**《嵌入式系统及应用》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2019级计算机科学与技术卓越** | | | **姓名** | **李燕琴 李放** |
| **实验题目** | **Traffic Light Controller** | | | | | |
| **实验时间** | **2022年5月15日** | | **实验地点** | **DS3 305** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 ■综合性** | | |
| 教师评价：  ■算法/实验过程正确； ■源程序/实验内容提交 ■程序结构/实验步骤合理；  ■实验结果正确； ■语法、语义正确； ■报告规范；  其他：  评价教师签名： | | | | | | |
| 1. 实验目的 2. This lab has these major objectives: 1) the understanding and implementing of indexed data 3. structures; 2) learning how to create a segmented software system; and 3) the study of real- 4. time synchronization by designing a finite state machine controller. Software skills you will 5. learn include advanced indexed addressing, linked data structures, creating fixed-time delays 6. using the SysTick timer, and debugging real-time systems. Please read the entire lab before 7. starting. | | | | | | |
| 1. 实验项目内容   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. In a similar fashion, when a car sensor is 0V, it means no cars are waiting to enter the intersection. 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. 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.*  Traffic should not be allowed to crash. I.e., there should not be only a green or only a yellow LED on one road at the same time there is only a green or only a yellow LED on the other road. 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. Cars should not be allowed to hit the pedestrians. 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 4 times.  The automatic grader can only check for function (does a pattern of inputs, result in the correct outputs.) 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 be yellow for at least ½ sec. 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 pattern such within one 20-second period that the west light is green for at least 1 second, the south light is green for at least 1 second and the “walk” light is green for at least 1 second.  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 linked 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 pointers (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 make the FSM engine trivial.  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.  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 1 to 2 seconds of real people time. *Flashing “don’t walk”* on for ½ sec, off for a ½ sec and repeat 3 times.  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 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.  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 10.13.  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 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 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 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.  As with all graders, it begins by checking initialization registers. During the I/O portion of the grading, we get a notification whenever you write to either of the output ports. The grader checks for valid output pattern sequences. We have defined 9valid output patterns, listed below. For each valid output pattern, there are a only finite number of valid output patterns that could be next.  Pattern 1) **All lights are red.** Once the output is at this pattern, the valid next patterns are {1,2,4,6,7,8,9}  Pattern 2) **West is green**, south is red, don’t walk is red**.** Once the output is at this pattern, the valid next patterns are {2,3}  Pattern 3) **West is yellow**, south is red, don’t walk is red**.** Once the output is at this pattern, the valid next patterns are {1,3,4,6}  Pattern 4) **South is green**, west is red, don’t walk is red**.** Once the output is at this pattern, the valid next patterns are {4,5}  Pattern 5) **South is yellow**, west is red, don’t walk is red**.** Once the output is at this pattern, the valid next patterns are {1,2,5,6}  Pattern 6) **Walk is green**, south is red, west is red**.** Once the output is at this pattern, the valid next patterns are {1,6,7}  Pattern 7) **Don’t Walk is off**, south is red, west is red**.** Once the output is at this pattern, the valid next patterns are {1,2,4,6,7,8,9} Pattern 8) **Don’t Walk is off**, west is red, **south is green.** Once the output is at this pattern, the valid next patterns are {4,5}  Pattern 9) **Don’t Walk is off**, **west is green**, south is red**.** Once the output is at this pattern, the valid next patterns are {2,3}  There are a couple of consequences of this grading algorithm:  1) You should output to the road lights first and then to the walk lights, 2) You should output to the walk and don't walk lights at the same time 3) For simulation, we do not check for timing, so make the delays short during simulation testing. | | | | | | |
| 1. 实验过程或算法（源程序） 2. 定义了结构体struct，来表示每一个状态下的车辆红绿灯变化情况和行人红绿灯变化情况。     其中vehicle表示车辆红绿灯变化情况，pedestrain表示行人红绿灯变化情况，ms表示每一个状态持续的时间。   1. 根据指导书定义红绿灯可能存在的所有状态。        1. 实验电路连线图        1. 为了代码实现起来更为方便，使用了如下的trick     将比较难记的PF,PB等通过define取一个方便记忆的名字，为后面编写代码提供了不小的便利。   1. 实验代码 2. // \*\*\*\*\* 0. Documentation Section \*\*\*\*\* 3. // TableTrafficLight.c for Lab 10 4. // Runs on LM4F120/TM4C123 5. // Index implementation of a Moore finite state machine to operate a traffic light. 6. // Daniel Valvano, Jonathan Valvano 7. // November 7, 2013 9. // east/west red light connected to PB5 10. // east/west yellow light connected to PB4 11. // east/west green light connected to PB3 12. // north/south facing red light connected to PB2 13. // north/south facing yellow light connected to PB1 14. // north/south facing green light connected to PB0 15. // pedestrian detector connected to PE2 (1=pedestrian present) 16. // north/south car detector connected to PE1 (1=car present) 17. // east/west car detector connected to PE0 (1=car present) 18. // "walk" light connected to PF3 (built-in green LED) 19. // "don't walk" light connected to PF1 (built-in red LED) 21. // \*\*\*\*\* 1. Pre-processor Directives Section \*\*\*\*\* 22. #include "TExaS.h" 23. #include "tm4c123gh6pm.h" 24. #define VEHICLE      (\*((volatile unsigned long \*)0x400043FC))   //PB 25. #define PEDESTRAIN (\*((volatile unsigned long \*)0x400253FC))   //PF 26. #define SENSORS      (\*((volatile unsigned long \*)0x400243FC))   //PE 28. // \*\*\*\*\* 2. Global Declarations Section \*\*\*\*\* 30. // FUNCTION PROTOTYPES: Each subroutine defined 31. **void** DisableInterrupts(**void**); // Disable interrupts 32. **void** EnableInterrupts(**void**);  // Enable interrupts 33. **typedef** **struct** State\_tag{ 34. unsigned **long** vehicle; 35. unsigned **long** pedestrain; 36. unsigned **long** ms; 37. }State; 38. State states[9] = 39. { 40. {0x24, 0x2, 1}, // S0:all red 41. {0x0C, 0x2, 1}, // S1:west is green, other red 42. {0x14, 0x2, 1}, // S2:west is yellow, other red 43. {0x21, 0x2, 1}, // S3:south is green, other red 44. {0x22, 0x2, 1}, // S4:south is yellow, other red 45. {0x24, 0x8, 1}, // S5:walk is green, other red 46. {0x24, 0x0, 1}, // S6:don't walk off, other red 47. {0x21, 0x0, 1}, // S7:don't walk off, south green, other red 48. {0x0C, 0x0, 1}  // S8:don't walk off, west green, other red 49. }; 50. **void** SysTick\_Wait(unsigned **long** delay){ 51. NVIC\_ST\_RELOAD\_R = delay-1;  // number of counts to wait 52. NVIC\_ST\_CURRENT\_R = 0;       // any value written to CURRENT clears 53. **while**((NVIC\_ST\_CTRL\_R&0x00010000)==0){ // wait for count flag 54. } 55. } 57. **void** SysTick\_Wait10ms(unsigned **long** delay){ 58. unsigned **long** i; 59. **for**(i=0; i<delay; i++){ 60. SysTick\_Wait(800000);  // wait 10ms 61. } 62. } 64. **void** PortE\_Init(**void**){ 65. **volatile** unsigned **long** delay; 66. SYSCTL\_RCGC2\_R |= 0x00000010;     // 1) activate clock for Port F 67. delay = SYSCTL\_RCGC2\_R;           // allow time for clock to start 68. GPIO\_PORTE\_LOCK\_R = 0x4C4F434B;   // 2) unlock GPIO Port F 69. GPIO\_PORTE\_CR\_R = 0x07;           // allow changes to PF4-0 70. // only PF0 needs to be unlocked, other bits can't be locked 71. GPIO\_PORTE\_AMSEL\_R = 0x00;        // 3) disable analog on PF 72. GPIO\_PORTE\_PCTL\_R = 0x00000000;   // 4) PCTL GPIO on PF4-0 73. GPIO\_PORTE\_DIR\_R &= !0x07;          // 5) PF4,PF0 in, PF3-1 out 74. GPIO\_PORTE\_AFSEL\_R = 0x00;        // 6) disable alt funct on PF7-0 75. //GPIO\_PORTE\_PUR\_R  = 0x11;          // enable pull-up on PF0 and PF4 76. GPIO\_PORTE\_DEN\_R |= 0x07;          // 7) enable digital I/O on PF4-0 77. } 78. **void** PortB\_Init(**void**){ 79. **volatile** unsigned **long** delay; 80. SYSCTL\_RCGC2\_R    |= 0x00000002; 81. delay                        = SYSCTL\_RCGC2\_R; 82. GPIO\_PORTB\_LOCK\_R  = 0x4C4F434B; 83. GPIO\_PORTB\_CR\_R     |= 0x3F; 84. GPIO\_PORTB\_AMSEL\_R = 0x00; 85. GPIO\_PORTB\_PCTL\_R  = 0x00; 86. GPIO\_PORTB\_DIR\_R    |= 0x3F; 87. GPIO\_PORTB\_AFSEL\_R = 0x00; 88. GPIO\_PORTB\_DEN\_R    |= 0x3F; 89. } 90. **void** PortF\_Init(**void**){ **volatile** unsigned **long** delay; 91. SYSCTL\_RCGC2\_R |= 0x00000020;     // 1) activate clock for Port F 92. delay = SYSCTL\_RCGC2\_R;           // allow time for clock to start 93. GPIO\_PORTF\_LOCK\_R = 0x4C4F434B;   // 2) unlock GPIO Port F 94. GPIO\_PORTF\_CR\_R = 0x1F;           // allow changes to PF4-0 95. // only PF0 needs to be unlocked, other bits can't be locked 96. GPIO\_PORTF\_AMSEL\_R = 0x00;        // 3) disable analog on PF 97. GPIO\_PORTF\_PCTL\_R = 0x00000000;   // 4) PCTL GPIO on PF4-0 98. GPIO\_PORTF\_DIR\_R = 0x0A;          // 5) PF4,PF0 in, PF3-1 out 99. GPIO\_PORTF\_AFSEL\_R = 0x00;        // 6) disable alt funct on PF7-0 100. //GPIO\_PORTF\_PUR\_R = 0x11;          // enable pull-up on PF0 and PF4 101. GPIO\_PORTF\_DEN\_R = 0x1F;          // 7) enable digital I/O on PF4-0 102. } 103. // Initialize SysTick with busy wait running at bus clock. 104. **void** SysTick\_Init(**void**){ 105. NVIC\_ST\_CTRL\_R = 0;                   // disable SysTick during setup 106. NVIC\_ST\_RELOAD\_R = 0x00FFFFFF;        // maximum reload value 107. NVIC\_ST\_CURRENT\_R = 0;                // any write to current clears it 108. NVIC\_ST\_CTRL\_R = 0x00000005;          // enable SysTick with core clock 109. } 110. // \*\*\*\*\* 3. Subroutines Section \*\*\*\*\* 111. **volatile** unsigned **long** counter; 112. **volatile** unsigned **long** cur\_state = 1; 113. **int** main(**void**){ 114. TExaS\_Init(SW\_PIN\_PE210, LED\_PIN\_PB543210); // activate grader and set system clock to 80 MHz 115. PortE\_Init(); 116. PortB\_Init(); 117. PortF\_Init(); 118. SysTick\_Init(); 119. EnableInterrupts(); 120. **while**(1){ 121. **volatile** unsigned **long** sensors; 122. VEHICLE = states[cur\_state].vehicle; 123. PEDESTRAIN = states[cur\_state].pedestrain; 124. SysTick\_Wait10ms(states[cur\_state].ms/10); 125. sensors = SENSORS; 126. **switch**(cur\_state){ 127. **case** 0: 128. **if**(sensors & 0x1){ 129. cur\_state = 1; 130. } 131. **else** **if**(sensors & 0x2){ 132. cur\_state =  3; 133. } 134. **else** { 135. cur\_state = 1; 136. } 137. **break**; 138. **case** 1: 139. **if**(sensors!=0x1){ 140. cur\_state = 2; 141. } 142. **break**; 143. **case** 2: 144. **if**(sensors & 0x2){ 145. cur\_state  = 3; 146. } 147. **else** **if**(sensors & 0x4){ 148. cur\_state = 5; 149. } 150. **else**{ 151. cur\_state = 3; 152. } 153. **break**; 154. **case** 3: 155. **if**(sensors!=0x2){ 156. cur\_state = 4; 157. } 158. **break**; 159. **case** 4: 160. **if**(sensors & 0x4){ 161. cur\_state = 5; 162. } 163. **else** **if**(sensors & 0x1){ 164. cur\_state = 1; 165. } 166. **else**{ 167. cur\_state = 5; 168. } 169. **break**; 170. **case** 5: 171. **if**(sensors != 0x4){ 172. cur\_state = 6; 173. } 174. **break**; 175. } 176. } 177. } | | | | | | |
| 1. 实验结果及分析和（或）源程序调试过程   能够看到grade结果为满分，程序运行正常。    并且因为测试程序只测试红绿灯对于不同输入的响应。所以我将每个状态持续的时间调小，这样大大可以加快了完成测试的速度。 | | | | | | |
| 1. 实验心得   本实验学习了如何构建一个红绿灯。其中我们学会了编写代码实现一个能正常运行的红绿灯。对课堂上学习的状态机学习更直观，同时也进一步加深了对GPIO通信的理解。 | | | | | | |