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

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 7 - Toaster Oven

Commit ID Form: https://goo.gl/forms/ZKdtHSFGArkBjB3v1

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 stated as such. They are a very powerful technique for organizing the behavior of complex systems[1] (and more 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
- Free Running Counters
- Event-driven programming

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

- Finite state machines

Reading

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

Provided files

- toaster_oven.c – contains `main()` and other setup code.
- BOARD.c/h - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values. **You will not be modifying these files at all!**
- Lab7SupportLib.a (Oled.h, OledDriver.h, Ascii.h, Adc.h, Buttons.h) – These files provide helper functions that you will need to use within your toaster_oven.c file. The Lab7SupportLib.a file is a precompiled library containing all of the functions listed in the Oled, Adc, and Buttons header files. You will only need to use the functions listed within Oled.h, Adc.h, and Buttons.h to do this lab, the other headers are merely required for Oled.h to compile.
    - To use these functions, add Lab7SupportLib.a as a Library File within your MPLAB X project by right-clicking on the Libraries folder in your project and selecting "Add Library/Object File..."
    - Be sure to read up on how to use the Adc library as this is a new one!
    - Ascii.h describes the various characters available for use with the display. There are 4 special characters defined here for drawing the heating elements.

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 toaster_oven.c utilizing the provided libraries.

- **Toaster Oven Functionality**:

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

- 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]
- 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). 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.
- The cooking time is derived from the ADC value obtained from the Adc library by using only the top 8 bits and adding 1 resulting 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.
- 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

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

[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).

the button press. Additionally, if the cooking mode was Bake, the time/temp selector should stay the same as it was when baking started.

- When the toaster oven is on, the 8 LEDs indicate the remaining cook time in a horizontal bar graph to compliment the text on the OLED. At the start of cooking, all LEDs should be on. After 1/8 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 or the defaults described above), and the input selector displayed if in bake mode.

- **Code requirements:** When implementing the toaster oven functionality, your code should adhere to the following restrictions.
  - No floating point numbers are allowed in your code. You will be performing integer math to get all required values.
  - Implement all logic within a single state machine in main() that uses a single switch statement to check the state variable and perform the necessary actions. This state machine code should not be called directly by an interrupt, but should be in a while loop in main(). Interrupts should set event flags that are checked by the state machine loop.
  - Create a single enum for holding all the state constants for your toaster oven. Use this enum as the data type for the variable holding the system state.
  - All constants (except for those used in mathematical calculations) must be declared as constants using either #define, enum or const variable.
  - 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.
  - Create a single struct that holds all toaster oven data: cooking time left, initial cook time, temperature, cooking mode, oven state (what state the oven state machine is in), button press counter, and input selection (whether the pot affects time or temp). Create a single instance of this struct as a module variable. Make sure each member variable has comments indicating what they hold and their units, if any.
    - The cooking times are stored as integers but your timer interrupt is at 2Hz. As such, you will need to store double the time and update the OLED with half this value.[5]

[5] Note that when using integer math, especially when dividing by a power of 2, shifts are much faster (and cleaner) than dividing. Thus a ÷2 is equivalent to right shift by 1.

    - Updating the OLED is only done when the system state changes. This process is too slow to do every time through the switch-statement for the system state. The OLED is also only updated by writing characters to all 4 lines of text on the screen.[6] The heating elements are stored in characters 0x01 (top, on), 0x02 (top, off), 0x03 (bottom, on), and 0x04 (bottom, off).
    - When using the ADC library you should use the AdcChanged() value as an event for determining if you need to update anything.
    - Updating the LEDs should also only be done when the system state changes.
    - The 100Hz timer should be used exclusively to check for button events. This ensures the system is very responsive to button presses.
    - The 2Hz timer is used exclusively for setting a flag when its interrupt occurs that can be checked in the state machine.
    - The 5Hz timer also sets an event flag when its interrupt occurs as well, but also increments the free running counter for use in distinguishing if a LONG_PRESS has occurred (where LONG_PRESS is a constant you will define with a value of 5, for representing 1 second of time). You will not be resetting this counter but instead be recording the time as part of the event. You can then check if the timing event has occurred by subtracting this count off the current count (i.e. (FreeCount-StartCount) >LONG_PRESS would indicate that a long press has occurred).
- Format your code to match the style guidelines that have been provided.
- Make sure that your code triggers no errors or warnings when compiling. Compilation errors in libraries will result in no credit for that section. Compilation errors in the main file will result in NO CREDIT. Any compilation warnings will result in two lost points.
- **Extra credit:** Add an additional 2 states to your state machine (with their corresponding names added to the state enum) and use them to invert the screen at 2Hz once the cooking timer expires (look for useful functions within Oled.h). When in either of these states pressing BTN4 should return to the START state for the user to start cooking again.
- Create a readme file named README.txt containing the following items. 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.

[6] Writing to the OLED is very slow, so if you end up doing this every time through the event loop it's likely your program or OLED won't work correctly.

- First you should list your name & the names of colleagues who you have collaborated with.[7]
- 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'd 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?

Required Files

- toaster_oven.c
- Leds.h[8]
- README.txt

Grading

This assignment consists of 21 points:

- 8 points – Followed the specs for code organization, interrupt handling, and output.
- 12 points – Implemented all toaster functionality.
- 1 point - Provided a readme file
- 1 point – Follows style guidelines (<= 5 errors in all code) and has appropriate commenting

[7] 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.

[8] Leds.h **IS NOT PROVIDED!** You will need to use your header file from Bounce and use it in this lab (or rewrite it if you didn't get it working then). You will submit this file with the rest of your assignment. This file must be written by you, if it is not then that is cheating!

- 1 point – GIT Hygiene
- **1 point extra credit** – Invert the display at 2Hz after cooking completes.

You will lose points for the following:

- NO CREDIT for compilation errors
- -2 points: any compilation warnings
- -2 points: for using gotos, extern, or global variables

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 if 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.[9] 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:

elapsedTime = freeRunningCounter – startTime;

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 use floats if we must, but try to minimize their use to absolute necessities.

[9] 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;

The larger truth is that you don't often need floating point, but can get by using integer 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.[10]

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 (and in C):

$$\boldsymbol{ADC}\% = (\frac{\boldsymbol{ADCReading}}{\mathbf{1023}}) \times \mathbf{100}$$

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

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).[11] 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 2SC 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 usually cleaner.

Finite State Machines

As defined by CKO, 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 all active at a single point of time. 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.

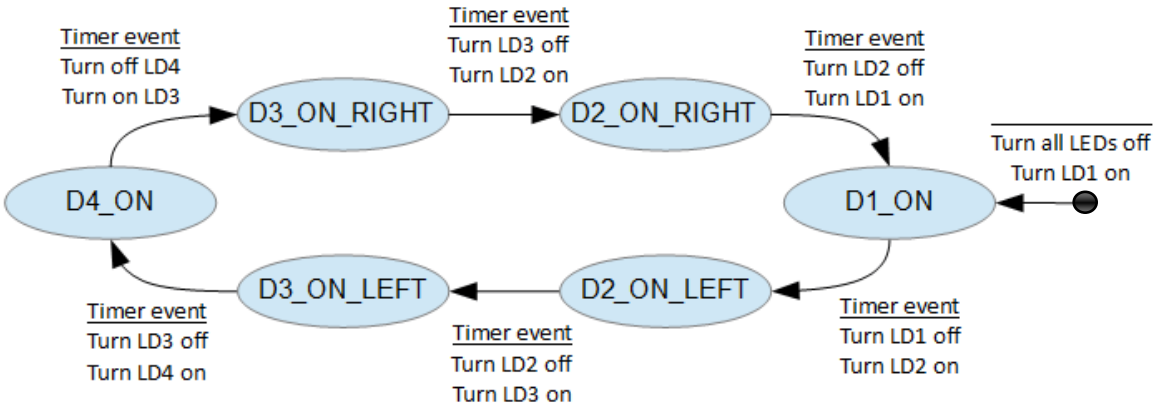
[10] This drives mathematicians crazy, but it makes sense when you think about it.

[11] 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.

As an example of how applicable FSMs are, the following diagram shows the behavior of an LED bouncing between LEDs LD1 and LD4, a subset of the bouncing behavior you implemented in the Bounce lab. It uses common FSM notation. Note that the state names are descriptive, indicating which LED is on and what direction the LED is going.[12]

The primary features in this diagram are:

- The arrow from a dot to D1_ON. An arrow from a dot to a state indicates the initial state of the system, in this case the initial state of the system is in D1_ON.
- Transitions between states are written with the conditions of the transition above a line and the actions during the transition below the line. Sometimes there are no conditions and sometimes there are no actions, either of which is perfectly acceptable.
- 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.



Using an FSM simplifies the logic as there's no more explicit tracking of which LED is lit or what direction the LED is bouncing; that data is encoded directly in the states themselves and their transitions. For some problems, using a FSM can drastically simplify the code and make implementation much simpler.

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:

```
enum {D1_ON, D2_ON_LEFT, D3_ON_LEFT, D4_ON, D3_ON_RIGHT, D2_ON_RIGHT
     } state = D1_ON;

// Perform the initial transition
```

[12] Note here that we wouldn't implement the LEDs bouncing in this way, but it is an example and builds off of what you have seen already in the bounce lab. You will find it very useful to implement this state machine and see how the LEDs move.

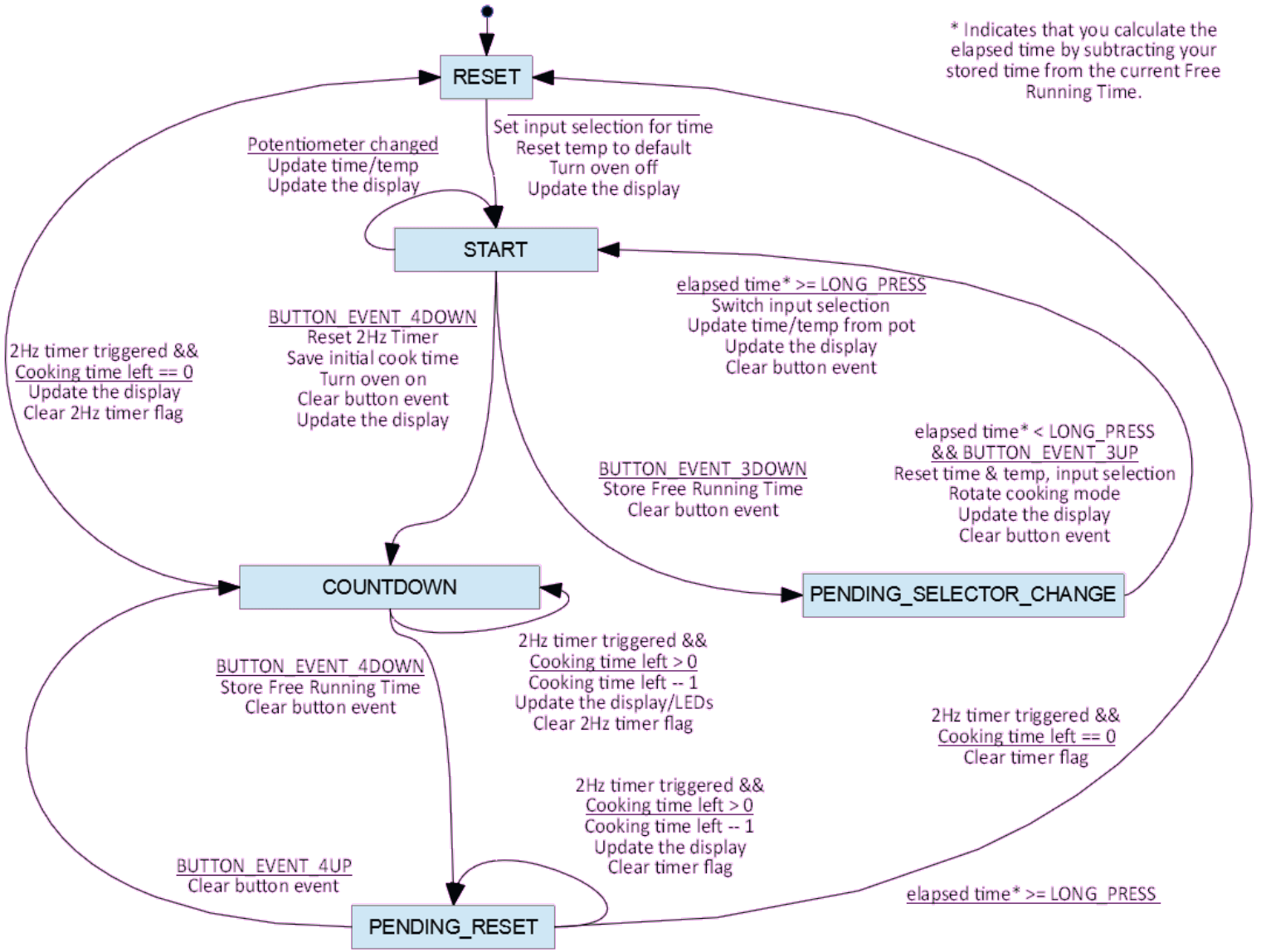
```
LEDS_SET(0x01);
while (1) {
      switch (state) {
      case D1_ON:
            if (timerResult.event) {
                  state = D2_ON_LEFT;
                  LEDS_SET(0x02);
                  timerResult.event = FALSE;
            }
            break;
      case D2_ON_LEFT:
            if (timerResult.event) {
                  state = D3_ON_LEFT;
                  LEDS_SET(0x04);
                  timerResult.event = FALSE;
            }
            break;
      case D2_ON_RIGHT:
            if (timerResult.event) {
                  state = D1_ON;
                  LEDS_SET(0x01);
                  timerResult.event = FALSE;
            }
            break;

      ...
```

Toaster Oven FSM

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

You have access to 3 separate timers now: a 2Hz, 5Hz, and 100Hz timer. Some functionality, like the primary cook timer, will use the 2Hz timer. The 5Hz timer is used for tracking how long the buttons are held down through a free running timer. And the 100Hz timer is exclusively checking for button events.


* Indicates that you calculate the elapsed time by subtracting your stored time from the current Free Running Time.
RESET
Set input selection for time
Reset temp to default
Turn oven off
Update the display
Potentiometer changed
Update time/temp
Update the display
START
elapsed time* >= LONG_PRESS
Switch input selection
Update time/temp from pot
Update the display
Clear button event
BUTTON_EVENT_4DOWN
Reset 2Hz Timer
Save initial cook time
Turn oven on
Clear button event
Update the display
2Hz timer triggered &&
Cooking time left == 0
Update the display
Clear 2Hz timer flag
elapsed time* < LONG_PRESS
&& BUTTON_EVENT_3UP
Reset time & temp, input selection
Rotate cooking mode
Update the display
Clear button event
BUTTON_EVENT_3DOWN
Store Free Running Time
Clear button event
COUNTDOWN
PENDING_SELECTOR_CHANGE
2Hz timer triggered &&
Cooking time left > 0
Cooking time left -- 1
Update the display/LEDs
Clear 2Hz timer flag
BUTTON_EVENT_4DOWN
Store Free Running Time
Clear button event
2Hz timer triggered &&
Cooking time left == 0
Clear timer flag
2Hz timer triggered &&
Cooking time left > 0
Cooking time left -- 1
Update the display
Clear timer flag
BUTTON_EVENT_4UP
Clear button event
elapsed time* >= LONG_PRESS
PENDING_RESET

Approaching this lab

There is no library for this lab, which is the usual place to start. Instead you will be solely integrating 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 just look at the OLED.

A high-level plan is detailed below.

1. Initialize and populate the OLED with static data, using the Example Output as a template.

2. Create a struct for storing all the oven state data[13]: cooking time left, initial cook time, temperature, cooking mode, oven state (what state the oven state machine is in), button press counter, and input selection (whether the pot affects time or temp). Initialize one of these structs and write a function that populates the OLED given this struct with the correct output. Reproduce the same output as from Step 1.

3. Create the RESET state and START state constants as well as the transition from RESET to START. This will properly display the initial toaster oven state.

4. Implement the transition from START 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 START's BUTTON_3DOWN state transition, the PENDING_SELECTOR_CHANGE state, and the button counter < 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 button counter >= 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 if it is on or off, and (3) the selector state.

7. Implement the COUNTDOWN state and the necessary transitions to have the oven start cooking and countdown (don't reset yet). 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 back to the same state it had been cooking in.
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: