Question summary

1. Where do I start?
2. That seems clear...but don't I have to write some code?
3. What exactly is a *stub*?
4. OK, all my high level stubs are in place. What next?
5. OK, interrupts work. Now what?
6. Do I need to connect the *eebot* to test this part of the lab?
7. Any suggestions of how to approach the second part (bumper switches)?
8. What's with all the wierd @ signs in the comments?
9. Do I need the *eebot* for this part?

Question and answers

---

1. Where do I start?

At the beginning!

Seriously, read the lab and determine the requirements. As I see it, there are three requirements:

1. Demonstrate that Timer Alarms work.
2. Demonstrate that Timer Alarms can do something.
3. Demonstrate that Bumper Switches can be read and displayed.

Reading the requirements more carefully, you may come to the conclusion that only the last two requirements matter. I suggest that your aim be should be to meet these two checkpoints.

---

2. That seems clear...but don't I have to write some code?

Indeed! OK, now you have to come up with a strategy for developing and testing your program(s). And you should do *all* of your work within a single file named `lab4.asm`.

Of course, you will start working on one requirement before the other; and you may even work on both before either is finished. *Avoid putting each part in a separate file.* Instead, use something like the following boilerplate:

```
; Lab 4 (ele538) "Timed Alarms, Bumper Detection"
; BY: !!!!!PUT YOUR OWN NAME herE!!!!
;*******************************
;*       EQUATES
;*******************************
```

```
;i.e. stuff like...
LCD_CONTROL      equ $1400            ;Lower 7 bits are control


;*********************************
;*       PROGRAM SEGMENT (may be in ROM)
;*********************************
        org $6000
        jsr doTimerAlarms
        swi  ;exit to Buffalo monitor; PC will be 6004

        org $6010
        jsr doRealBumpers
        swi  ;exit to Buffalo monitor; PC will be 6014

doTimerAlarms:
        rts
doRealBumpers:
        rts
```

This boilerplate code is designed around the central idea that there are two different things done by the demonstration program and each one has a separate entry point. Here, if the program is downloaded, the commands `GO 6000` and `GO 6010` will exercise either of the two requirements. Note that although the code does *nothing*, it does compile with no errors and if you download and run from either entry point, control is returned (quickly!) to the Buffalo monitor with the specified PC values.

3. What exactly is a *stub*?

A *stub subroutine* is not fully functional. But stub subroutines are very useful during development since they allow you to use a well-designed software architecture at the outset.

Stub functions often do nothing; they are simply place holders for future real code. However, they can be used! Suppose a function is supposed to calculate `(AccA+3)*2` and has been implemented as the following "do-nothing" stub:

```
;fooCalc modifies AccA --- returned AccA is (AccA+3)*2
fooCalc:
        rts
```

Suppose you need this function to work during testing, but have not yet implemented it. No problem! Just set a breakpoint at the subroutine address. When the breakpoint is hit, look at AccA, figure out what the new value should be, use the `RM` monitor command to set AccA to the proper value and continue execution.

Suppose, instead, you did not want to use breakpoints, but always called `fooCalc(4)` during the initial testing. In this case, you could use the following stub (which always returns `(4+3)*2 = 14`.)

```
;fooCalc modifies AccA --- returned AccA is (AccA+3)*2
fooCalc:
        ldaa #14
        rts
```

4. OK, all my high level stubs are in place. What next?

I suggest you start with the Timer Alarm program. One way to get started is to make the `doTimerAlarms` subroutine do a bit more; i.e.

```
doTimerAlarms:
        jsr initTOFIntrpt
        cli               ;Turn on CPU recognition of interrupts
        ;check if TOFcounter has increased at 15 Hz rate
forever bra forever  ;enter INFINITE loop (see remarks below)
        rts
```

Run this program. It will call the routine to start the TOF interrupt and then loop endlessly in the `forever bra forever` code.

Manually reset the HC11. Examine `TOFCounter`. It should have increased by 15 counts for every second the program was running. (It helps, of course, if you know the initial value of the `TOFcounter` although everything should work no matter what the initial value is.)

Run the program again and reset manually. The count should have increased further.

Remarks

1. Note that 15 is almost equal to 16. If it were exactly 16, the upper 4 bits of `TOFcounter` would be incremented by one every second (exactly). Thus if `TOFcounter` is initially zero (0x00), it will be 0x4- after about 4 seconds.
2. You can set a known initial value either with an `fcb` directive or explicitly in your initialization routine; in the former case, the initial value is known only immediately after downloading, in the latter case, it is reset each time you run the program.

5. OK, interrupts work. Now what?

The basic steps in fulfilling the first requirement are quite straightforward once you are convinced that interrupts are working. However, you need to display things on the LCD and, although you have done this before (I hope!), you have to be careful when incorporating a previously working subroutine (say from Lab 2) into a new lab.

When code sections are cut and pasted from one lab to another, you have to beware of the following pitfalls:

1. Ensure there are no duplicate labels. (Hence avoid generic labels such as `done`, `endif`, `loop`, etc.)
2. Ensure that symbolic equates (i.e. `EQU` statements) needed by the code you are cutting and pasting are also copied.
3. Similarly, ensure that global variables(i.e. `FCB`, `RMB`, etc. statements) needed by the code are also copied. Do the same for subroutine names.

6. Do I need to connect the *eebot* to test this part of the lab?

No.

7. Any suggestions of how to approach the second part (bumper switches

One possibility is to follow the same approach I used, but you are certainly free to use any approach you want. (My approach may not be the best; it does have one advantage: I *know* it works!)

In my approach, I choose not to worry about the details of how to read the bumper switches. Instead, I concentrated on the main loop of displaying their status on the LCD. I did, however, design the interface to the subroutine that would read the switches and used the following interface and initial stub implementation:

```
;; @name getBumpers
;; Determines state of bumper switches.
;; @param none
;; @return AccA Bow bumper if == 0 ==> OFF; != 0 ==> ON
;; @return AccB Stern bumper if == 0 ==> OFF; != 0 ==> ON
;; @side CC modified
;; @example
;;
;;     jsr getBumpers
;;     tsta
;;     bne frontHit
;;
getBumpers:
     ;stub implementation....set breakpoint here
        rts
```

Some of you may be astounded that I wrote so many comments for a subroutine that *does nothing*. I find, however, that doing so allows me to focus on how I want the subroutine to behave before I actually write it. Once it is written, I need only look at the comments to remember how to use and do not have to think about messy details like device addresses, bit positions and so forth.

I then wrote the main loop. It was not clear whether this loop should last forever, or only for some length of time, or until some condition was met (like both bumpers active). Eventually, I realized that I could use the Timer Alarm feature to run the loop for a specific duration (say 10 seconds). In this way, I could demonstrate both the bumper switch program and the Time Alarm implemenation in a single integrated program. (You may wish to do the same thing, but it is not required.)

8. What's with all the wierd @ signs in the comments

These are used by the `asmdoc` tool to convert structured comments into cross-indexed, formatted web pages. See the web pages on assembly language coding conventions for details.
(The `asmdoc` command is available in the directory "`~courses/ele538/bin`".)

Briefly, special tags like `@param` are used to formally specify things like *parameters* within public comments (comments that begin with two colons.)

9. Do I need the *eebot* for this part?

Yes...you need it to fully test and demonstrate this part. However, you can do most of the development and testing without a real robot connected to the system.

I had two different routines that implemented
the `getBumpers` interface: `getRealBumpers` (which read the real bumper switches on
the *eebot* and `getVirtBumpers`, which used the potentiometer on the base station to simulate Bumper Switches at run time.

Obviously, I am not going to give you the code for `getRealBumpers`, but you are welcome to use or modify the following code for `getVirtBumpers`:

```
;; @name getVirtBumpers
;; Determines state of "virtual bumpers".  On the base station, the
;; potentiometer can simulate the front and rear bumpers. When the pot
;; is turned fully counterclockwise (minimum), both bumpers are off.
;; When it is turned fully clockwise (maximum), both bumpers are on.
;; At about 1/4 turn, the Stern bumper comes on; at about 1/2 turn, Stern
;; is off and Bow is on.
;; @param none
;; @return AccA Bow bumper if == 0 ==> OFF; != 0 ==> ON
;; @return AccB Stern bumper if == 0 ==> OFF; != 0 ==> ON
;; @side CC modified
;; @example
;;
;;     jsr getVirtBumpers
;;     tsta
;;     bne frontHit
```

```
;;
getVirtBumpers:
        ldx #rawADbuf
        jsr getAllADChannels
        ldaa #$80
        anda 0,X
        ldab #$40
        andb 0,X
        rts
```

Actually, you cannot use the code "as is" unless you implemented the subroutine for reading the AD converter using the same interface and names that I did. Since this is highly unlikely, you would have to figure out what the code does and modify it for your interfaces...