UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

ECE-013: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 7 - Toaster Oven
23 Points

Warning

This lab is much more involved than the previous ones, and will take you more time. You will need to start early and read this lab carefully. Failure to plan is planning to fail.

Introduction

This lab introduces finite state machines as a tool for programming reactive systems. FSMs are commonly used, though they are rarely explicitly described with those words. They are a very powerful technique for organizing the behavior of complex systems[1] (and just as importantly debugging them). FSMs are used, for example, in your microwave, dish washer, thermostat, car, and many others. Mastering software state machines is one of the fundamental skills you will learn in this class—and a skill that will be quite valuable to you in your programming careers.

For this lab you will implement a toaster oven by following the state machine diagram we have provided. This lab builds on the event-driven programming used in the Bounce lab. A proper state machine uses events to trigger state transitions with an idealized instantaneous transition. This is a key idea in embedded systems.

You will need to able to understand and reason about state machines to do well in this lab. Make sure you do the reading before attempting the lab. Use the example bounce FSM provided here to rewrite your Bounce code using a state machine implementation as a test of your understanding. Ask a TA/tutor for help in working through this and you'll understand much better how to implement this lab.

Concepts

- `const` variables
- Timer interrupts

[1] You can even come up with a state machine description of the ground state of a hydrogen atom in quantum physics.

- Free Running Counters
- Event-driven programming
- Finite state machines

Reading

- This lab manual (you need to read this carefully, probably several times)
- **CKO** – Chapters 5.8– 5.11
- Wikipedia on FSMs: https://en.wikipedia.org/wiki/Finite-state_machine

Required Files:

- Lab07_main.c
- Leds.h
- README.md

Lab Files:

- **DO NOT EDIT these files:**
  - BOARD.c/.h – Standard hardware library for CSE13E.
  - Oled.h, Ascii.h – These files provides the interface for manipulating the oled Oled. No .c files are needed here, as Lab7SupportLib contains compiled definitions for the functions in Oled.h.
  - Buttons.h , ADC.h – Same as the files from the previous lab. However, the definitions in these functions are now implemented in Lab7SupportLib.a.
  - Lab7SupportLib.a – This is a “static library” file. It’s similar to a *.o file. It contains the executable code for the functions in Buttons.h and ADC.h. This allows you to use the ADC and Buttons libraries, without having direct access to our code.
    - To use Lab7SupportLib.a, right-clicking on the “Libraries” folder in your MPLAB X project panel and selecting "Add Library/Object File...".
    - Do not create a Buttons.c or ADC.c file!
- **Edit** these files (these are provided as *_template.c/h files, you must rename them):

- Leds.h – a simple macro file. It serves the same purpose as the Leds_Lab06 library from the previous lab, but implements all "functions"using macros.
  - Do not create a Leds.c file!
- Lab07_main.c – This file will contain all of the executable code that you submit for this lab.

Assignment requirements:

In order to master finite state machines, you are going to implement a toaster over on the microcontroller. In this case, you are going to display the state of the toaster oven graphically on the OLED rather than have you interact with actual heating elements.[2] This lab requires you to implement all the required toaster oven functionality in Lab07_main.c utilizing the provided libraries.

Toaster Oven Functionality:

- The system displays (on the OLED) the heating elements state in a little graphical toaster oven, the cooking mode, the current time (set time or remaining time when on), and the current temperature (except for when in toast mode). Additionally a greater-than sign (>) is used in Bake mode to indicate whether time or temp is configurable through the potentiometer.[3]
  - A function called updateOvenOLED() should contain all OLED-altering code. Note that the file Ascii.h contains several special characters that are useful for drawing the oven.
- First, the user will select a mode and configure a cook time and/or temperature:
  - The toaster oven has 3 cooking modes, which can be rotated through by pressing BTN3 for < 1s. They are, in order: bake, toast, and broil.
    - **Bake mode:** Both temperature and time are configurable, with temperature defaulting to 350 degrees F and time to 0:01. Switching between temp and time can be done by holding BTN3 for > 1s (defined as a LONG_PRESS).

[2] As you gain experience with embedded systems, you will realize that once you have the full state machine working on the OLED, adding the hardware to control an actual toaster oven is not all that more difficult. In many situations, the software behavior is more difficult than the hardware interface.

[3] See the Expected Output section for how the output should look. Yours should be very similar, but not necessary identical.

Whichever is selected has an indicator beside its label (the selector should always default to time when entering this mode). Both top and bottom heating elements are used when cooking in bake mode.

- **Toast mode:** Only the time can be configured in this mode, and the temperature is not displayed. There is no selector indicator on the display. Only the bottom heating elements come on in toast mode.
- **Broil mode:** The temperature is fixed at 500 degrees F and only time is configurable in this mode. The temperature is displayed in broil mode. Again, the input selector indicator is not displayed. Only the top heating elements come on in broil mode.

- While in this phase of operation, the user can rotate the potentiometer to adjust the time or temperature. There is a 2-state variable, called the "settings selector", which determines which setting the pot controls.
  - The settings selector is switched by holding BTN3 for >1 second.
  - The cooking time is derived from the ADC value obtained from the Adc library by using only the top 8 bits of the ADC reading and adding 1. This results in a range from 0:01 to 4:16 minutes.[4]
  - The cooking temperature is obtained from the potentiometer by using only the top 8 bits of the ADC value and adding 300 to it. This allows for temperatures between 300 and 555.

- Once a mode, time, and (if appropriate) temperature are selected, then cooking is started by pressing down on BTN4. This turns on the heating elements on the display (as they're otherwise off) and the LEDs (see below).
  - Cooking can be ended early by holding down BTN4 for >1 second. This should reset the toaster to the same cooking mode that it was in before the button press.
    - Additionally, if the time/temp selector should return to their settings when baking started. So, if the user selects 1:00 minute, then cooks for 30 seconds, then cancels cooking, the timer should now say 1:00 minute.
  - When the toaster oven is on, the 8 LEDs indicate the remaining cook time in a horizontal "progress bar" to complement the text on the OLED. At the start of cooking, all LEDs should be on. After 1/8

[4] That is, you will need to mask or shift to isolate the top 8 bits of the ADC, and each one of those ticks is worth one additional second to the time set (4:15 is 256 seconds).

of the total time has passed, LD1 will turn off. After another 1/8 of the original cook time, LD2 will turn off, and so on until all LEDs are off at the end.
    - After cooking is complete, the system will return to the current mode with the last used settings; the heating elements should be off, the time and temperature reset to the pot value, and the input selector displayed if in bake mode.

- **Extra credit:** After cooking is complete, the toaster oven enters an "alert" mode, blinking its screen at intervals.
    - **o** You will need to use the Oled library's invert function, and add a state to the state machine.
    - **o** Describe your implementation in your readme.

Code requirements:

When implementing the toaster oven functionality, your code should adhere to the following restrictions:

- The template code has two timer ISRs. Keep them very short and simple! The idea is to get back to your main code as soon as possible.
    - The 100Hz timer should be used exclusively to check for button events and ADC events. This ensures the system is very responsive to button presses.
    - The 5Hz timer:
        - Sets a TIMER_TICK event flag
        - Increment the freerunning timer. This timer is never reset, but is instead used as a "global" timer. This timer is used to determine whether a button press was a long press or a short press, and is used to determine cooking progress. This is a useful technique when you need to time multiple events using a single timer (discussed below).

- No floating point numbers are allowed in your code. You will be performing integer math to get all required values.
    - Floating point operations are much slower than standard arithmetic! While it doesn't matter much in this lab, it's important to build good habits now[5]

- Implement all of your Toaster Oven's behavioral logic with a single state machine, which you should keep in the runOvenSM() function.
    - Your state machine should use a single switch statement to check the state variable, and each case should use if()

[5]Since your PIC32 doesn't have a floating-point unit as part of its CPU.

statements to handle particular events. Don't forget to break or return!
    - You may (of course) use helper functions to keep your state machine clean, but the overall logic must still follow these rules.

- Your toaster should be an entirely event-driven system.
    - Your state machine should be called from your main loop (ie, in a while(1) inside of your main()), but it should only run when an event occurs.
    - Your main loop should not do anything besides polling the event flag and calling the state machine when appropriate.
        - It should DEFINITELY not call updateOvenOLED() directly. This function is very slow, and should only be called when necessary.
    - This state machine code should *NOT* be called directly by an interrupt. Interrupts should set event flags that are checked by the state machine loop and passed into runOvenSM().
    - Don't forget to clear your event flags once your state machine is done with them!
- Create a single struct that holds all toaster oven data: oven state (what state the oven state machine is in), temperature, cooking mode, button press time, cooking start time, and input Selector (whether the pot affects time or temp).
    - Create a single instance of this struct as a module-level variable.
    - Make sure each member variable has comments indicating what they hold and their units, if any.

- Used named constants and types:
    - Create a single typedef'd enum to name your states. Use an instance of this type in your module-level struct.
    - Use the same technique to name your cook modes and Selector settings.
    - All constants must be declared as constants using either #define, enum or const variable. This includes constants like 300 (degrees) or 5 (ticks per second).

- Use safe data practices:
    - Any variables created outside of main must be declared static so that they exist only as module level variables.
    - Any strings used to specify formatting for (s)printf() should be declared as const to allow for compiler optimizations.

General:

- Format your code to match the style guidelines in the ECE013E_StyleGuidelines document.
- Make sure that your code triggers no errors or warnings when compiling. **Compilation errors in libraries will result in no credit for that section**, which is nearly all of the points available in this lab!
    - Commit often and submit early! That way you can be sure to have a compiling version that you can submit.
- Compilation errors in the main file will result in NO CREDIT. Any compilation warnings will result in two lost points.

Lab Writeup:

Use README.md containing the following items, utilizing Github-flavored Markdown to format your document. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.

- First you should list your name & the names of colleagues who you have collaborated with.[6]
- In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
- The subsequent section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? How did you work with other students in the class and what did you find helpful/unhelpful?
- The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What did you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understanding this lab or would more teaching on the concepts in this lab help?

[6] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

Grading

This assignment consists of 23 points, with a 1-point "cushion".

Note that all components are human-graded (there's just not much we can automate in this lab!)

- 9 points: Toaster oven functionality
  - 3 points: Oven cycles through three selectable modes of operation, and settings can be selected and adjusted as appropriate.
  - 2 points: Oven counts down, and resets when complete.
  - 3 points: Oven correctly displays the required information on the OLED and LEDs.
  - 1 point: Oven cannot get "stuck," so that the board must be reset.
- 10 points – Followed the specs for code organization, interrupt handling, and output.
  - 5 points: Oven is fully event-driven
  - 3 points: State machine is implemented correctly
  - 1 point: Oven timing is correct.
  - 1 point: All data is appropriately scoped.
- **1 point extra credit** – Invert the display at 2Hz after cooking completes.
- 2 points: Code style
- 2 points: Writeup
- 1 point: Leds.h macros get, set, and initialize LEDs correctly.
- -X points:
  - **NO CREDIT for code with compilation errors**
  - -2 points: at least 1 compilation warning
  - -2 points: for using gotos, extern, or global variables
  - Additional deductions at grader discretion
  - Additional bonuses at grader discretion, particularly for README.md explanations of features that didn't quite work.

Free Running Counters

In an embedded system, it is often a requirement to know how much time has elapsed between two events. You can see this in everyday use in many everyday electronic items. There are several ways to accomplish this. One is to dedicate a hardware timer to be started on the first event, and stopped when the second occurs. This is very precise, assuming you can spare the hardware timer, and that the longest time elapsed between events is within the range of the timer. However, if you are going to do this for more than a single pair of

events, then you will be using up all of your hardware timers on this task alone.

Another method is to use what is called a free running counter: a timer is set to periodically interrupt, and increments the free running counter.[7] To find out the time elapsed, copy the value of the free running counter to a variable, startTime, when the first event occurs, and on the second event:

Elapsed Time = current FreeRunning Time – start Time;

The elapsed time will be in units of timer event ticks. Note that you can do this for as many things you want elapsed time for without using any additional hardware. Because this is all happening in integer 2's-complement math, a single rollover on the free running counter does not alter the results of the calculation. Two rollovers of the free running counter between first and second event would be required to give you the wrong elapsed time. If your timer is ticking at 5Hz, and you use a 16-bit integer, then you have over 3 ½ hours before you get a wrong elapsed time.

Integer Math

Most embedded systems or microcontrollers do not have a full floating point unit built into their hardware. As such, floating point math must be emulated and is very processor intensive. We can use floats if we must, but try to minimize their use to absolute necessities.

The larger truth is that you don't often need floating point, but can get by using integer math (or fixed-point math) most of the time. However, using integer math does require some care in the order of operations to make sure you don't get the wrong results.[8]

For example, let's say you wanted to convert your ADC reading (an unsigned 10 bit integer) to a percentage. The mathematical formula is straightforward:

$$ADC\% = \left(\frac{ADCReading}{1023}\right) \times 100$$

```
ADCpercent = (ADCGetValue() / 1023) * 100;
```

[7] The free running counter is a module-level variable, which can be accessed by any function within the module. It is always declared as a static, and is usually made large enough that double roll-overs do not occur very often. For example: static uint16_t freeRunningCounter;

[8] This drives mathematicians crazy, but it makes sense when you think about it.

However if you implement it that way, you will always get 0% as a result because the integer divide occurs first (and the result will always be 0).[9] Thus the correct way to implement it would be:

```
ADCpercent = (ADCGetValue() * 100) / 1023;
```

You must take care that the initial multiplication can fit into the variable size without overflowing. If it might overflow, then use a "cast" to temporarily increase the size for the calculation:

```
ADCpercent = ( (uint32_t) ADCGetValue() * 100) / 1023;
```

Lastly, integer math additions and subtractions work when using 2s complement math such that you get the right result even when going past the number boundaries. If you are going to divide or multiply by a power of 2, use shifts instead. They are much faster and typically directly supported by underlying hardware.

Macros

Macros are a very powerful tool that can make code simpler to implement, easier to understand, and faster (not necessarily all three every time!). But with great power, comes great responsibility. The macro system in C (unlike some more modern languages) does not do any input checking and mostly does straight textual substitution. This makes it very easy to run into problems.

The most basic usage for macros are for declaring constants using the `#define` preprocessor directive like:

```
#define TRUE 1
```

Using a constant like this makes your code easier to read, so instead of a number which you may not understand, you have a name that hopefully hints to its function. The other advantage to this is that, unlike using the `const` modifier, no memory space is used to store this number; it is merely substituted in for the name wherever it appears in the source code.[10]

---

[9] The result will be 0% and then at the very top occasionally 100%, and nothing in between. The reason is that the integer divide by 1023 will result in either 0 or 1 (when ADC is 1023). That is why you need to do the multiplication first, then the divide. Since both multiplication and division have the same order of precedence, you need to force it with parentheses.

[10] This also means that if you need to change a "magic number" in your code, you only change it at the #define line and the rest of the code is changed automatically. This ensures that you don't have to hunt down every usage of the magic number (and likely introduce a bug when you forget one).

Using macro constants is the most basic usage of macros. A more advanced use is for writing multiple-statement blocks. These aren't functions in the regular sense, they don't return anything, but can take arguments. What they are is really a smarter way to do a straight text replacement, basically templated textual replacement.

For example, here's a macro that just divides the PR2 by a number:

```
#define DIVIDE_IT(num) (PR2 = PR2 / (num))
```

So if you call it like follows:

```
DIVIDE_IT(5);
```

Everything will compile properly and the resulting code would be:

```
(PR2 = PR2 / (5));
```

So whatever text is passed in as the argument is substituted for "num" in the output text. Also note that we included parenthesis around num. This is to make sure the following works correctly: `DIVIDE_IT(5 + 12*x);` If we didn't include the parenthesis, the text output would be:

```
(PR2 = PR2 / 5 + 12*x);
```

, which is not what we intended. We also didn't include a semicolon in the macro because we want the user to add one as they normally do after regular statements in C and for the user to understand that they can treat the macro as a single statement, much like a conventional C function call. But with multiple statements in a macro we have to do something a little extra to keep that functionality. We use curly braces to ensure that the compiler treats this code as a single block (the "\" are to continue the macro on the next line):

```
#define INIT_ECAN(bsize) { \
      CB_Init(&ecan1TxCBuffer, txDataArray, bsize); \
      CB_Init(&ecan1RxCBuffer, rxDataArray, bsize); \
}
```

So with that macro written using a {} block, we can call it like a single function call:

```
INIT_ECAN(10);
```

And now the semi-colon works properly and the code is grouped into a single statement.

To help debug macro-related issues, MPLAB X gives you a very powerful tool. Right click anywhere on your code and select Navigate -

> View Macro Expansion. The window that opens at the bottom shows the same portion of the code that you have shown in the main coding window, but with all the macros expanded and inlined.

Finite State Machines

A finite state machine (FSM) is a construct used to represent the behavior of a reactive system. Reactive systems are those whose inputs are not continuous signals, but rather momentary events. In the context of this class inputs are the same as the events we have already used (i.e. Timers, Buttons, ADC, etc.).

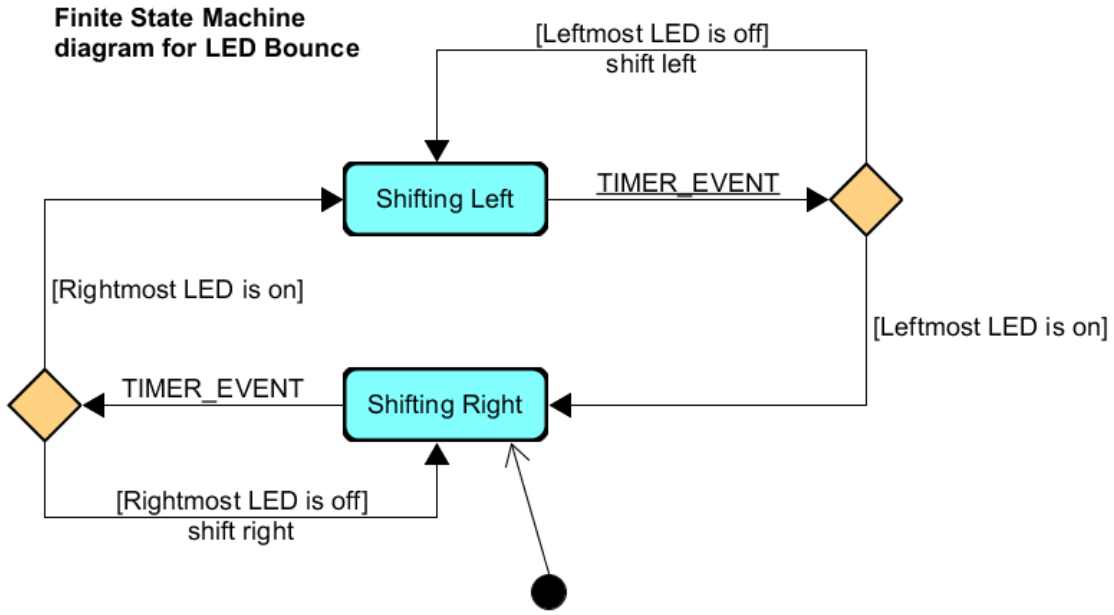
The Wikipedia link along with the assigned reading for this lab should give you a fairly complete picture of state machines.

As an example of how applicable FSMs are, the following diagram shows the behavior of an LED bouncing with the behavior you implemented in the Bounce lab. It uses common FSM notation[11].

[11] In particular, we (attempt to) use Unified Modeling Language (UML) notation, an ISO standard notation for describing state machines. If you want to see the specification, it's in chapter 15 of http://www.omg.org/spec/UML/2.2/Superstructure/PDF. This notation was codified by David Harel in http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf, which is a much better read.

The primary features in this diagram are:

- The arrow from a dot to “Shifting Right”. An arrow from a dot to a state indicates the initial state of the system.
- Transitions between states are written with an event above a line and the actions during the transition below the line. Sometimes there are no actions.
    - If no transition conditions are met for a state it is implied that the system stays in that state until one of the conditions is met.
    - On some occasions, a state will have an arrow transitioning to itself. The state machine will take the indicated action, but not change its state.
- Some events are handled differently depending on some other condition.[12] These “guard conditions” are denoted with [square brackets], and are sometimes highlighted using diamond-shaped nodes.



[12] This extra information is sometimes called the “extended state” of the state machine. It is, technically, part of the state of the system, so a pure state machine wouldn’t have guard conditions. However, representing a complex system fully as a pure state machine would require a *lot* of paper and ink. For example, a PIC32 has 512kb of memory, so it’s full state machine would have 8^(512k) states! The whole purpose of state machine design and analysis is to conceptually *simplify* a system, so extended state is inevitable.

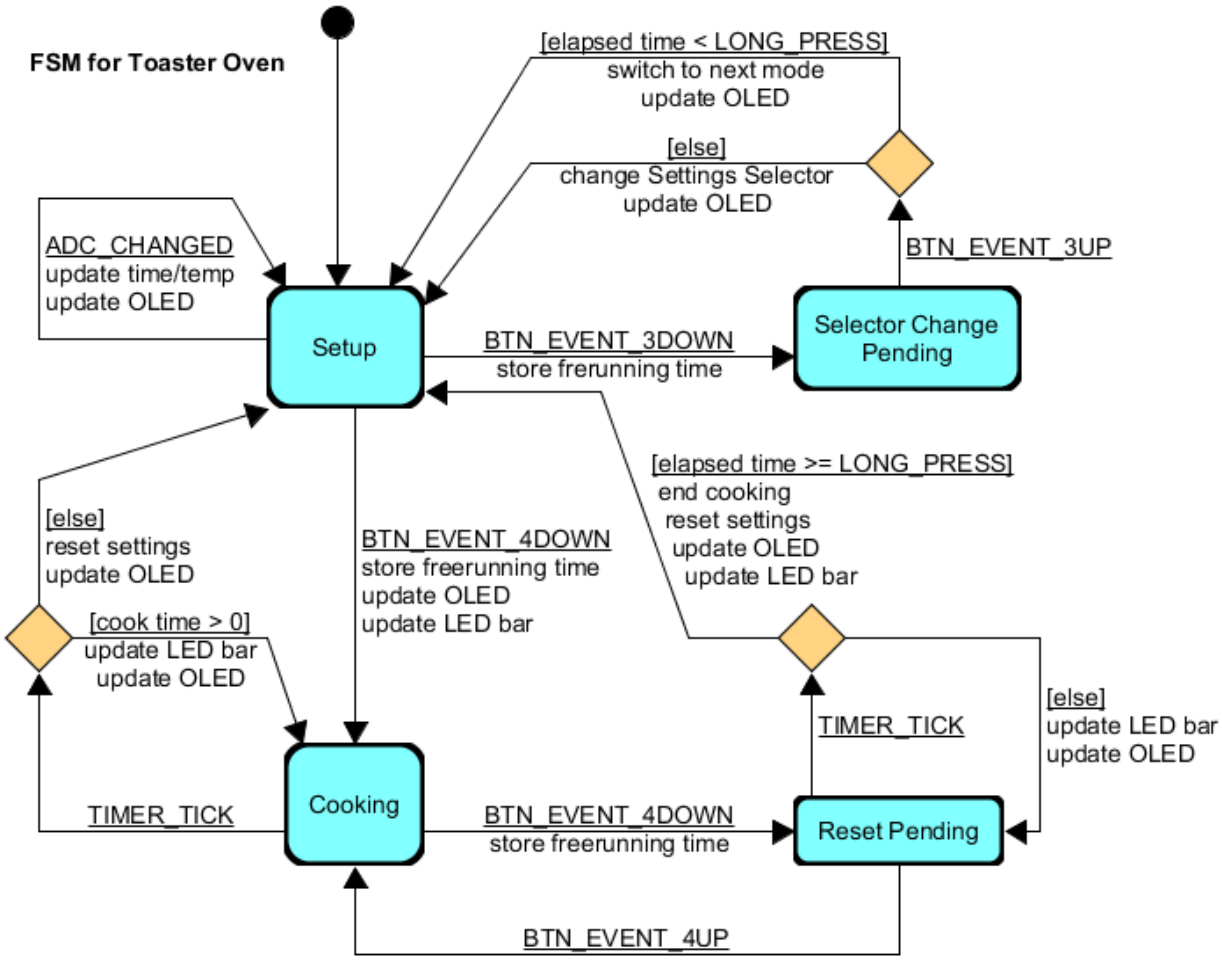
Now to actually implement state machines a large switch-statement is used that switches over the system state. The corresponding C code for the above state machine would look like:

```
// Initialize:
enum {SHIFTING_LEFT, SHIFTING_RIGHT}
      state = SHIFTING_LEFT;
LEDS_SET(0x01);

// Run state machine:
while (1) {
      if(timerResult.event){
            switch (state) {
            case SHIFTING_LEFT:
                  //check guard condition:
                  if(LEDS_GET() & 0x80){
                        state = SHIFTING_RIGHT;
                  } else {
                        LEDS_SET(LEDS_GET()<<1);
                  }
                  break;

            case SHIFTING_RIGHT:
                  //check guard condition:
                  if(LEDS_GET() & 0x01){
                        state = SHIFTING_LEFT;
                  } else {
                        LEDS_SET(LEDS_GET()>>1);
                  }
                  break;
            }
            timerResult.event = FALSE;
      }
}
```

Note that state machines can get very hard to navigate as the get large. Keep your state machine diagram close at hand – it's a good idea to print out a copy.

Toaster Oven FSM

For this lab you will implement the following state machine within a single switch statement in `main()` in the provided Lab07_main.c file provided to you.



There are four states in this FSM: SETUP, SELECTOR_CHANGE_PENDING, COOKING, and RESET_PENDING. Six events are of concern, although any button event is possible: TIMER_TICK, ADC_CHANGED, and the button events for buttons 3 and 4.
Trace through this diagram many times before you sit down to code it!

Approaching this lab

There is no library for this lab. Instead you will be integrating previously compiled libraries, including a new one, Adc.h. Since you will be implementing a state machine, the easiest approach is to implement one state transition at a time. By starting with the display functionality in this lab, being able to confirm state transitions will be simple because then you can check your state data by looking at the OLED.

A high-level plan is detailed below.

1. Write the updateOvenOLED() function to populate the OLED with constant data, using the Example Output as a template. As you go, you can modify this function to report more data dynamically. (You will almost certainly want to use several calls to sprintf() to do this).

2. Create a struct for storing all the oven state data[13]: cooking time left, initial cook time, temperature, button press time, and your states/modes. Make an instance of this struct modify your function to populate the OLED, given this struct with the correct output. Test your OLED function.

3. Create the SETUP state constants. This will properly display the initial toaster oven state.

4. Implement the transition from SETUP to itself when the potentiometer changes. You should now see the initial toaster oven state displayed and updated when the potentiometer is changed.

5. Implement SETUP's BUTTON_3DOWN transition, the PENDING_SELECTOR_CHANGE state, and the elapsed_time < LONG_PRESS transition. You now are able to change the time for all 3 states and cycle between them.

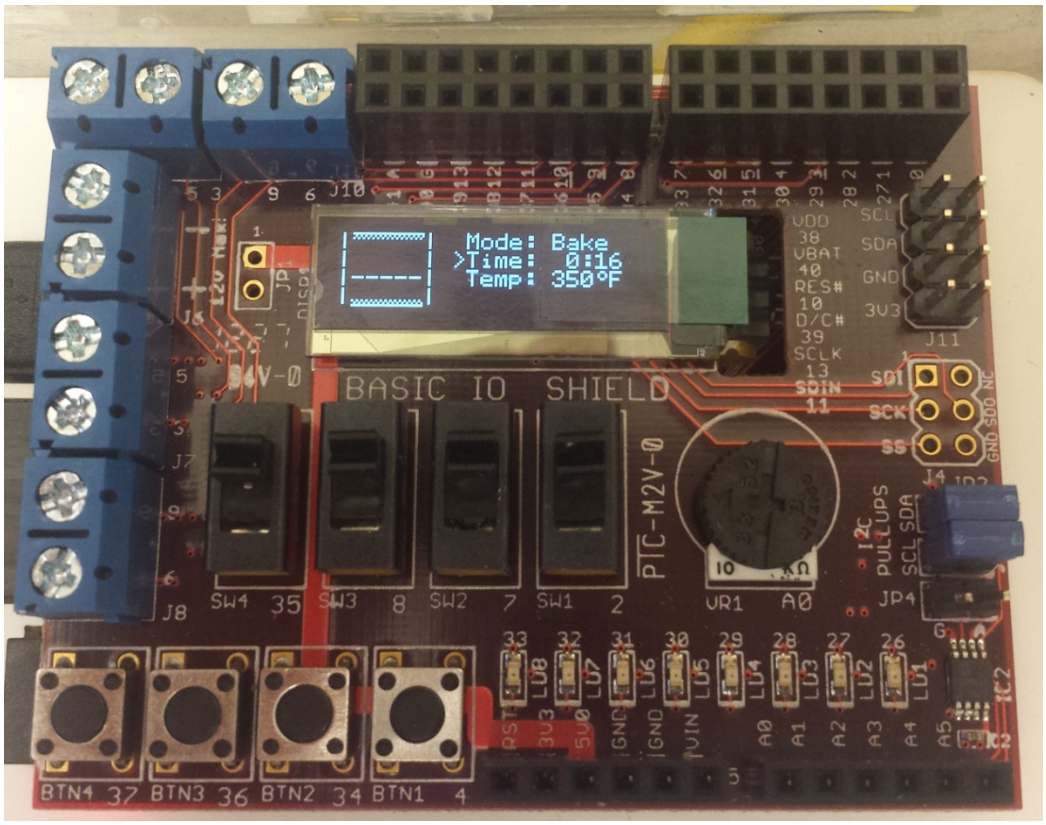
6. Implement PENDING_SELECTOR_CHANGE's elapsed_time >= LONG_PRESS transition. Now the time **and** temperature can be changed when in bake mode. The selector should now also be properly displayed for the bake mode depending on whether time or temperature is being modified (and not displayed in the other cooking modes).

[13] Note that there are three separate states (or modes) you need to keep track of: (1) the state within the state machine diagram, (2) the cooking mode, and (3) the selector state.

7. Implement the COOKING state and the necessary transitions to have the oven start cooking and count down the time. This includes updating the LEDs and the OLED time left.
8. Add the reset transition so that after the toaster oven has finished cooking, it resets its settings to before cooking started, and returns to the SETUP state.
9. Implement the PENDING_RESET state so that pressing and holding BTN4 resets the oven state.
10. Check your coding style. Make sure you have enough comments!
11. Check that you fulfilled all lab requirements by re-reading this manual.
12. Submit your assignment.
13. Implement the extra credit.
14. Submit again.

Example output

The following is a picture of the toaster oven when off in the bake mode:


Mode: Bake
>Time: 0:16
Temp: 350°F
BASIC IO SHIELD
PTC-M2V-0
BTN4 37 BTN3 36 BTN2 34 BTN1 4
SW4 35 SW3 8 SW2 7 SW1 2
VR1 A0
SDIN 11
SCK
SS
I2C PULLUPS SCL SDA
VDD 38
VBAT 40
RES# 10
D/C# 39
SCLK 13
3V3
GND
SCL
SDA
J11
J8
J10
JP4