., '.' or a '-') in the `BUTTON_PATTERN` buffer.

The LCD update loop can now output the `BUTTON_PATTERN` on the upper-left portion of the display, where a "dash dot dash" looks like:

-	.	-												<	>
*	*	*	*	*	*						0	0	0	0	3

Up to ten dot/dashes should be shown on the LCD display. If we exceed this number of button presses, then no more dots/dashes will be added, but the rest of the program should continue to work, i.e., button presses will still increase the count; button presses will still be displayed as asterisks in the lower-left of the display; the heartbeat will continue.

What you must submit

- Your four completed parts: `a3part1.asm`, `a3part2.asm`, `a3part3.asm` and `a3part4.asm`. **Do not change the name of these files!** Do not submit the LCD files. **Submit only the `.asm` files!**

- Your work must use the the provided skeleton files. Any other kinds of solutions will not be accepted.

Evaluation

- 4 marks: Solution for part 1
- 8 marks: solution for part 2
- 3 marks: solution for part 3
- 5 marks: solution for part 4

Therefore the total mark for this assignment is 20.

Some of the evaluation 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.

Unlike other assignments, A#3 will be evaluated via a code demo. That is, each student will meet with a member of the teaching team, who will have access to the student's submitted code and will ask that student questions about their code. Instructions on how to sign up for a demo will be given sometime during the week of July 11th.