EE 226 –MICROPROCESSORS
LABORATORY MANUAL –4 –
TRAFFIC LIGHT CONTROLLER

Design Overview

Consider a 2 street intersection as shown below. There are two one-way streets, labeled ***South*** and ***West*** for southbound and westbound cars to travel on.

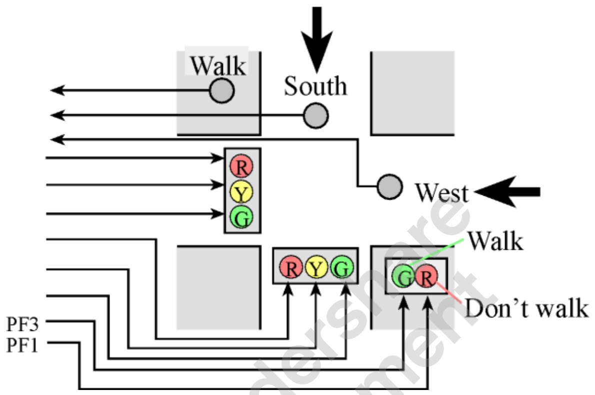


Figure 5.1. Traffic Light Intersection (3 inputs and 8 outputs).

- **INPUTS - 3**
  1. ***South Sensor:*** This sensor or button be activated (set to logic 1), if one or more cars are near the intersection on the South road. You will simulate this on the hardware by interfacing a button and pressing it down.
  2. ***West Sensor:*** Same as south sensor but for westbound traffic.
  3. ***Walk Button:*** Will be pushed by a pedestrian when he or she wishes to cross in any direction. You will simulate this effect by pushing a button.
- **OUTPUTS - 8**

  You will use 8 outputs from your microcomputer that control the traffic lights.

  1. ***(6 LEDS) South & West, R/Y/G Lights***: You will have to interface a total of 6 total LED's for the South and West's red, yellow and green lights.

  2. ***(1 LED) Walk light:*** This will be the green LED (PF3) on the LaunchPad, and will be turned on when pedestrians are allowed to cross. To request a walk, a pedestrian must push and hold the walk button for at least 2 seconds, after which person can release the button, and the walk request should be serviced eventually. If the user does not hold down the button for 2 seconds this document does not specify what will happen, you may either go to the walking states or not. We leave this freedom up to the engineers designing the FSM.

The walk sequence should be realistic, showing three separate conditions:

1. ***Walk:*** Your walk light should be on signifying the pedestrians may cross.
2. ***Warning***: Flash your don't walk LED signifying that pedestrians need to hurry up. You may decide how long you want to flash it.
3. ***Don't Walk:*** Your don't walk LED should be on and constant.

3. ***(1 LED) Don't Walk light:*** This will be the red LED (PF 1) on the LaunchPad.

   a. When the "don't walk" condition flashes (and the two traffic signals are red), pedestrians should hurry up and finish crossing in any direction.
   b. When the "don't walk" condition is on steady, pedestrians should not enter the intersection.

   All other inputs and outputs must be built on the breadboard.

Procedure

The basic approach to this lab will be to first develop and debug your system interface with actual lights and switches on a physical TM4C123. You are encouraged to adjust the timing so that the operation of your machine is convenient for you to debug and for the TA to observe during demonstration.

Part a - Pin/Port Selection

Decide which port pins you will use for the inputs and outputs. Avoid the pins with hardware already connected. Table 4.4 in the book lists the pins to avoid. Which ports are available for the lights and switches; these choices are listed in Tables 5.1 and 5.2. The "don't walk" and "walk" lights must be PF1 and PF3 respectively, but you have some flexibility when deciding where to attach the remaining signals. In particular, Table 5.1 shows you three possibilities for how you can connect the six LEDs that form the traffic lights. Table 5.2 shows you three possibilities for how you can connect the three positive logic switches that constitute the input sensors. Obviously, you will not connect both inputs and outputs to the same pin. Please note that the possibilities listed in Tables 5.1 and 5.2 are not the only possibilities.

Stoplight Signal	Possibility 1	Possibility 2	Possibility 3
Red south	PA7	PB5	PE5
Yellow south	PA6	PB4	PE4
Green south	PA5	PB3	PE3
Red west	PA4	PB2	PE2
Yellow west	PA3	PB1	PE1
Green west	PA2	PB0	PE0

Table 5.1. Possible ports to interface the traffic lights (PF1=red don't walk, PF3=green walk).

Stoplight Sensor	Possibility 1	Possibility 2	Possibility 3
Walk sensor	PA4	PB2	PE2
South sensor	PA3	PB1	PE1
West sensor	PA2	PB0	PE0

Table 5.2. Possible ports to interface the sensors.

If you are using PD0, PD1, PB7, PB6, PB1 or PB0, make sure R9, R10, R25, and R29 are removed from your LaunchPad. R25 and R29 are not on the older LM4F120 LaunchPads, just the new TM4C123 LaunchPads. The TM4C123 LaunchPads I bought did not have R25 and R29 soldered on, so I just had to remove R9 and R10. The R9 R10 jumpers are only needed for some MSP430 booster packs running on the TM4C123, so there is little chance you will ever need R9 and R10.

Part b - FSM Design

Design a finite state machine that implements a traffic light system. Include a picture of your finite state machine in the deliverables showing the various states, inputs, outputs, wait times and transitions. It is advisable that you come to the TAs to verify you FSM design before continuing to code.

It may be helpful to look at the **Civil Engineering** and also the **Tips and Tricks** sections below for guidance.

Part c - Construct and Test Circuit

The first step is to interface three push button switches for the sensors. You should implement positive logic switches. *Do not place or remove wires on the breadboard while the power is on.* Build the switch circuits and test the voltages using a voltmeter. You can also use the debugger to observe the input pin to verify the proper operation of the interface.

(Optional) The next step is to build six LED output circuits. Build the system physically in a shape that matches a traffic intersection, so the TA can better visualize the operation of the system. Look up the pin assignments in the 7406 data sheet. Be sure to connect +5V (labeled VBUS on the Launchpad) power to pin 14 and ground to pin 7. Write a simple main program to test the LED interface. Set the outputs high and low, and measure the following three voltages: ground to the input to 7406, ground to the output from 7406 which is the LED cathode voltage, and ground to the LED anode voltage.

Part e - Debug on Real Hardware

Debug your combined hardware/software system on the actual TM4C123 board.

Demonstration

During checkout, you will be asked to show both the simulated and actual TM4C123 systems to the TA. The TAs will expect you to know how the **`SysTick_Wait`** function works, and know how to add more input signals and/or output signals. An interesting question that may be asked during checkout is how you could experimentally prove your system works. In other words, what data should be collected and how would you collect it? If there were an accident, could you theoretically prove to the judge and jury that your software implements the FSM? What type of FSM do you have? What other types are there? How many states does it have? In general, how many next-state arrows are there? Explain how the linked data structure is used to implement the FSM. Explain the mathematical equation used to calculate the address of the next state, depending on the current state and the input. Be prepared to write software that delays 1 second without using the timer (you can use a calculator and manual). How do you prove the delay will be 1 second? What does it mean for the C compiler to align objects in memory? Why does the compiler perform alignment? List some general qualities that would characterize a good FSM.

Civil Engineering Questions

There are many civil engineering questions that students ask. How you choose to answer these questions will determine how good a civil engineer you are, but will not affect your grade on this lab. For each question, there are many possible answers, and you are free to choose how you want to answer it. It is reasonable however for the TA to ask how you would have implemented other answers to these civil engineering questions using the same FSM structure.

1. How long should I wait in each state? *Possible answer*: 1 to 2 seconds of real TA time.
2. What happens if I push 2 or 3 buttons at a time? *Possible answer*: cycle through the requests servicing them in a round robin fashion (service one, then another)
3. What if I push the walk button, but release it before 2 seconds are up? *Possible answer*: service it or ignore it depending on exactly when it occurred.
4. What if I push a car button, but release it before it is serviced? *Possible answer*: ignore the request as if it never happened (e.g., car came to a red light, came to a full stop, and then made a legal turn). *Possible answer*: service the request or ignore it depending on when it occurred.
5. Assume there are no cars and the light is green on the North, what if a car now comes on the East? Do I have to recognize a new input right away or wait until the end of the wait time? *Possible answer:* no, just wait until the end of the current wait, then service it. *Possible answer*: yes; break states with long waits into multiple states with same output but shorter waits.
6. What if the walk button is pushed while the don't walk light is flashing? *Possible answer*: ignore it, go to a green light state and if the walk button is still pushed, then go to walk state again. *Possible answer:* if no cars are waiting, go back to the walk state. *Possible answer*: remember that the button was pushed, and go to a walk state after the next green light state.
7. Does the walk occur on just one street or both? *Possible answer*: stop all cars and let people walk across either or both streets.
8. How do I signify a walk condition? *Answer*: You must use the on board green LED for walk, and on board red LED as the don't walk.

Figure 5.3. Kızılay walk signified by red and yellow (phased out in 2018) [1].

In real products that we market to consumers, we put the executable instructions and the finite state machine linked data structure into the nonvolatile memory such as Flash. A good implementation will allow minor changes to the finite machine (adding states, modifying times, removing states, moving transition arrows, changing the initial state) simply by changing the linked data structure, without changing the executable instructions. Making changes to executable code requires you to debug/verify the system again. If there is a 1-1 mapping from FSM to linked data structure, then if we just change the state graph and follow the 1-1 mapping, we can be confident our new system operates the new FSM properly. Obviously, if we add another input sensor or output light, it may be necessary to update the executable part of the software, re-assemble or re-compile and retest the system.

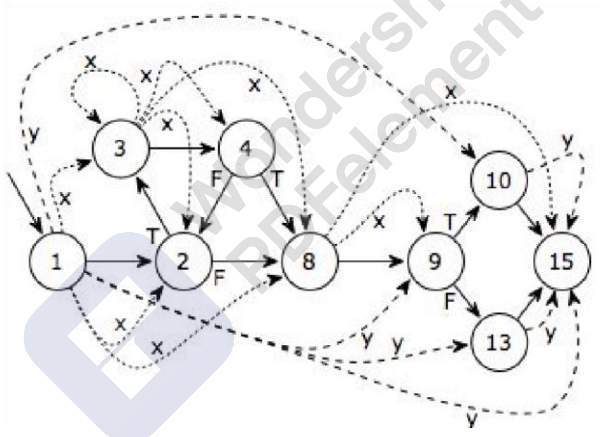
Tips and Tricks Section:

- When a car sensor is released or deactivated (set to logic 0), it means no cars are waiting to enter the intersection. I.E. When you are not pressing the button no cars are on the road.
- You should exercise common sense when assigning the length of time that the traffic light will spend in each state; so that the system changes at a speed convenient for the TA (stuff changes fast enough so the TA doesn't get bored, but not so fast that the TA can't see what is happening).
- There is no single, "best" way to implement your system. However, your scheme must use a linked data structure stored in ROM. There should be a 1-1 mapping from the FSM states and the linked elements. An example solution will have about 10 to 30 states in the finite state machine, and provides for input dependence.
- Try not to focus on the civil engineering issues. I.e., the machine does not have to maximize traffic flow or minimize waiting. On the other hand, if there are multiple requests, the system should cycle through, servicing them all. Build a quality computer engineering solution that is easy to understand and easy to change.

Handling the 2 Second Walk

- One way to handle the 2-second walk button requirement is to have a simplified state graph centered around a center home or check all state. The idea is that you have one state that you can always come back to that controls what your state graph is doing. Initially when thinking about this problem you can identify that a servicing traffic on the east or west road or servicing the pedestrians can be thought of a set sequence of events that have to start and end somewhere. So just make them start and end at the check all state. This approach may make sense to some of you or there is another approach mentioned next.
- Another way to handle the 2-second walk button requirement is to add a duplicate set of states. The first set of states means the walk button is not pressed. The second set of states means the walk operation is requested. Go from the first set to the second set whenever the walk is pushed. Go from the second back to the first whenever the walk condition is output. The two sets of states allow you to remember that a walk has been requested; in this way the request is serviced when appropriate.

Aside: If your FSM starts to look like the image below, you may want to go to a TA for advice on cleaning it up. A ***Clear State Graph*** *should be observed for the grading requirements.*



Drawing your FSM

- Because we have three inputs, there will be 8 next state arrows. One way to draw the FSM graph to make it easier to read is to use X to signify don't care. For example, compare the Figure 6.9 in the book to the FSM graph in Figure 5.2 below. Drawing two arrows labeled **01** and **11** is the same as drawing one arrow with the label **X1**. When we implement the data structure, we will expand the shorthand and explicitly list all possible next states.

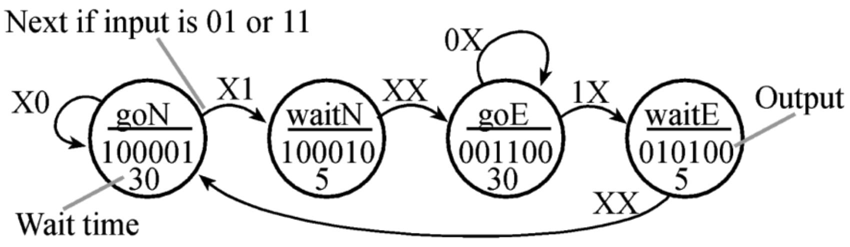


Figure 5.2. FSM from the book Figure 6.20 redrawn with a shorthand format.

References
[1] *https://www.haberankara.com/sehir-plancilarindan-o-kazayla-ilgili-aciklama-isiklar-kosar-adim-gecmeye-zorluyor/69656/, Date of Access: 04.16.2019*
[2] *http://users.ece.utexas.edu/~valvano/Volume1/, Date of Access: 04.16.2019*
[3]https://www.youtube.com/watch?v=NjaMe4s0Zz8
[4]https://www.youtube.com/watch?v=kgABPjf9qLI&ab_channel=JonathanValvano