1. Introduction   
   The objective of this lab was to design a circuit that would simulate a stoplight intersection with pedestrian walk functionality. Unlike previous labs this is written in C code rather than assembly. The main framework of this assignment is centered on Finite State Machines which will be relevant within a professional level of embedded systems.
2. Procedure/Discussion  
   For part 1 of the lab, code was directly referenced from the course textbook in one of the example codes given for a similar stoplight design. The cardinal directions were swapped appropriately to fit this lab in the state names. To add the pedestrian walk button, 19 more states were defined and added to handle the walk LED’s function and the timing of the button being held along with increasing the state array within the struct to a size of 8 to accommodate an extra button input. A new variable OutW was added to the struct for outputting to the Port 2 LED’s on the board.  
     
   Initially variables were used to handle the timing and walk button, but this was not allowed under the lab specifications **and** made the task harder, so the addition of many more states was chosen instead. Within the state array that denotes the next state based on button inputs, the latter 4 values which were added specifically for part 4 had state logic identical to the original function but instead using extra states that roughly indicated how long the button was held for (e.g goS is goS1 when walk button is held). These numbered states are additions specifically for part 2’s walk button and go up to 2 (levels). Level 2 of any of go states will default to level 2 of the respective wait state of that direction, which in turn defaults to the starting walk state that allows pedestrians to walk in any direction. After about a second of the board’s green LED being on, the walk state will successively default to an order of on and off states for the red board LED for about 1 second (with 100ms between state changes, so 5 rotations of on and off). The final state of the flashing states will read the car sensor switches and assign the next go state appropriately, with priority given to the south direction (this defaulted to the south direction when I demoed but has been fixed). This go state is not numbered.   
   While progressing up the levels of states, if the button is released before hitting level 2 which denotes the point at which the input is remembered (point of no return), the code will assign a non-numbered state to simulate forgetting about the button press.   
     
   To experimentally prove the system works, the stoplight states at any given time should be recorded to show that at any point, the walk and green light are never on at the same time. Also, this will be used to show that both green and yellow lights for each direction are never on at the same time as well. As long as these conditions are proven, then the fault will always lie on red light running or negligence on a driver’s part.  
     
   Regarding efficiency, the light changes from green to green within about 1.5 seconds assuming the button presses are polled at the start of a state transition. Assuming a car shows up every second, about 2 cars should be waiting on average when the light turns green for their direction. This also assumes cases where a car arrives at a yellow light but cannot clear without running a red. When a car arrives every half a second, 3 cars should be waiting on average.   
     
   The design is a Moore FSM the LED outputs depend solely on the current state. With a mealy design, a switch or button would directly affect the LED outputs during state transitions. There are 23 states in the FSM and approximately 43 next state transitions. To maintain the same walk button functionality (the timing), 23 states is the fewest that can be used. Going any lower would compromise the consistency of the 2 second button hold being remembered/forgotten. The linked data structure is implemented by a state pointer that will decide the current state. It is also used to access the next state by accessing the struct which it points to. This method is easy to implement and also easy to understand, since only the current state is important to the logic flow. Hence why it is ideal for the FSM.
3. Conclusion  
   The design worked as intended in the end, and only lacked next state logic coming out of a walk cycle based on car switch inputs during the demonstration (which was fixed by one line of code). Stoplights are an integral part of daily life so it’s no surprise that the lab tasks have real-world implications, albeit inefficient compared to professional Civil Engineering work.   
   As the first assignment in C, an understanding of basic C syntax was learned. Also Systick had unexpected faults when not initialized every run of the loop, hence why that function call also was nested in the while loop. If this lab could be redone, variables would not have been considered at all as they were more trouble to implement than altering the state machine transitions and states. A few hours were lost trying to make a variable work within an FSM.
4. References  
   Students consulted:  
   - Ryan Villanueva  
   - Griffin Ye  
   Embedded Systems: Introduction to the MSP432 Microcontroller, Vol. 1 (2nd Edition) by J.W Valvano
5. Appendix  
   Main Code: **#include** <stdint.h>

**#include** "SysTick.h"

**#include** "msp432p401r.h"

**struct** State {

uint32\_t Out;

uint32\_t Time;

**const** **struct** State \*Next[8];

uint32\_t OutW;

};

**typedef** **const** **struct** State State\_t;

**#define** goS &FSM[0]

**#define** waitS &FSM[1]

**#define** goW &FSM[2]

**#define** waitW &FSM[3]

**#define** walk &FSM[4]

**#define** walk1 &FSM[5]

**#define** walk2 &FSM[6]

**#define** walk3 &FSM[7]

**#define** walk4 &FSM[8]

**#define** walk5 &FSM[9]

**#define** walk6 &FSM[10]

**#define** walk7 &FSM[11]

**#define** walk8 &FSM[12]

**#define** walk9 &FSM[13]

**#define** walkUp &FSM[14]

**#define** goS1 &FSM[15]

**#define** goS2 &FSM[16]

**#define** waitS1 &FSM[17]

**#define** waitS2 &FSM[18]

**#define** goW1 &FSM[19]

**#define** goW2 &FSM[20]

**#define** waitW1 &FSM[21]

**#define** waitW2 &FSM[22]

State\_t FSM[23] = {

{0x21, 100, {goS, waitS, goS, waitS, goS1, waitS1, goS1, waitS1}, 0x01},

{0x22, 50, {goW, goW, goW, goW, goW1, goW1, goW1, goW1}, 0x01},

{0x0C, 100, {goW, goW, waitW, waitW, goW1, goW1, waitW1, waitW1}, 0x01},

{0x14, 50, {goS, goS, goS, goS, goS1, goS1, goS1, goS1}, 0x01},

{0x24, 100, {walk1, walk1, walk1, walk1, walk1, walk1, walk1, walk1}, 0x02},

{0x24, 10, {walk2, walk2, walk2, walk2, walk2, walk2, walk2, walk2}, 0x01},

{0x24, 10, {walk3, walk3, walk3, walk3, walk3, walk3, walk3, walk3}, 0x00},

{0x24, 10, {walk4, walk4, walk4, walk4, walk4, walk4, walk4, walk4}, 0x01},

{0x24, 10, {walk5, walk5, walk5, walk5, walk5, walk5, walk5, walk5}, 0x00},

{0x24, 10, {walk6, walk6, walk6, walk6, walk6, walk6, walk6, walk6}, 0x01},

{0x24, 10, {walk7, walk7, walk7, walk7, walk7, walk7, walk7, walk7}, 0x00},

{0x24, 10, {walk8, walk8, walk8, walk8, walk8, walk8, walk8, walk8}, 0x01},

{0x24, 10, {walk9, walk9, walk9, walk9, walk9, walk9, walk9, walk9}, 0x00},

{0x24, 10, {walkUp, walkUp, walkUp, walkUp, walkUp, walkUp, walkUp, walkUp}, 0x01},

{0x24, 10, {goS, goW, goS, goS, goS, goW, goS, goS}, 0x00},

{0x21, 100, {goS, waitS, goS, waitS, goS2, waitS1, goS2, waitS1}, 0x01},

{0x21, 100, {waitS2, waitS2, waitS2, waitS2, waitS2, waitS2, waitS2, waitS2}, 0x01},

{0x22, 50, { goW, goW, goW, goW, goW2, goW2, goW2, goW2}, 0x01},

{0x22, 50, { walk, walk, walk, walk, walk, walk, walk}, 0x01},

{0x0C, 100, {goW, goW, waitW, waitW, goW2, goW2, waitW1, waitW1}, 0x01},

{0x0C, 100, {waitW2, waitW2, waitW2, waitW2, waitW2, waitW2, waitW2, waitW2}, 0x01},

{0x14, 50, { goS, goS, goS, goS, goS2, goS2, goS2, goS2}, 0x01},

{0x14, 50, { walk, walk ,walk ,walk ,walk ,walk ,walk ,walk}, 0x01}

};

**void** **main**(**void**){

State\_t \*Pt; //state pointer

uint8\_t Input;

SysTick\_Init();

P2->SEL0 &= ~0x03;

P2->SEL1 &= ~0x03;

P2->DIR |= 0x03;

P4->SEL0 &= ~0x3F;

P4->SEL1 &= ~0x3F;

P4->DIR |= 0x3F;

P5->SEL0 &= ~0x07;

P5->SEL1 &= ~0x07;

P5->DIR &= ~0x07;

Pt = goS;

**while**(1)

{

SysTick\_Init();

P4->OUT = (P4->OUT&~0x3F)|(Pt->Out);

P2->OUT = (P2->OUT&~0x03)|(Pt->OutW);

SysTick\_Wait10ms(Pt->Time);

Input = (P5->IN&0x07);

Pt = Pt->Next[Input];

}

}

Systick main:

**#include** <stdint.h>

**#include** "msp432p401r.h"

// Initialize SysTick with busy wait running at bus clock.

**void** **SysTick\_Init**(**void**){

SysTick->CTRL = 0; // disable SysTick during setup

SysTick->LOAD = 0x00FFFFFF; // maximum reload value

SysTick->VAL = 0; // any write to current clears it

SysTick->CTRL = 0x00000005; // enable SysTick with no interrupts

}

// Time delay using busy wait.

// The delay parameter is in units of the core clock. (units of 333 nsec for 3 MHz clock)

**void** **SysTick\_Wait**(uint32\_t delay){

// method #1: set Reload Value Register, clear Current Value Register, poll COUNTFLAG in Control and Status Register

**if**(delay <= 1){

// without this step:

// if delay == 0, this function will wait 0x00FFFFFF cycles

// if delay == 1, this function will never return (because COUNTFLAG is set on 1->0 transition)

**return**; // do nothing; at least 1 cycle has already passed anyway

}

SysTick->LOAD = (delay - 1);// count down to zero

SysTick->VAL = 0; // any write to CVR clears it and COUNTFLAG in CSR

**while**((SysTick->CTRL&0x00010000) == 0){};

// method #2: repeatedly evaluate elapsed time

/\* volatile uint32\_t elapsedTime;

uint32\_t startTime = SysTick->VAL;

do{

elapsedTime = (startTime-SysTick->VAL)&0x00FFFFFF;

}

while(elapsedTime <= delay);\*/

}

// Time delay using busy wait.

// This assumes 3 MHz system clock.

**void** **SysTick\_Wait10ms**(uint32\_t delay){

uint32\_t i;

**for**(i=0; i<delay; i++){

SysTick\_Wait(30000); // wait 10ms (assumes 3 MHz clock)

}

}