CSE 474 Lab Report #2

Luke Jiang, 1560831

Abstract:

In this lab, I explored the GPTM interface and interrupt on TM4C123G launchpad.
In Section A, I used a timer to re-implement section A in lab1, which is to blink one of the three LEDs every second. Then, I updated the traffic light control system in lab1 so that each LED turns on for 5 seconds and that the system will respond only if the user holds down the button for at least 2 seconds.
In Section B, I first updated the FSM implementation in Section A by incorporating timer interrupt into the implementation. Then, I explored interrupts for GPIO Port F by making the on-board LED stay red if SW1 is pressed; or flash the blue LED if switch2 is pressed.

Introduction:

This lab involves using TI TM4C123G launchpad and IAR Workbench IDE as developing environment. The purpose of this lab contains 6 parts:

1) Introduction to timers and interrupts.
2) Familiarity with the vector table.
3) Application of interrupts.
4) Writing interrupt service routines.
5) Extracting information from datasheet.

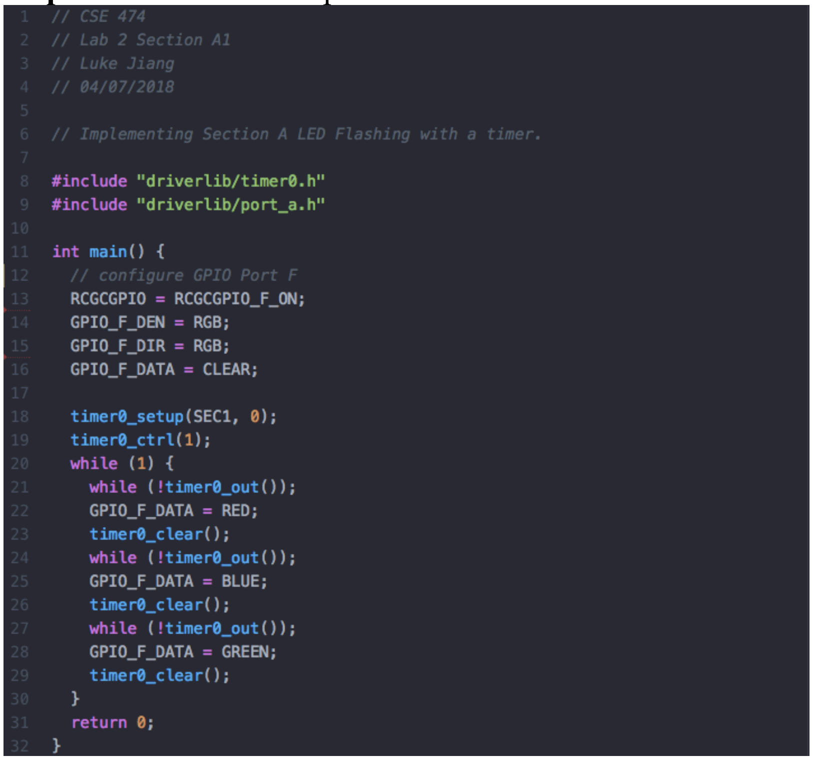
The methods I applied while developing this Lab includes:

1) Looking up datasheet for information about the registers used in this lab.
2) Maintaining a clear code structure for better readability and performance.

Procedure:

Section A:

Step 1: The code that I updated based on lab 1 Section A is the following:

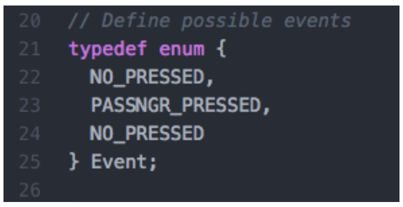


```c
// CSE 474
// Lab 2 Section A1
// Luke Jiang
// 04/07/2018

// Implementing Section A LED Flashing with a timer.

#include "driverlib/timer0.h"
#include "driverlib/port_a.h"

int main() {
  // configure GPIO Port F
  RCGCGPIO = RCGCGPIO_F_ON;
  GPIO_F_DEN = RGB;
  GPIO_F_DIR = RGB;
  GPIO_F_DATA = CLEAR;

  timer0_setup(SEC1, 0);
  timer0_ctrl(1);
  while (1) {
    while (!timer0_out());
    GPIO_F_DATA = RED;
    timer0_clear();
    while (!timer0_out());
    GPIO_F_DATA = BLUE;
    timer0_clear();
    while (!timer0_out());
    GPIO_F_DATA = GREEN;
    timer0_clear();
  }
  return 0;
}
```

I first include two drivers for interfacing GPIO Port A and GPTM at line 8 and 9. Due to code length concerns, the configuration of driver libraries and comments are presented in the driverlib folder (reference 5 and 6).
In the main function, I first configure GPIO Port F from line 13 to 16 just like I did in lab1. Then, I set up timer0 of 1Hz with interrupt disabled at line 18. At line 19, I turn on the timer. In the while(1) loop, I wait at line 21 until 1 second has passed and turn on red LED at line 22. After that, I open another 1s interval by clearing the timer. This pattern is repeated for handling subsequent blue and green LEDs.

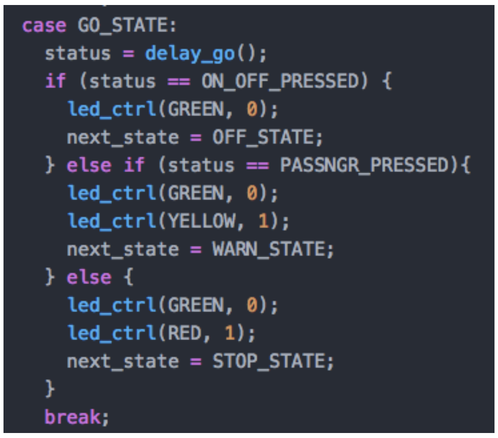
Step 2: The modified FSM implementation is the following:
The code I used to implement the functionality of flashing LEDs continuously is the following:
Before adding timers, I reorganized my previous FSM implementation for a better code structure and readability.

1) I added a enum called "Event" that defines possible inputs in each state of the FSM:



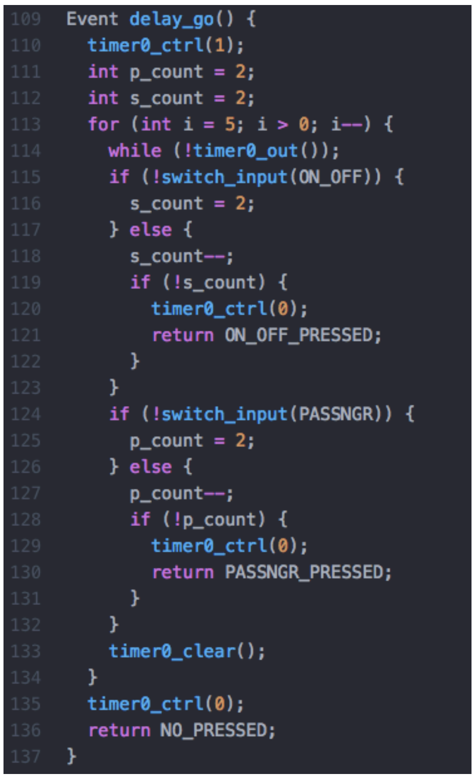
```
// Define possible events
typedef enum {
  NO_PRESSED,
  PASSNGR_PRESSED,
  NO_PRESSED
} Event;
```

2) I reorganized each case of the FSM implementation so that they follow the same pattern. Take GO_STATE for an example:



```
case GO_STATE:
  status = delay_go();
  if (status == ON_OFF_PRESSED) {
    led_ctrl(GREEN, 0);
    next_state = OFF_STATE;
  } else if (status == PASSNGR_PRESSED){
    led_ctrl(GREEN, 0);
    led_ctrl(YELLOW, 1);
    next_state = WARN_STATE;
  } else {
    led_ctrl(GREEN, 0);
    led_ctrl(RED, 1);
    next_state = STOP_STATE;
  }
  break;
```

When the process enters the GO_STATE, it first calls the delay_go() function, which handles delays and inputs happened in GO_STATE. Each state has its own delay function and each delay function returns a status that signifies what input has occurred at this state. Based on the input, the program decides to go to the next state and controls the LED output accordingly. It is in the delay functions that a 5s timer and switch_input() are used together for controlling how much time the program stays in a state.
The implementation of delay_go() function is the following:

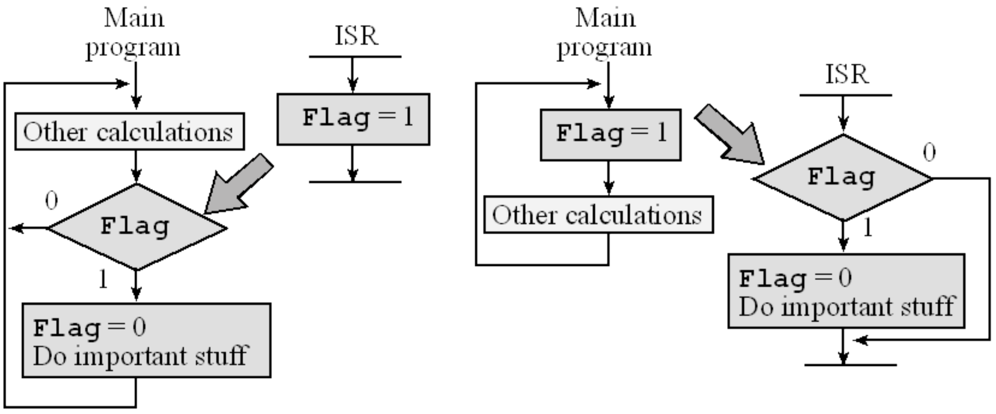


```
109 Event delay_go() {
110   timer0_ctrl(1);
111   int p_count = 2;
112   int s_count = 2;
113   for (int i = 5; i > 0; i--) {
114     while (!timer0_out());
115     if (!switch_input(ON_OFF)) {
116       s_count = 2;
117     } else {
118       s_count--;
119       if (!s_count) {
120         timer0_ctrl(0);
121         return ON_OFF_PRESSED;
122       }
123     }
124     if (!switch_input(PASSNGR)) {
125       p_count = 2;
126     } else {
127       p_count--;
128       if (!p_count) {
129         timer0_ctrl(0);
130         return PASSNGR_PRESSED;
131       }
132     }
133     timer0_clear();
134   }
135   timer0_ctrl(0);
136   return NO_PRESSED;
137 }
```

The function first turns on the 1-second timer and repeat 5 times at line 113 to implement a 5s timer. In order to know how long each switch has been pressed, I used two counters for the passenger switch and on/off switch. After each 1s has passed, I check if either of them is continuously pressed. If the switch input is 0, then either the user is not pressing at all, or he/she just released the button. Either case, the counter is cleared and reset back to 2. However, if the switch input is 1, then the user holds onto the switch for 1 second, so the counter decreases by 1. Once the counter hits 0, which means the user has continuously holds on for 2 seconds, the program should break from the normal 5s period and returns back to main FSM with a status and enters the corresponding next state. If no effective input happens during the 5s period, the program turns off the timer and also returns to main.
The rest parts of the implementation follow the same patterns described above, and are referenced as traffic_light.c in the code folder (reference 7).

Section B:

Step 1: Incorporating timer interrupts into traffic light control FSM
After updating the cstartup_M.c file to set up ISR for timer A0, I used the binary semaphore pattern to enable communication between the ISR process and the main process:



(image from http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm)

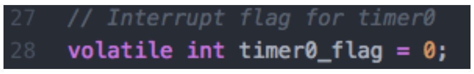
Flag represents whether an interrupt has occurred and has not yet been processed. For the timers, the flag signifies whether the time period has ended or not. When flag == 0, which means the timer is still running, the program would do one branch of calculation or simply loops back and keeps checking the flag. Once time is out, the ISR would be called, which acknowledges the interrupt and changes the flag into one. Then, after the processor switches back to user-mode, it would execute the interrupt handling branch before clearing the flag for future interrupts to happen.

The advantage of this pattern is the following:

1) It keeps the ISR short by only updating the flag after interrupt clear. This makes the program works better with the operating system.
2) The structure of main program doesn't need to change a lot, which reduces bugs and increases readability during development
3) The algorithm is simple to understand and implement in C.

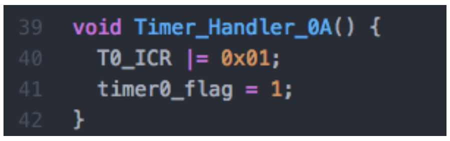
The modification I made based on Section A2 is the following:

1) I need to declare and define the flag in the main program. Since the flag is changed not due to its context, it needs the volatile specifier to avoid compiler optimization:



```c
// Interrupt flag for timer0
volatile int timer0_flag = 0;
```

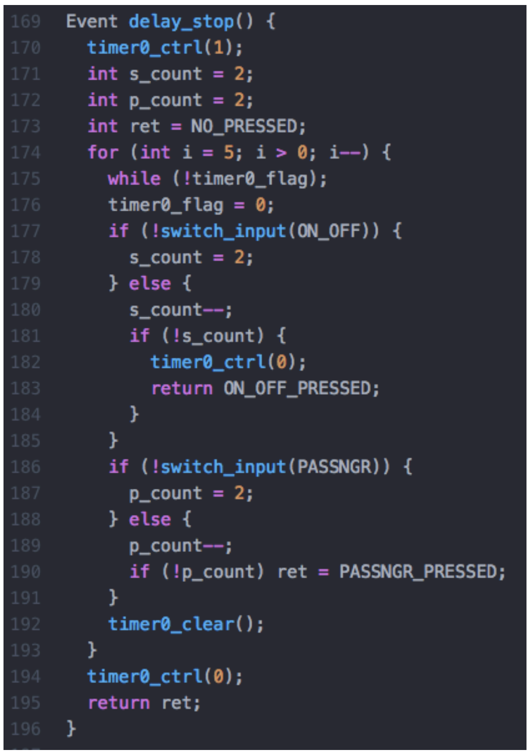
2) I need to declare and define the ISR. Like previously mentioned, it need to first acknowledge the interrupt, then set the timer0_flag into 1. The former is achieved by writing 1 to P0 bit of T0_ICR:



```c
void Timer_Handler_0A() {
  T0_ICR |= 0x01;
  timer0_flag = 1;
}
```

3) For each delay() function implementation, instead of calling timer0_out() constantly and checking from the T0_RIS register, I use polling timer0_flag to know whether time is out. Once the ISR returns back to main, I need to set timer0_flag back to 0 indicating the previous interrupt is handled:

The same modification need to be applied to all delay() functions, while the main FSM structure stays the same. The complete code is referenced as traffic_light_interrupt.c in the code folder (reference 8).

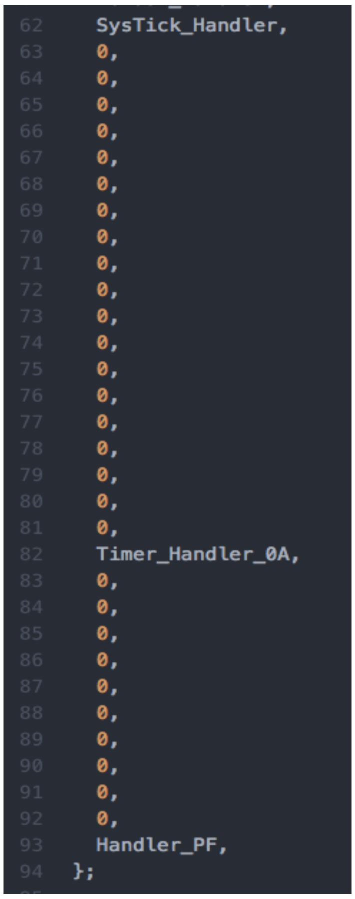


```c
169 Event delay_stop() {
170   timer0_ctrl(1);
171   int s_count = 2;
172   int p_count = 2;
173   int ret = NO_PRESSED;
174   for (int i = 5; i > 0; i--) {
175     while (!timer0_flag);
176     timer0_flag = 0;
177     if (!switch_input(ON_OFF)) {
178       s_count = 2;
179     } else {
180       s_count--;
181       if (!s_count) {
182         timer0_ctrl(0);
183         return ON_OFF_PRESSED;
184       }
185     }
186     if (!switch_input(PASSNGR)) {
187       p_count = 2;
188     } else {
189       p_count--;
190       if (!p_count) ret = PASSNGR_PRESSED;
191     }
192     timer0_clear();
193   }
194   timer0_ctrl(0);
195   return ret;
196 }
```

Note: The only difference between this piece of code and delay_off() in section A is at line 175 and 176.

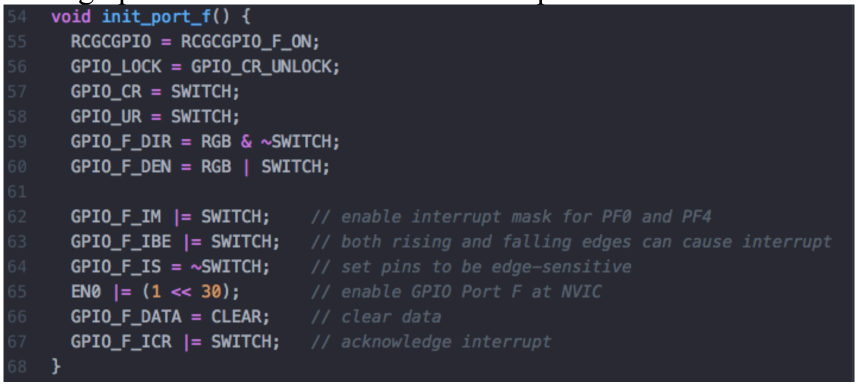
Step 2: Using interrupts for GPIO and timer to control on-board pushbuttons and LEDs

1) I added the handler for GPIO Port F and the handler for timer 0A into the startup file:



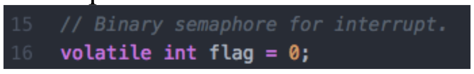
```c
    SysTick_Handler,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    Timer_Handler_0A,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    Handler_PF,
  };
```

2) Setting up GPIO Port F so that the interrupt is enabled:



```
void init_port_f() {
  RCGCGPIO = RCGCGPIO_F_ON;
  GPIO_LOCK = GPIO_CR_UNLOCK;
  GPIO_CR = SWITCH;
  GPIO_UR = SWITCH;
  GPIO_F_DIR = RGB & ~SWITCH;
  GPIO_F_DEN = RGB | SWITCH;

  GPIO_F_IM |= SWITCH;    // enable interrupt mask for PF0 and PF4
  GPIO_F_IBE |= SWITCH;   // both rising and falling edges can cause interrupt
  GPIO_F_IS = ~SWITCH;    // set pins to be edge-sensitive
  EN0 |= (1 << 30);       // enable GPIO Port F at NVIC
  GPIO_F_DATA = CLEAR;    // clear data
  GPIO_F_ICR |= SWITCH;   // acknowledge interrupt
}
```

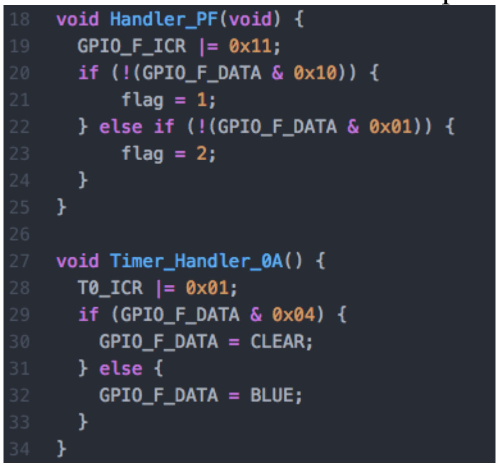
3) Following the same procedure described in Section B1, I declare a flag for PF interrupt:



```
// Binary semaphore for interrupt.
volatile int flag = 0;
```

It turns out that the overall logic is not complex, so only the flag for GPIO is really needed

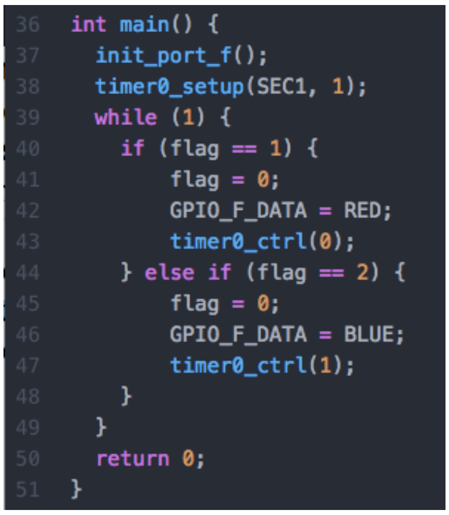
4) Declare and define the two interrupt handlers:



```
18  void Handler_PF(void) {
19    GPIO_F_ICR |= 0x11;
20    if (!(GPIO_F_DATA & 0x10)) {
21        flag = 1;
22    } else if (!(GPIO_F_DATA & 0x01)) {
23        flag = 2;
24    }
25  }
26
27  void Timer_Handler_0A() {
28    T0_ICR |= 0x01;
29    if (GPIO_F_DATA & 0x04) {
30      GPIO_F_DATA = CLEAR;
31    } else {
32      GPIO_F_DATA = BLUE;
33    }
34  }
```

The Handler_PF first acknowledges the interrupt by writing P0 and P4 into GPIO_F_ICR, then, depending on whether switch1 (line 20) or switch2 (line 22) is pressed, the handler sets the flag into different values. Similarly, the timer handler first acknowledges the timer interrupt, then, it switches between on and off states: If the previous state of blue LED is on (line 29), then the next state is off; If the previous state of blue LED is off (line 31), then the next state is on. This implements the blinking requirement in the spec.

5) Finally, the main program controls initializations and timers:



```
36  int main() {
37    init_port_f();
38    timer0_setup(SEC1, 1);
39    while (1) {
40      if (flag == 1) {
41          flag = 0;
42          GPIO_F_DATA = RED;
43          timer0_ctrl(0);
44      } else if (flag == 2) {
45          flag = 0;
46          GPIO_F_DATA = BLUE;
47          timer0_ctrl(1);
48      }
49    }
50    return 0;
51  }
```

It first calls init_port_f() and timer0_setup() to properly configure the GPIO and the timer. Then, in the while (1) loop, it handles the GPIO interrupt flag: If switch1 is pressed (line 40), then the LED should stay red and timer0 should be turned off. Else; if switch2 (line 44) is pressed, the LED should turn blue and timer0 is enabled.

Results

The implementation of traffic light control FSM works fine under the following preconditions:

1) If the switch for on/off and the switch for passenger are pushed together, the first switch that stays on for 2 seconds is of effect. The other one is ignored.
2) A switch push delay of 1 second (at most) is tolerated. This scenario would happen if someone pushes the button just after the 1-second clock is cleared and it would take more than 2 seconds but less than three seconds to change into the next state. It is possible to increase the speed of the clock for better accuracy.
3) A switch push happened after 3s at each clock cycle is of no effect. It is possible to implement the FSM with two timers that, when a switch press happens after 3s, would enable a 2s timer and checks if the switch is constantly pressed in this 2 seconds. If so, the 5s timer would be handled only after the 2s timer is handled, which means it is possible to have a delay longer than 5s in some states.

The challenges I faced while developing this lab are:

1) Configuring GPIO Port F to enable interrupt. The procedure described in lecture_5.pdf doesn't work on my board, so I spent a lot of time trying to make the interrupt work.
2) Deciding how many timers to use for the traffic light system implementation.

Conclusion

I learned the following skills in this lab:

1) Looking up datasheet.
2) Using general purpose timers in the project.
3) Configuring interrupts for timers and GPIOs.
4) Implementing ISR with binary semaphore pattern.

The concepts and techniques introduced in this lab is crucial for future labs and projects. Timers offer a way of controlling delays with accuracy, and interrupts makes it easier for me to handle input events and incorporate the result into the main process.

Reference

1. TM4C123 data sheet
2. http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm
3. lecture_4.pdf and ledture_5.pdf
4. lab2_code/drivelib/led_switch_driver.h
5. lab2_code/drivelib/timer0.h
6. lab2_code/drivelib/port_a.h
7. lab2_code/traffic_light.c
8. lab2_code/traffic_light_interrupt.c