Embedded Systems and Applications

Traffic Light Controller – Lab 10

Using SysTick Timer, PLL, FSMs

Preparation

You will need a LaunchPad, a three switches, three 10kΩ resistors, six LEDs, and six 470Ω resistors. Run the Lab10 starter file in the simulator and on the real board just to make sure the configurations are correct. Download the data sheet for the LED **HLMP-4700.pdf**. Read the data sheet to look up the typical forward voltage and current for the three colored LED. Think about the operating voltage and current that will occur when interfaced with the 470 Ω resistor.

Getting past the grader

Lab 10 was the most difficult grader for us to write in this class (John Valvano, Ramesh Yerraballi). The biggest challenge for us is that the correct answer not only depends on the current input, but also on the time and values of previous outputs. A few suggestions on things that do not matter, but will make it easier for the grader to see your solution in a better light. 1) When outputting to two ports, output to the car LEDs first, and output to the Port F (walk, don't walk) second; 2) When outputting to a port, do not write intermediate values; 3) Do not bother being friendly, it just confuses the grader. For example, this code confuses the grader because it looks like you turned off all the lights.

```
GPIO_PORTF_DATA_R &= ~0x0A;
GPIO_PORTF_DATA_R |= FSM[State].PFOut;
```

Rather, we suggest that you simply output the new data to the ports in this order, assuming the LEDs are on Port B

```
GPIO_PORTB_DATA_R = FSM[State].PBOut;
GPIO_PORTF_DATA_R = FSM[State].PFOut;
```

Starter project

Lab10_TrafficLight

Also, Example Programs are available in the TExasWare Folder.

Example YouTube Videos

(1) https://www.youtube.com/watch?v=NjaMe4s0Zz8
(2) https://www.youtube.com/watch?v=kgABPjf9qLI
(3) Example Programs are available in the TExasWare Folder.

Purpose

This lab has these major objectives: 1) the understanding and implementing of indexed data structures; 2) learning how to create a segmented software system; and 3) the study of real-time synchronization by designing a finite state machine controller. Software skills you will learn include advanced indexed addressing, linked data structures, creating fixed-time delays using the SysTick timer, and debugging real-time systems. Please read the entire lab before starting.

System Requirements

Consider a 4-corner intersection as shown in Figure 10.1. There are two one-way streets, labeled South (cars travel South) and West (cars travel West). There are three inputs to your LaunchPad, two are car sensors, and one is a pedestrian sensor. The South car sensor will be true (3.3V) if one or more cars are near the intersection on the South road. Similarly, the West car sensor will be true (3.3V) if one or more cars are near the intersection on the West road. The Walk sensor will be true (3.3V) if a pedestrian is present and he or she wishes to cross in any direction. This walk sensor is different from a walk button on most real intersections. This means when you are testing the system, you must push and hold the walk sensor until the FSM recognizes the presence of the pedestrian. Similarly, you will have to push and hold the car sensor until the FSM recognizes the presence of the car. In this simple system, if the walk sensor is +3.3V, there is pedestrian to service, and if the walk sensor is 0V, there are no people who wish to walk. The walk sensor and walk light will service pedestrians who wish to cross in any direction. This means the roads must both be red before the walk light is activated. In a similar fashion, when a car sensor is 0V, it means no cars are waiting to enter the intersection. The don't walk light should be on while cars have a green or yellow light.

You will interface 6 LEDs that represent the two Red-Yellow-Green traffic lights, and you will use the PF3 green LED for the “walk” light and the PF1 red LED for the “don’t walk” light. When the “walk” condition is signified, pedestrians are allowed to cross in any direction. When the “don’t walk” light flashes (and the two traffic signals are red), pedestrians should hurry up and finish crossing. When the “don’t walk” condition is on steady, pedestrians should not enter the intersection.

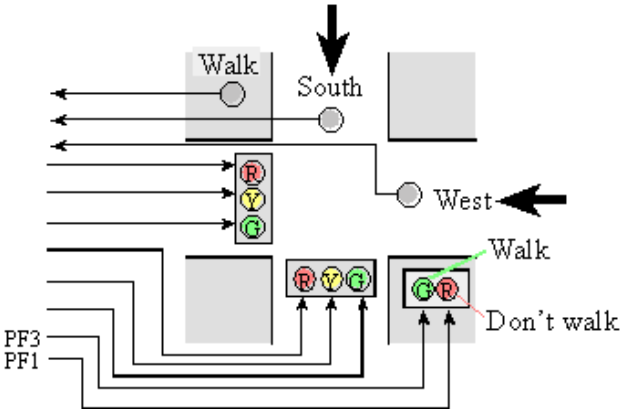

Figure 10.1. Traffic Light Intersection.

Your solution **should not need to be friendly**. In other words you should assume the FSM controls all bits in each of the ports it uses.

Traffic should not be allowed to crash. i.e., there should not be a green or yellow on one road at the same time there is a green or yellow LED on the other road. In other words, while traffic is flowing in one direction, there should be a red light in the other direction. You should exercise common sense when assigning the length of time that the traffic light will spend in each state; so that the grading engine can complete the testing in a reasonable amount of time. Each traffic light pattern must be on for at least ½ second but for at most 5 seconds. During simulation grading, we suggest you make all times between ½ and 1 second. Cars should not be allowed to hit the pedestrians; both reds on the roads should be on whenever the walk light is on. The walk sequence should be realistic, showing three separate conditions: 1) "walk", 2) "hurry up" using a flashing LED, and 3) "don't walk". You may assume the three sensors remain active for as long as service is required. The "hurry up" flashing should occur at least twice but at most four times.

The **automatic grader** checks for function. In other words the grader sets the three inputs, and then checks to see of the output pattern is appropriate. In particular, the grader performs these checks:

0) At all times, there should be exactly one of the {red, yellow, green} traffic lights active on the south road. At all times, there should be exactly one of the {red, yellow, green} traffic lights active on the west road. To switch a light from green to red it must sequence green, yellow then red. The grader checks for the proper sequence of output patterns but does not measure the time the FSM spends in each state. The "walk" and "don't walk" lights should never both be on at the same time.

1) Do not allow cars to crash into each other. This means there can never be a green or yellow on one road at the same time as a green or yellow on the other road. Engineers do not want people to get hurt.

2) Do not allow pedestrians to walk while any cars are allowed to go. This means there can never be a green or yellow on either road at the same time as a "walk" light. Furthermore, there can never be a green or yellow on either road at the same time as the "don't walk" light is flashing. If a green

light is active on one of the roads, the “don’t walk” should be solid red. Engineers do not want people to get hurt.

3) If just the south sensor is active (no walk and no west sensor), the lights should adjust so the south has a green light within 5 seconds (I know this value is unrealistically short, but it makes the grading faster). The south light should stay green for as long as just the south sensor is active.

4) If just the west sensor is active (no walk and no south sensor), the lights should adjust so the west has a green light within 5 seconds. The west light should stay green for as long as just the west sensor is active.

5) If just the walk sensor is active (no west and no south sensor), the lights should adjust so the “walk” light is green within 5 seconds. The “walk” light should stay green for as long as just the walk sensor is active.

6) If all three sensors are active, the lights should go into a circular pattern in any order with the west light green, the south light green, and the “walk” light is green. Of course, the road lights must sequence green-yellow-red each time.

The grading engine can only check for function, not for the quality of your software. This section describes, in qualitative manner, what we think is good design. There is no single, “best” way to implement your system. A “good” solution will have about 9 to 30 states in the finite state machine, and provides for input dependence. Try not to focus on the civil engineering issues. i.e., first build a quality computer engineering solution that is easy to understand and easy to change, and then adjust the state graph so it passes the functional tests of the automatic grader. Because we have three inputs, there will be 8 next state links. 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 two FSM graphs in Figure 10.2. Drawing two arrows labeled **01** and **11** is the same as drawing one arrow with the label **X1**. When we implement the data structure, however, we will expand the shorthand and explicitly list all possible next states.

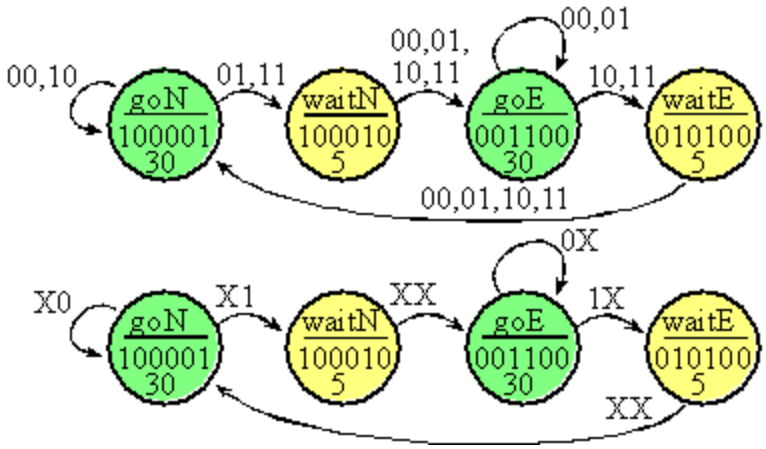

Figure 10.2. FSM from Chapter 10 redrawn with a short hand format.

The following are some qualitative requirements, which we think are important, but for which the **automatic grader** may or may not be able to evaluate.

0) The system provides for input dependence. This means each state has 8 arrows such that the next state depends on the current state and the input. This means you cannot solve the problem by simply cycling through all the states regardless of the input.

1) Because we think being in a state is defined by the output pattern, we think you should implement a Moore and not a Mealy machine. However, your scheme should use a table data structure stored in ROM.

2) There should be a 1-1 mapping between FSM graph and data structure. For a Moore machine, this means each state in the graph has a name, an output, a time to wait, and 8 next state links (one for each input). The data structure has exactly these components: a name, an output, a time to wait, and 8 next state indices (one for each input). There is no more or no less information in the data structure then the information in the state graph.

3) There can be no conditional branches in program, other than the **while** in **SysTick_Wait** and the **for** in **SysTick_Wait10ms**. This will simplify debugging the FSM engine. The main loop of your program should be similar to the traffic FSM described in this chapter.

4) The state graph defines exactly what the system does in a clear and unambiguous fashion. In other words, do not embed functionality (e.g., flash 3 times) into the software that is not explicitly defined in the state graph.

5) Each state has the same format of each state. This means every state has exact one name, one 8-bit output (could be stored as one or two fields in the struct), one time to wait, and 8 next indices.

6) Please use good names and labels (easy to understand and easy to change). Examples of bad state names are **S0** and **S1**.

7) There should be 9 to 30 states with a Moore finite state machine. If your machine has more than 30 states, you have made it more complicated than we had in mind. Usually students with less than 9 states did not flash the “don’t walk” light, or they flashed the lights using a counter. Counters and variables violate the “no conditional branch” requirement.

In real products that we market to consumers, we put the executable instructions and the finite state machine data structure into the nonvolatile memory such as flash EEPROM. 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 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 operate 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 and retest the system.

The grader will activate the PLL so the system runs at 80 MHz, you must not modify this rate.

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 should 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.

0) How long should I wait in each state? *Possible answer*: traffic lights at ½ to 1 seconds of real people time. *Flashing "don't walk"* on for ½ sec, off for a ½ sec and repeat 2 times. If you make the wait time long, the grader may timeout giving you an error.

1) What happens if I push 2 or 3 buttons at a time? *Required operation*: cycle through the requests servicing them in a round robin fashion.

2) What if I push the walk button, but release it before the light turns to walk? *Possible answer*: ignore the request as if it never happened. *Possible answer*: service it or ignore it depending on exactly when it occurred.

3) 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.

4) Assume there are no cars and the light is green on the South, what if a car now comes on the West? 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.

5) 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.

6) Does the walk occur on just one street or both? *Required operation*: stop all cars and let people walk across either or both streets. A green (or yellow) light in any direction while the "walk" light is on will cause the automatic grader to penalize you for failing check #2. The pedestrian sensor does not know which street the pedestrian(s) want to cross, so you must direct all cars to stop while pedestrians may be in the road. You are not allowed to add additional pedestrian sensor because the automatic grader is built to handle only the configuration shown in Figure 10.1.

Let's walk through the steps of building a **state transition table**. A state transition table has exactly the same information as the state transition graph, but in tabular form. Let's begin with the format of the table. There will be a column for the state number; we will number the states sequentially from 0. We will skip the state name for now, and come back to it later. The next two columns will define the output patterns for six traffic lights and two walk lights. In this table we specified the output for the 6 LEDs as "West Green", but once we translate it to software we will replace "West Green" with the explicit value to output, e.g., 0x0C . Often we will need to have multiple states with the same output value, but for now we will simply add a state for each possible output pattern, and later we will add addition rows as needed. The next column is the time to wait with this output. The last eight columns will be the next states for each possible input pattern. Recall the FSM controller will 1) output, 2) wait, 3) input, and 4) change to the next state depending on the current state and the input. So notice the columns of this table are given in this 1,2,4 order.

Num	Name	6-LED	PF3,PF1	Time	In=0	In=1	In=2	In=3	In=4	In=5	In=6	In=7
0	GoWest	West green, South red	Red									
1		West yellow, South red	Red									
2		South green, West red	Red									
3		South yellow, West red	Red									
4		Both roads red	Green									
5		Both roads red	Red									
6		All roads red	off									

Note: to solve this lab you must add more rows. For example, flashing the don't walk 3 times will require 6 states.

Note: to solve this lab you must add more rows. For example, flashing the don't walk 3 times will require 6 states.

Each state should have a descriptive name. These names will be labels in the C code, so use no spaces or special characters. The name should be short. Most importantly the state name should describe *the state of the system, or what you believe to the true*. For example, **GoWest** could mean cars can go on the west road. Make the times short for this lab, neither grader checks the times. It can be 500ms for both simulation and the real board. Next, fill what to do for each input. The **in=0** field is easy (0 means no cars or people). For example, what should you do if the west road is green and there are no cars or people? Another easy one is **in=7** (busy with cars and people). For **in=7**, we need to cycle through west, south and walk: {west green, west yellow, south green, south yellow, walk, don't walk, both walk lights off, don't walk, both walk lights off}. To make the don't walk light flash twice, we will need to add two more states 7,8 (rows in the table). State 7 will have the road outputs red and the don't walk. State 8 will have the road outputs red and the walk LEDs off. In summary, the **in=7** input pattern will cycle through states 0,1,2,3,4,5,6,7,8 over and over again.

Once you have this table, you convert it to C code. This is the structure from Program 10.4

```
struct State {
  unsigned long Out;
  unsigned long Time;
  unsigned long Next[4];};
```

Modify this structure adding a line for Outputs to PF3, PF1. Change the 4 to 8, because with 3 inputs, each state needs 8 possible next states. Convert the above state table into the FSM structure. The engine will execute:

1) Output 6-LEDs in an unfriendly manner
2) Output PF3, PF1 in an unfriendly manner
3) Wait time
4) Input from sensors
5) Go to next state

This C code will be similar to the Traffic FSM example in the Program 10.4. The grader does not check for friendly code, so you are allowed to write to the entire port while updating the LEDs.