EE2028 Assignment 2 Report

FitNUS

Done by:

Wang Haozhe (A0168337W)

Wang Shuhui (A0173413M)

Tuesday Afternoon Lab Group

Table of Contents

Introduction

The aim of this project is to simulate a fitness tracking system, known as FitNUS. The fitness tracking system has 3 modes of operation: Initialization Mode, Climb Mode and Emergency Mode and will be transmitting data wirelessly and periodically to a server known as FitTrackX when certain conditions are met. The Initialization Mode is the mode in which FitNUS is started. Climb Mode is the mode in which FitNUS is monitoring the climbing statistics (light intensity, temperature and net acceleration). Emergency Mode is activated when the danger of the climber falling is detected.

Below is the list of peripherals used and their roles:

1. OLED Display — To display the mode of operation, sensor readings and warning messages
2. 7 Segment LED Display — To display numbers and letters
3. Blue and Red LED — For BLINK_BLUE and ALTERNATE_LED when SW3 is pressed and in Emergency Mode respectively
4. Temperature Sensor — To monitor the body temperature of the climber (read using EINT3 interrupt)
5. Accelerometer — To detect the falling event of the climber
6. Light Sensor — For ambient light detection (read using EINT3 interrupt)
7. LED Array — To indicate the how dim the surroundings are when ambient light falls below 3000 lux
8. SW3 — Used for MODE_TOGGLE. Press SW3 to enter the transition from Initialization Mode to Climb Mode (read using EINT0 interrupt)
9. SW4 — Used for EMERGENCY_OVER to exit Emergency Mode and re-enter Climb Mode. Press SW3 and SW4 simultaneously to exit Emergency Mode
10. Xbee module— Wireless transmitter to send data to FitTrackX.

Key Objectives in Each Mode of Operation

Initialization Mode:

1. Once the system is powered on, the OLED display will display “Initialization mode. Press TOGGLE to climb”.
2. The message “Start” will be sent to FitTrackX.
3. Once SW3 is pressed, the OLED display will display “INITIALIZATION COMPLETE. ENTERING CLIMB MODE”, the 7 segment display counts down from 9 to 0 with the blue LED blinking in 1 second intervals simulatanously. The system automatically enters Climb Mode afterwards.

Climb Mode:

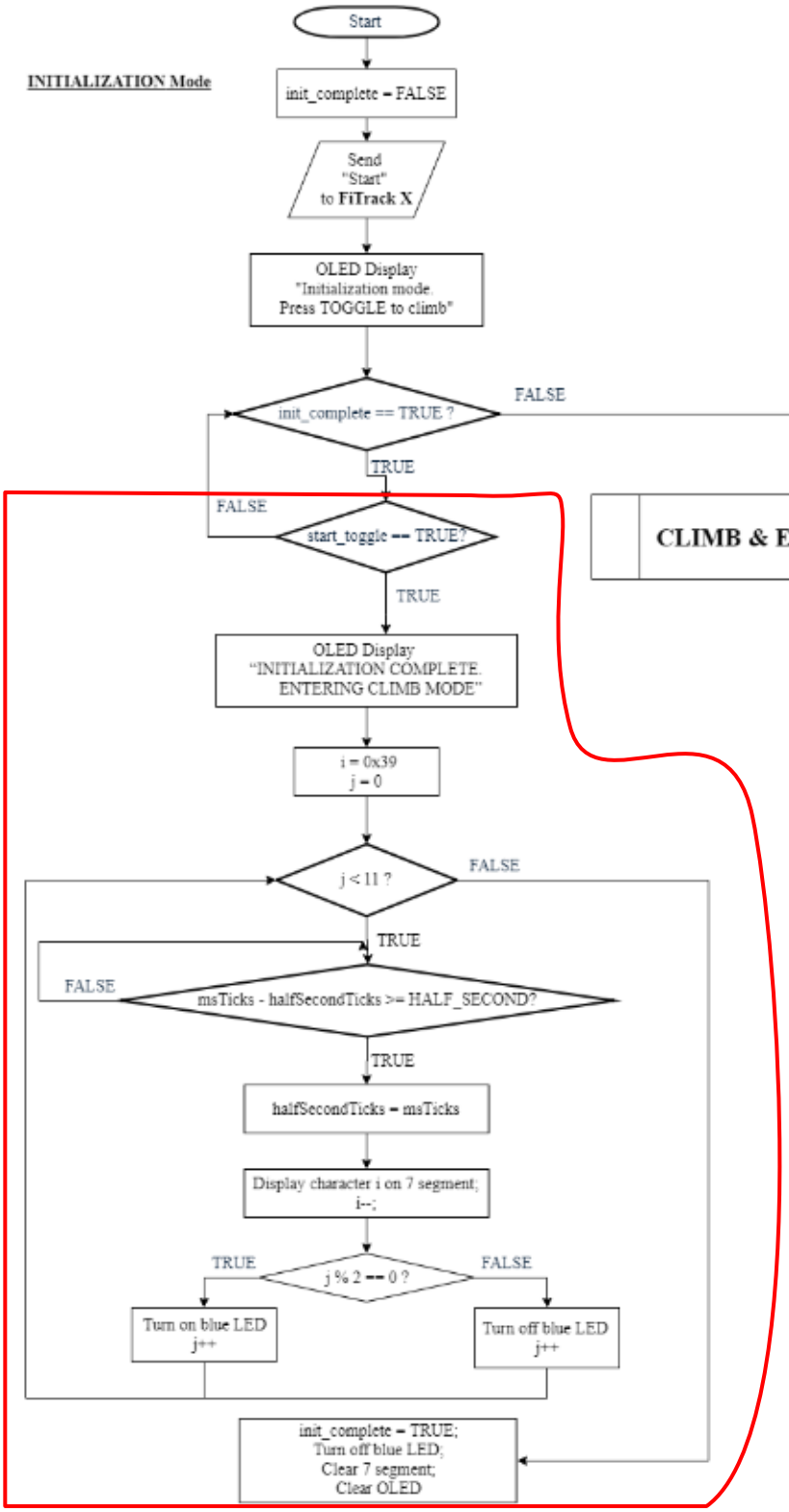
1. An accelerometer reading will be read first.
2. OLED display will display “CLIMB” as well as the light intensity, temperature and net acceleration.
3. If the light intensity is below the threshold of 3000 lux, the OLED display will display an additional “DIM” message and light up a corresponding number of LEDs on the LED array depending on how dim the surroundings are (the dimmer the surroundings, more LEDs will light up).
4. If the temperature is above the threshold of 28.0°C, the OLED display will display “REST NOW” for 3 seconds before returning to Climb Mode. This will only be triggered every time the temperature crosses the threshold.
5. If the net acceleration is higher than the threshold of 0.1g, the system will enter Emergency Mode.

Emergency Mode:

1. The OLED display will display “EMERGENCY Mode!” as well as the net acceleration, temperature and the duration for which FitNUS has been in Emergency Mode.
2. The red and blue LEDs will blink alternately every 500ms.
3. When SW3 and SW4 are pressed simultaneously, the 7 segment display will display the letters S-A-U-E-D, with the letter changing every second and the blue LED blinking in 1 second intervals simultaneously. The system will take one accelerometer reading and return to Climb Mode automatically afterwards.

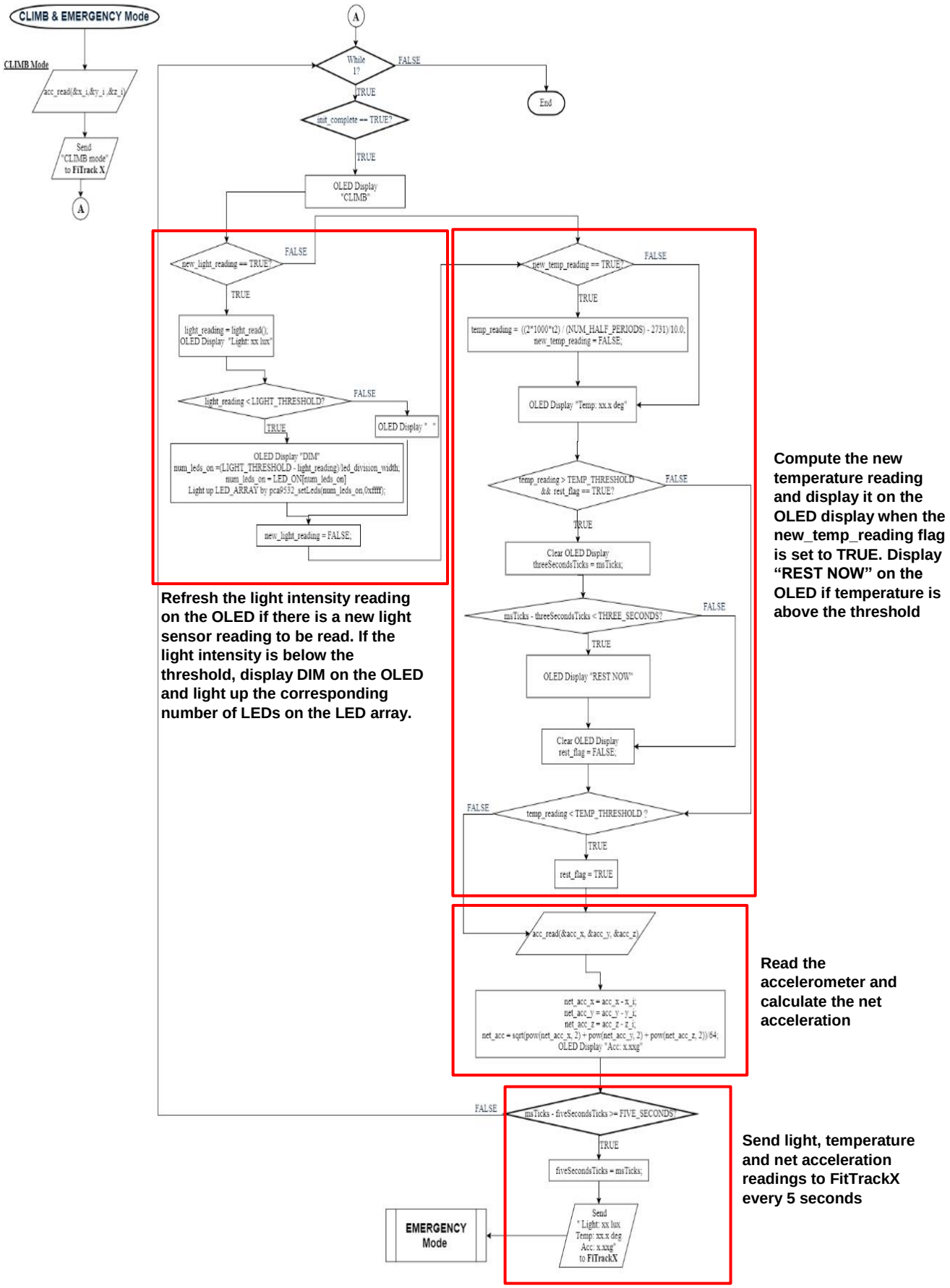
Flowcharts and Detailed Implementation

1. Initialization Mode



Once start_toggle is set to TRUE (SW3 is pressed), the toggling process from Initialization Mode to Climb Mode begins. The init_complete flag will be set to TRUE at the end of the toggling process to indicate that the system is ready to enter into Climb Mode.

2. Climb Mode



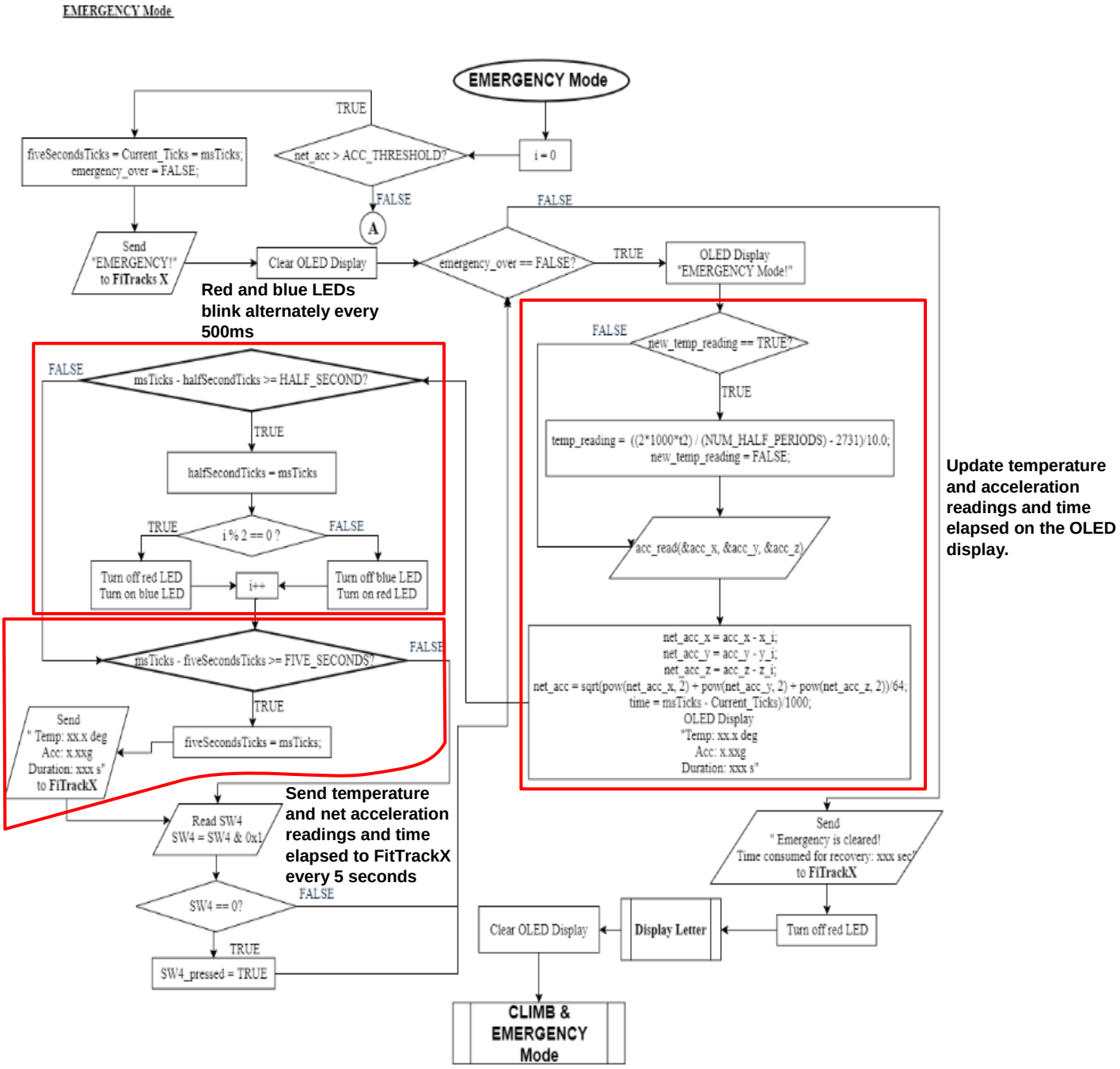
Refresh the light intensity reading on the OLED if there is a new light sensor reading to be read. If the light intensity is below the threshold, display DIM on the OLED and light up the corresponding number of LEDs on the LED array.

Compute the new temperature reading and display it on the OLED display when the new_temp_reading flag is set to TRUE. Display "REST NOW" on the OLED if temperature is above the threshold

Read the accelerometer and calculate the net acceleration

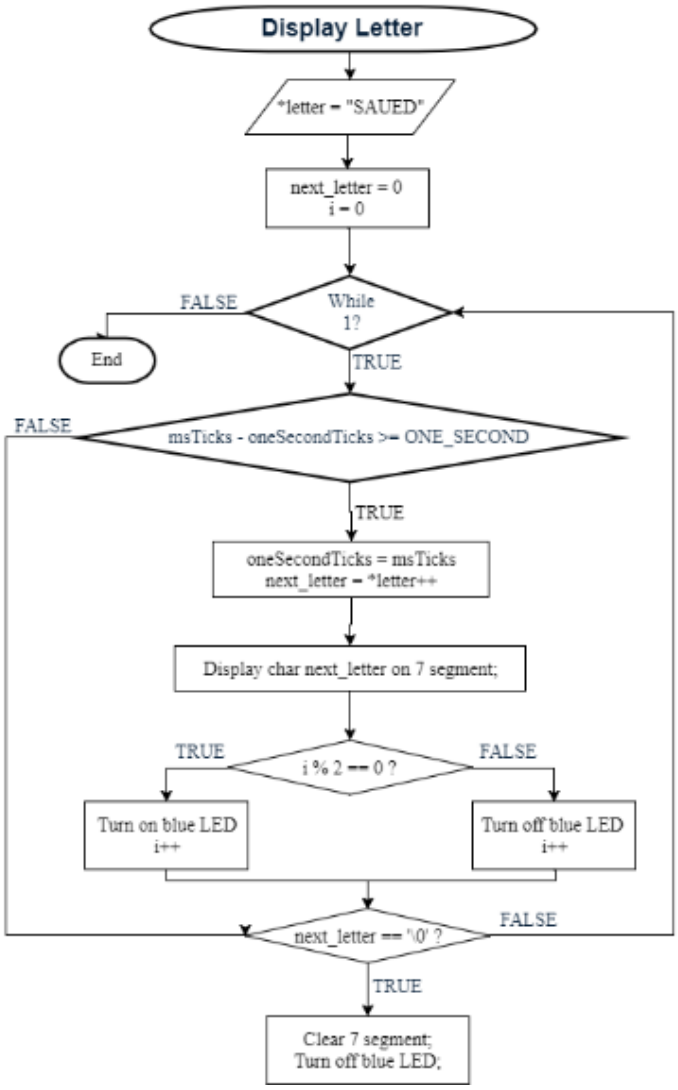
Send light, temperature and net acceleration readings to FitTrackX every 5 seconds

3. Emergency Mode



4. The displayLetter function (for exiting emergency mode)

The displayLetter function is a function that displays the letters S-A-U-E-D on the 7 segment display and controls the toggling of the state of the blue LED while displaying the letters. The state of the blue LED is inverted every 1 second. The flow chart for the implementation of this function shown below.



5. Priority Levels of the Interrupts Used in FitNUS

Priority Level	Interrupt
0	Systick Interrupt
1	EINT0 Interrupt (for reading the state of SW3)
2	EINT3 Interrupt (for reading light and temperature sensors)

Accurate time-keeping is always the top priority of FitNUS, hence Systick interrupt should have the highest priority. SW3 is need for toggling from Initialization Mode to Climb Mode as well as to exit Emergency Mode. These are more important tasks than reading sensors therefore EINT0 interrupt has a higher priority than EINT3 interrupt.

6. Reading the State of SW3 Using EINT0 Interrupt

As we will be reading the light and temperature sensors using interrupts, it is important that the interrupt for SW3 should be able to preempt the light and temperature sensor interrupts. Therefore we chose to use EINT0 interrupt for SW3 instead of EINT3 so that we could set the priority of the SW3 interrupt above that of the light and temperature sensors. We have configured EINT0 to be an edge triggered interrupt by writing 1 to bit 0 of the External Interrupt Mode (EXTMODE) register and to trigger an interrupt on the falling edge by writing 0 to bit 0 of the External Interrupt Polarity (EXTPOLAR) register. The interrupt is cleared by writing 1 to bit 0 of the External Interrupt Flag (EXTINT) register. The code snippets below show how the EINT0 interrupt was implemented in our program.

Inside EINT0_IRQHandler function:

```
void EINT0_IRQHandler(void)
{
    if(SW4_pressed == TRUE)
    {
        emergency_over = TRUE;
        SW4_pressed = FALSE;
        LPC_SC->EXTINT = (1<<0);
    }
    else
    {
        start_toggle = TRUE;
        LPC_SC->EXTINT = (1<<0);
    }
}
```

Inside the main code:

```
LPC_SC->EXTMODE = (1<<0);
LPC_SC->EXTPOLAR = (0<<0);
NVIC_EnableIRQ(EINT0_IRQn);
```

Project Enhancement

As time-keeping in FitNUS is implemented in a non-blocking manner, any delays due to the time needed to read and compute sensor readings can cause FitNUS to be unable to accurately keep track of the time. Our solution to mitigate this problem is to read the light sensor and temperature sensor using interrupts.

1. Reading Light Sensor Readings Using EINT3 Interrupt

When the light sensor interrupt is triggered, the new_light_reading flag is set to TRUE in the EINT3_IRQHandler function and then the interrupt is cleared. The function light_read is then called upon checking that the new_light_reading flag has been set to TRUE. The code snippets below show the implementation of the light sensor interrupt in our program.

Inside EINT3_IRQHandler function:

```
if((LPC_GPIOINT->IO2IntStatF>>5) & 0x1)
{
    new_light_reading = TRUE;
    LPC_GPIOINT->IO2IntClr = 1<<5;
    light_clearIrqStatus();  // clear interrupt active status
}
```

Inside the main code:

```
if(new_light_reading == TRUE)
{
    light_reading = light_read();
```

With the light sensor interrupt, the light_read function is now only called when there is a new light reading to be read.

2. Reading Temperature Sensor Readings Using EINT3 Interrupt

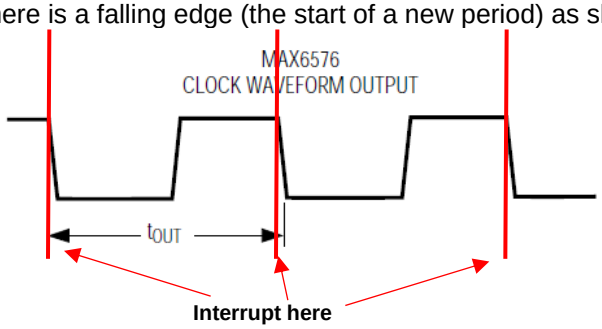
We have chosen to use interrupts to read the temperature as well. As stated in the datasheet for the MAX6576 temperature sensor, the sensor converts the current temperature to a period. The output of the device is a free-running, 50% duty-cycle square wave with a period that is proportional to the absolute temperature (°K) of the device. Hence, in order to ensure a reliable temperature reading, an average reading (of 170 readings) is used instead of just one. However, the temp_read helper function in the temp.h library is blocking in nature as it waits for the 340 state changes (340 half periods) to pass before reading calculating the time elapsed for 340 state changes and using that to calculate the temperature as shown in the code snippet of the temp_read function below.

```
state = GET_TEMP_STATE;

/* get next state change before measuring time */
while(GET_TEMP_STATE == state);
state = !state;

t1 = getTicks();

for (i = 0; i < NUM_HALF_PERIODS; i++) {
   while(GET_TEMP_STATE == state);
    state = !state;
}

t2 = getTicks();
if (t2 > t1) {
    t2 = t2-t1;
}
else {
    t2 = (0xFFFFFFFF - t1 + 1) + t2;
}


return ( (2*1000*t2) / (NUM_HALF_PERIODS*TEMP_SCALAR_DIV10) - 2731 );
```

NUM_HALF_PERIODS = 340

Using a while loop to wait for 340 state changes

This is very resource inefficient and it can also cause the timing of our system (which was implemented in a non-blocking format) to be inaccurate. Our solution is to set the temperature sensor to trigger an interrupt whenever there is a falling edge (the start of a new period) as shown in the datasheet for the temperature sensor:



Since we are only interrupting once every period, we will set up a counter to count 171 interrupts[1] before computing the time elapsed and setting new_temp_reading flag to TRUE and then clearing the interrupt. The calculation of the temperature reading is then computed in the main code upon checking that the new_temp_reading flag has been set to TRUE.

[1] We are using 171 periods instead of 170 periods because the first time reading was technically taken before the for-loop in the temp_read function, whereas in our implementation of the temperature sensor interrupt we are effectively taking the first time reading within the for-loop when we compare it to the temp_read function. To prevent possible inaccuracies in the temperature reading, we chose to take one extra period.

The code snippets below show how the temperature sensor interrupt has been implemented.

Inside EINT3_IRQHandler function:

```
if((LPC_GPIOINT->IO0IntStatF>>2) & 0x1)
{
    if(counter == 0)
        t1 = msTicks;

    counter++;

    if(counter == 171)
    {
        t2 = msTicks;
        if (t2 > t1)
            t2 -= t1;

        else
            t2 += (0xFFFFFFFF - t1 + 1);

        new_temp_reading = TRUE;
        counter = 0;
    }
    LPC_GPIOINT->IO0IntClr = 1<<2;
}
```

Inside the main code:

```
// computes the temperature reading from temperature sensor
if(new_temp_reading == TRUE)
{
    temp_reading =  ((2*1000*t2) / (NUM_HALF_PERIODS) - 2731)/10.0;
    new_temp_reading = FALSE;
}
```

Issues Encountered During the Project

1. Program Gets Stuck during I²C Transmission

After our group implemented the light sensor interrupt and temperature sensor interrupts, we realized that our program gets stuck randomly in either the I2C_Start, I2C_GetByte or I2C_SendByte function in the LCP17xx_i2c.h library. We realized while we were debugging our program that the program was always stuck in the line below that is present in all 3 functions:

```
while (!(I2Cx->I2CONSET & I2C_I2CONSET_SI));
```

This line of code was causing the program to wait indefinitely for the SI bit in the I2C2CONSET register to be set to 1 before continuing the execution of the program. Our solution to this problem was to add a timeout condition to the I2C_Start, I2C_GetByte and I2C_SendByte function so that the program will continue to run after waiting for a certain period of time. The code snippet of our modification is shown below:

```
int i = 0;
while (!(I2Cx->I2CONSET & I2C_I2CONSET_SI) && i < 1000){
    i++;
}
```

2. Errors Present in rgb.h Helper Functions

There were also errors in the rgb_init and rgb_setLeds function rgb.h library. The errors are as follows:

```
void rgb_init (void)
{
    GPIO_SetDir( 2, 0, 1 );
    GPIO_SetDir( 0, (1<<26), 1 );
    GPIO_SetDir( 2, (1<<1), 1 );
}
```

Should be writing 1 to FIODIR register. Writing 0 has no effect

```
void rgb_setLeds (uint8_t ledMask)
{
    if ((ledMask & RGB_RED) != 0) {
        GPIO_SetValue( 2, 0);
    } else {
        GPIO_ClearValue( 2, 0);
    }
```

Should be writing 1 to FIOSET and FIOCLR registers. Writing 0 has no effect

Our solution was to not use the rbg,h library in our program and directly use the GPIO_SetValue and GPIO_ClearValue functions to turn the blue and red LEDs on or off. The code snippet below shows how this was done.

```
if (msTicks - halfSecondTicks >= HALF_SECOND){
    halfSecondTicks = msTicks;
    if (i%2 == 0){
        GPIO_ClearValue(2, 1<<0); // red LED off
        GPIO_SetValue(0, 1<<26); // blue LED on
    }
    else{
        GPIO_ClearValue(0, 1<<26); // blue LED off
        GPIO_SetValue(2, 1<<0);  // red on
    }
    i++;
}
```

Suggestions:

For this assignment, we feel like the importance of the flowchart should be highlighted by the Lab TAs at the start of the first lab session for this project. It will be better if a short section of lab is focused on introducing how to draw a flowchart for complex program. The flowchart is very helpful in assisting us to figure out the inevitable logic flaws in our program and change them. The work load of this assignment is also quite heavy as there are lots of overlapping of pins used by the devices in the baseboard and we need to design our program carefully to minimise the occurrence of conflicts. In general, this assignment is very useful in training us to be familiarised with hardware coding, but some small amendments can be made to make this project more effective and interesting in the future.

Conclusion

Over the course of this project, we have learnt how a microcontroller works and how to interface different peripherals with it. We have strengthened our ability to learn independently through implementing the enhancements and reading the datasheets to understand how different devices take readings and transmit data and how we can use them. We have also learnt how we can make a system's operation more resource friendly (mainly through using interrupts and writing efficient code). The issues that we faced during the project have also helped us to reinforce our theory knowledge about microcontroller programming and interfacing and make use of it to debug our programs.