Lab 3 - Virtual Train

Due Jul 6 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** txt

Introduction

In this lab we will be simulating a train using the Seven Segment Display (SSD) from the Grove Starter Kit with the MSP-EXP430FR2355. The SSD has 4 7-segment character displays which we will use to create the image of a "train" which will drive in a circle around the edge of the character display. The train will be able to be stopped and have its direction changed by use of the two push buttons on the bottom of the board. We will also use the LEDs on the bottom to indicate the state of train: when the train is moving the green LED will be illuminated and when the train is stopped the red LED will be illuminated.

Objective

This lab will exercise your ability to implement a software state machine to control the train. You will create custom functions to represent each state, use enumerations to define custom data types, and create structures for holding all of the state machine's inputs. You will also import a custom library for writing to the SSD. You will not need to understand how the interface works, only how to use the library.

Required Materials

- MSP-EXP430FR2355
- Seeed Studio Grove Seven Segment Display
- Grove connection cable
- USB micro cable
- Laptop (Windows preferable)

Procedure

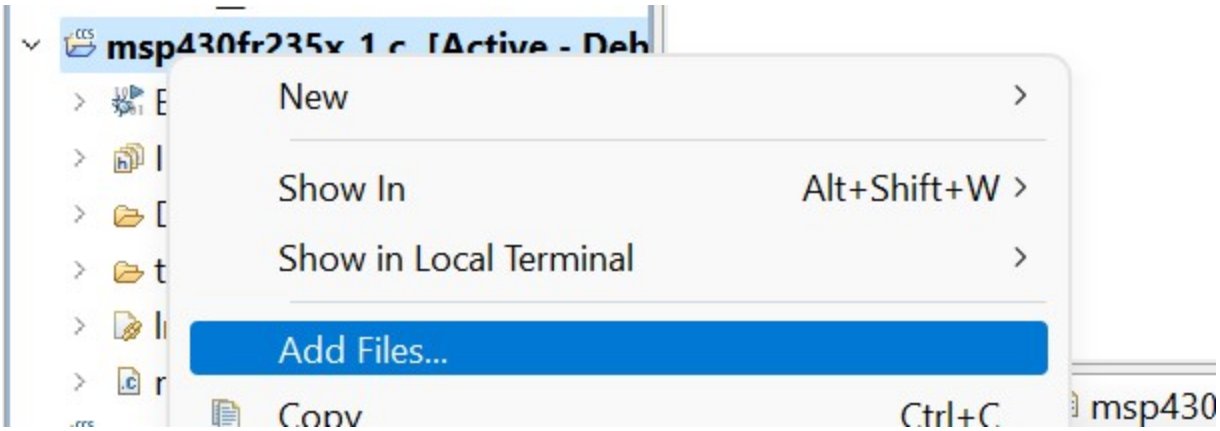
Step 1: Importing a New Library

The first step of this exercise is to import the custom library files for interfacing with the SSD. The library files can be found below:

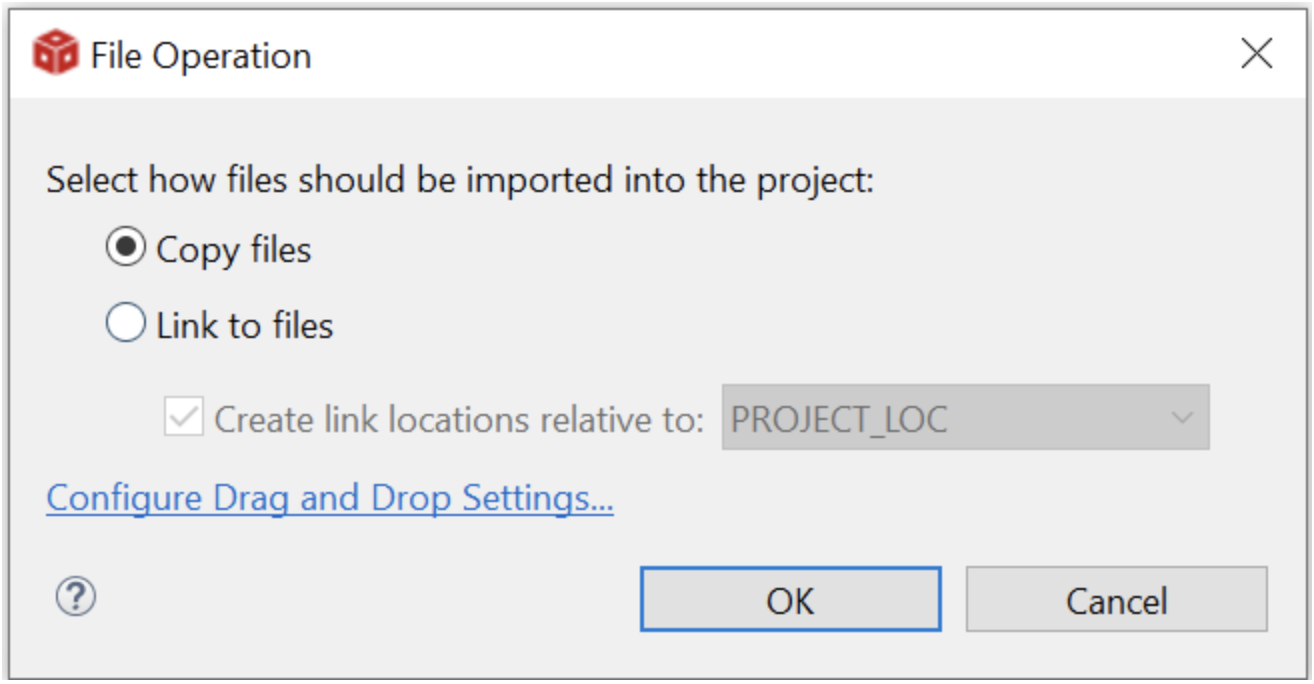
Four_Digit_Display.h (https://uncc.instructure.com/courses/197484/files/20876895?wrap=1) ⬇ **(https://uncc.instructure.com/courses/197484/files/20876895/download?download_frd=1)**

Four_Digit_Display.c (https://uncc.instructure.com/courses/197484/files/20876894?wrap=1) ⬇ **(https://uncc.instructure.com/courses/197484/files/20876894/download?download_frd=1)**

After downloading the files and **creating a new project (https://uncc.instructure.com/courses/197484/pages/creating-a-new-ccs-project)**, you will then need to import the library files. Right click your project name in the project and select "Add Files...".



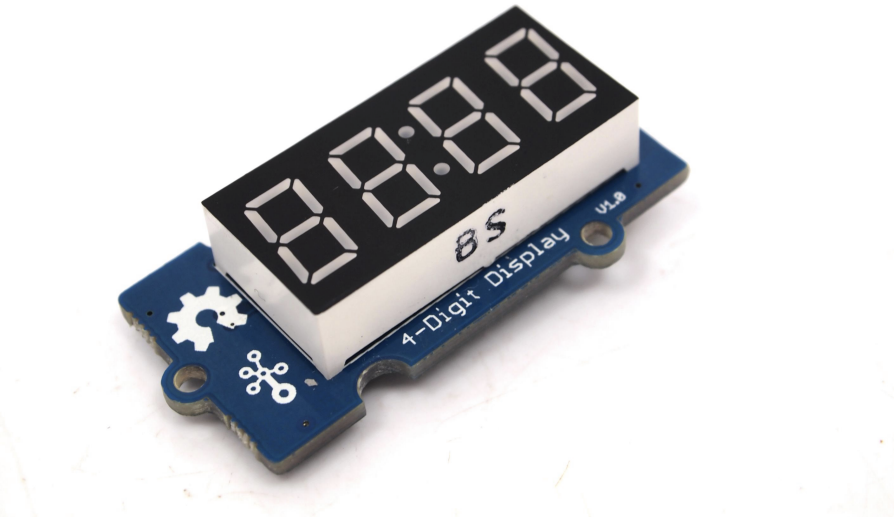
From here you will be able to navigate to the location where you downloaded the Four_Digit_Display.h and Four_Digit_Display.c files (you will need both). When you select the files and hit okay, you will be prompted with the following:



"Copy files" should be selected by default, but if it isn't, make sure that is what is selected. This will create a duplicate of the file and add it to your project workspace. From here, you can add the line, #include "Four_Digit_Display.h", to the top of your source file with main(). You will now have access to all of the function included within.

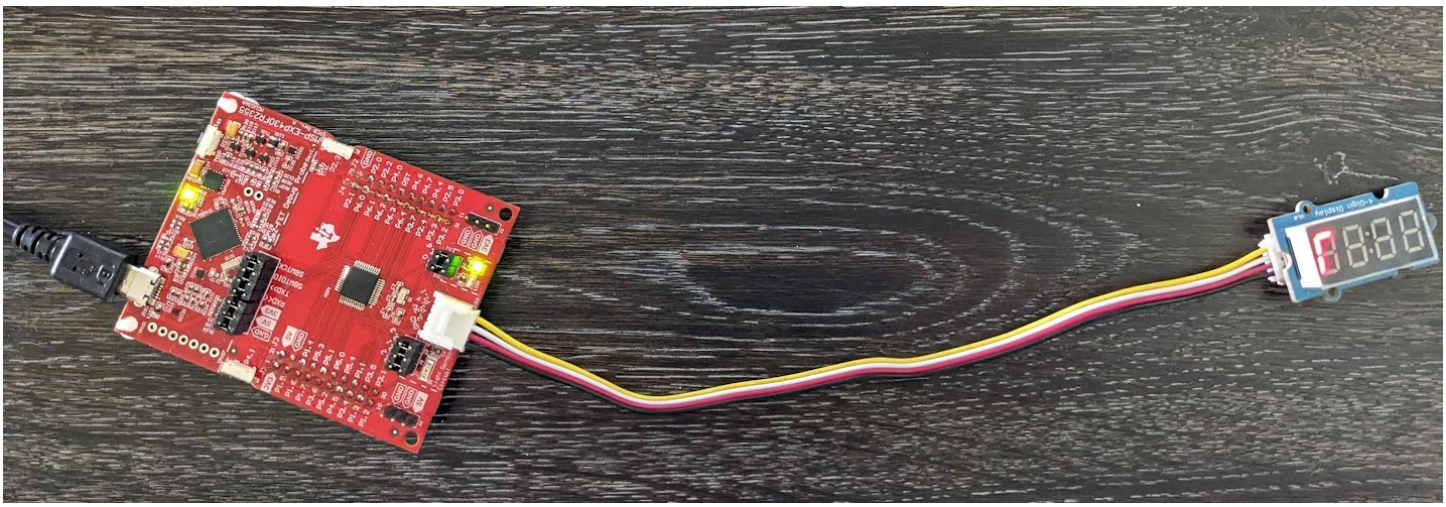
Step 2: Learning to Drive the SSD

The SSD contains four seven segment characters where each segment is its own individual LED that can be turned on and off. To aid with control of this display, an additional device called the "TM1637" is interfaced with it to make control simple. You can see this chip mounted on the bottom layer of the board.

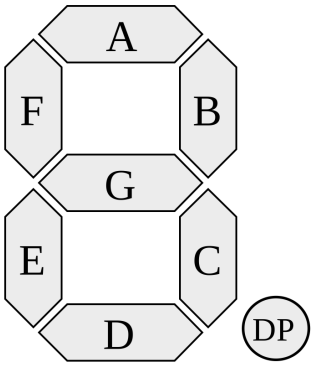


The TM1637 provides a two wire communication interface to set the individual segments of each character. To avoid having to learn and utilize this communication interface, we will be using the library we just imported. This library will provide functions that allow us to target which digit and segment we want to control.

This library assumes that the SSD will be connected to the grove connection on the MSP-EXP430FR2355, as shown below:



Each segment in a seven segment display is identified with a letter, as seen in the image below:



Let's take a look at the functions provided in Four_Digit_Display.h to drive the SSD:

```
// Seven segment display segment mappings
#define SEG_A 0b00000001
#define SEG_B 0b00000010
#define SEG_C 0b00000100
#define SEG_D 0b00001000
#define SEG_E 0b00010000
#define SEG_F 0b00100000
#define SEG_G 0b01000000

// Digit positions
#define POS_1 0
#define POS_2 1
#define POS_3 2
#define POS_4 3

// Pin connections
#define DATA_PIN        4
#define CLK_PIN         1
#define DATA_DIR_REG    P1DIR
#define CLK_DIR_REG     P1DIR
#define DATA_OUT_REG    P1OUT
#define CLK_OUT_REG     P1OUT
#define DATA_IN_REG     P1IN
#define CLK_IN_REG     P1IN

void four_digit_init();                                 // init

void four_digit_set_brightness(enum brightness_t b);        // set before calling display
void four_digit_set_point(bool disp_point);                  // set before calling display
void four_digit_display(uint8_t BitAddr, uint8_t DispData);
void display_segment(uint8_t BitAddr, uint8_t segment);
void four_digit_clear();
```

At the top of the file we can see several define macros. These macros abstract the details of SSD digit positions and segment bit positions. Next we have three interface functions. "four_digit_init()" initializes the associated hardware for controlling the SSD. This function needs to be called before all the other functions. "four_digit_clear()" will clear all written segments to the SSD. "display_segment()" is used to set individual segments on the SSD. An example usage of the write function is as follows:

```
display_segment(POS_1, SEG_F | SEG_E | SEG_D);
```

This function takes two parameters: the first being the digit we want to write to, and the second being the segments we want to illuminate. This will write the letter "L" to the first digit position on the SSD. Illuminating segments F, E, and D form the letter L, which you can see by referencing the segment

diagram above. We set multiple segments by using the SEG_X macros bitwise OR'd together. Individual segments can be written to as well using this function:

```
display_segment(POS_2, SEG_D);
```

which would only write to the bottom segment of the second digit. **Try experimenting with these functions and calling "four_digit_init()" before proceeding to the next step, so you have an adequate understanding of how to control the library.**

Step 3: Creating Interfaces

Now that we can control the SSD, let's go ahead and create some interface functions for our project. We know that we'll need to be setting the red and green LEDs and reading both button presses. Let's go ahead and create some functions for these. These should follow the design of the functions with similar purposes from the previous lab. Use those to model your new functions for writing to the LEDs and reading SW1 and SW2 (don't forget to debounce if you need to!).

The trickier part will be creating a function to "draw" or "render" the train. As mentioned before, the train will be represented by three consecutive segments on the SSD:

Here we can see segment D at positions 4, 5, and 6 are set. The front of the train is at position 4 and the cars being pulled are on positions 5 and 6. When the train is moving forward it will follow the following sequence:


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display


CLK
DIO
GND
VCC
4 Digit Display

And it will continue this pattern making a loop around the outside of the character segments. The train will continue to drive in this loop.

In order to make the process of setting these segments easy, we should track the "position" of the train itself, and just fill in 2 spots behind it for the cars. There are 12 segments where the train can be positioned. Create a function that will take a range of values, such as 1 - 12 or 0 -11, which will represent each "outer" segment. When the function is called, the SSD should be cleared, the "head"/"engine" segment of the train should be set to that location and the two segments "behind" it should be set. Which segment is which position number is arbitrary, and will be up to you to decide. (For example, maybe the bottom left segment is position 1 in your application, or maybe the top right segment is position 1. Then each position "after" it would be 2, then 3, and so on.)

Hint: this is another great spot to test your interface functions. Make sure your LED and button press functions work as expected. Create a simple application to test these before moving on.

Step 4: Diagraming the State Machine

Before writing a bunch of code, we first will need to abstractly describe our state machine. In order to describe the state machine we will need to define the functionality of our application. Let's begin by defining some requirements:

1. The train should be represented by 3 consecutively set segments on the perimeter of the SSD.
2. The train should be stopped on start.
3. Pressing SW1 should cause the train to begin moving. Pressing SW1 again will cause the train to go back to being stoppped.
4. Pressing SW2 should cause the train to change directions.
5. When the train is moving, the green LED should be on and the red LED should be off. When the train is stopped the red LED should be on and the green LED should be off.
6. The train should move at a rate of 1 segment per second.

A requirements list is the first thing to consider when developing a state machine for an embedded system application. We want to construct the state machine such that all the functionality of the application can be represented by it. Sometimes an application may require multiple state machines, fortunately, this one will only require one.

Given these requirements, ask yourself the following questions:

1. How many states do I need to represent the train's behavior?
2. What inputs would cause state changes?
3. What are the outputs at each state?

Considering these questions, draw the diagram of your state machine. We will use this diagram to influence the structure of our code. (Hint: you will need a minimum of two states for this project, but you can define as many as you need. This is your design, and you can make states as you see fit. A good design for this project will likely result in 2-4 states. I would not recommend making a state for each train position - if you are doing this, you are likely doing it wrong.) Also consider that each state might contain functionality that isn't represented by the diagram, such as incrementing the position of the train.

Step 5: Implementing the State Machine

Using one of the techniques described in class (switch statement, function table), implement your described state machine in your code. Before creating your state functions, you will need to create some custom data types for passing to and from the state functions.

Begin by creating a custom enumeration data type to represent your states. Once that is complete, create a custom struct data type to hold your state inputs (recall from Step 4 what your inputs to your system are. Anything that would cause a state transition could be considered an interrupt. Sometimes even a variable that gets passed between states might be considered an input.).

Functionalize each of your states. Your state functions should follow the structure where they return the next state and are passed the inputs structure. Within the body of your state functions you should be setting any appropriate outputs, updating variables, and computing the next state. Note that if you want any changes to be permanent within the structure you pass to your state function, you will need to pass a pointer to that state function.

An example of a state function may follow a structure like below:

```
state_t exampleState(inputs_t* inputs){
   state_t nextState;
   // set outputs
   setRedLed(true);
   // compute next state
   if(inputs->val > 10)
       nextState = EXAMPLE_STATE_2;
   else
```

```
        nextState = EXAMPLE_STATE_1;
    return nextState;
}
```

Once your state functions are complete, you might want to take a moment here and test each one of them. Make a test program that just calls one single state, and ensure that state does what you intend it to do. Be sure to verify that the next state being returned is also correct for the given inputs.

Now that the state functions and custom data types have been defined, we can implement the actual state machine. You may use either the switch statement method or the state table method discussed in class. See the notes and example code for implementation details.

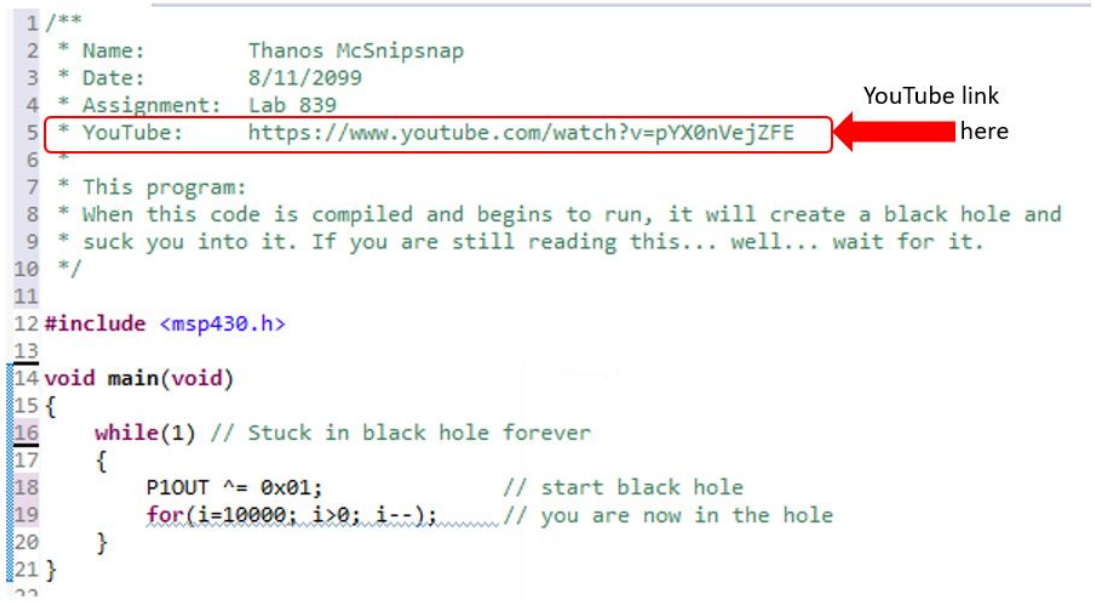
Step 6: Verify Functionality

Now is the time for the final test! Run your state machine and verify that it behaves according to your state diagram. Test all conditions.

How to Submit:

Take a video of your code working on your board. In your video, be sure to clearly demonstrate and explain all requirements `(which are listed in the grading rubric below)` of your working code.

Upload your video to **YouTube only** (do not use google drive or other platforms) and then provide a link within your code submission as a comment like the picture shows below. Remember to include your name, program details, etc., within the program header at the top. It is common coding practice to provide a program header, so make sure you do this. If you used an example template, remember to delete the previous program header and replace it with your own.




```
/**
 * Name:        Thanos McSnipsnap
 * Date:        8/11/2099
 * Assignment:  Lab 839
 * YouTube:     https://www.youtube.com/watch?v=pYX0nVejZFE
 *
 * This program:
 * When this code is compiled and begins to run, it will create a black hole and
 * suck you into it. If you are still reading this... well... wait for it.
 */

#include <msp430.h>

void main(void)
{
    while(1) // Stuck in black hole forever
    {
        P1OUT ^= 0x01;              // start black hole
        for(i=10000; i>0; i--);     // you are now in the hole
    }
}
```

Copy your lab code from CCS into a `text file (.txt)` using notepad, Word, etc. Save your text file as `lastname_firstname_lab#.txt` with `#` being the current lab you are submitting.

Submit this text file (.txt) to Canvas.

No other submission files will be accepted. You will only have two attempts to submit during the submission deadline. If a second attempt is needed, make sure all the necessary changes are made before submitting, as this would be your final attempt.

Lab 3 Rubric

Criteria	Ratings		Pts
Code is neatly formatted	**10 pts Full Marks**	**0 pts No Marks**	10 pts
The train should be represented by 3 consecutively set segments on the perimeter of the LCD	**10 pts Full Marks**	**0 pts No Marks**	10 pts
The train should be stopped on start.	**10 pts Full Marks**	**0 pts No Marks**	10 pts
Pressing SW1 should cause the train to begin moving. Pressing SW1 again will cause the train to go back to being stoppped.	**10 pts Full Marks**	**0 pts No Marks**	10 pts
Pressing SW2 should cause the train to change directions.	**10 pts Full Marks**	**0 pts No Marks**	10 pts
When the train is moving, the green LED should be on and the red LED should be off. When the train is stopped the red LED should be on and the green LED should be off.	**10 pts Full Marks**	**0 pts No Marks**	10 pts
The train should move at a rate of 1 segment per second.	**10 pts Full Marks**	**0 pts No Marks**	10 pts
States are properly functionalized in the code	**10 pts Full Marks**	**0 pts No Marks**	10 pts
State are represented with enumerated values	**10 pts Full Marks**	**0 pts No Marks**	10 pts

Criteria	Ratings		Pts
State machine is represented by a switch statement or function table	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
			Total Points: 100