



NUS

National University
of Singapore

National University of Singapore
Dept of Electrical & Computer Engineering (ECE)
EE2024 Programming for Computer Interfacing
Assignment 2 (Semester 2, AY2017/18)

Name: Deepak Buddha

Matric No: A0155454A

Name: Azizi Azfar

Matric No: A0155506H

Table of Contents

1. Overview	4
1. 1 Summary of modes	4
1. 2 Summary of warning states	5
2. Objectives	6
3. System Designs: Flowcharts	6
3. 1. Stationary Mode Flowchart	7
3. 2. Countdown Flowchart	7
3. 3. Launch Mode Flowchart	9
3. 4. Return Mode Flowchart	10
3. 5. TIMER 0 Flowchart	11
3. 6. EINT0 Flowchart	12
3. 7. UART3 Flowchart	13
3. 8. checkManualSettings (Enhancement) Flowchart	14
4. Implementation of code	14
4. 1. 1 Setup and initializations	15
4. 1. 2 Main function and while(1) loop	16
4. 2 Stationary mode	17
4. 2. 1 configStationary();	18
4. 2. 2 stationaryMode();	19
4. 3 Launch Mode	20
4. 3 .1 countDown()	21
4. 3 .2 configLaunch()	21
4. 3 .3 launchMode()	23

4. 4 Return Mode	24
4. 4. 1 configReturn()	25
4. 4. 2 returnMode()	26
4. 5 Interrupts	27
4. 5. 1 EINT0 : MODE_TOGGLE	27
4. 5. 1. 1 EINT0_IRQHandler	28
4. 5. 1. 2 modeToggle()	30
4. 5. 2 EINT3 Interrupt	30
4. 5. 2. 1 Light Sensor Interrupt	31
4. 5. 2. 2 Temperature Sensor Interrupt	32
4. 5. 3 UART3 Interrupt	33
4. 5. 3. 1 UART3_IRQHandler() & Call back function	33
4. 5. 4 TIMER0 Interrupt	34
4. 5. 4 .1 Initialization of TIMER0 Interrupt	34
4. 5. 4 .2 TIMER0_IRQHandler() & telemetryMsg()	35
4. 5. 5 TIMER1 Interrupt	36
4. 5. 5 .1 Initialization of TIMER1 Interrupt	36
4. 5. 4 .2 TIMER0_IRQHandler() & updateReturn()	37
4. 5. 6 Interrupt Priority Setting	38
4. 6 Application Logic Enhancement	39
4. 6. 1 checkManualSettings()	40
4. 6. 2 Control movements on Baseboard and NUSCloud display	42
5. Problems encountered and solutions proposed	43
6. Improvement suggestions	44
7. Conclusion	44
APPENDIX A	45
APPENDIX B	49

1. Overview

This assignment emulates a manned space flight system using the LPC1769 microcontroller and the Embedded Artists Baseboard. We shall refer to this system as **NUSpace**. This system sends data periodically to a server known as **NUSCloud** which is a terminal running on a remote device that communicates wirelessly with the XBee RF module.

NUSpace has three modes of operation: **Stationary, Launch and Return modes**, and will be transmitting to NUSCloud if certain conditions are met. The stationary mode is the mode in which the rocket is still in its launchpad. Launch mode is the mode in which the rocket is launched towards space. Return mode is the mode in which the space shuttle mounted on the rocket is detached and returns to and on earth.

We have implemented this assignment to match the context of *NUSpace* and *NUSCloud*. The exact details of the assignment requirements can be found in the EE2024 NUS wiki, however some basic information is provided in the subsections below, this report will instead focus on how we achieve those requirements; demonstrated through our flowcharts and analysis of our code. Next this report highlights the problems faced and improvements that could made before a concluding remark.

1. 1 Summary of modes

	STATIONARY	LAUNCH	RETURN
Sensors in use	Temperature sensor only	Temperature and accelerometer	Light sensor only
OLED Display	'STATIONARY ' and Temperature readings in real time	'LAUNCH' and Temperature readings and X,Y accelerometer readings in real time	'RETURN' only
7 Segment Display	'F' and F-0 when countdown is initiated	'0' only	'0' only

Peripherals required (apart from sensors, OLED and 7 segment display)	RGB lights for warning state UART SW3 (MODE_TOGGLE) SW4 (CLEAR WARNING) Joystick and Rotary switch (CUSTOM CONTROL :Extra Feature)	RGB lights for warning state UART SW3 (MODE_TOGGLE) SW4 (CLEAR_WARNING) Joystick and Rotary switch (CUSTOM CONTROL : Extra Feature)	16 LED ARRAY UART Joystick and Rotary switch (MANUAL CONTROL :Extra feature)
---	--	---	--

1. 2 Summary of warning states

	Temperature of Fuel Tank	Orientation of Rocket	Obstacle Detection
Threshold (self-defined and adjustable)	27°C	0.4g for both x.y-axes	3000 light intensity units
OLED Display	"Temp. too high"	"Veer off course"	"Obstacle Near"
RGB LED warning	Blue LED blinks 333ms	Red LED blinks 333ms	LED-Array lights up accordingly
UART message	"Temp. too high" Sent once each time temp warning triggers	"Veer off course" Sent once each time acc warning triggers	"Obstacle Near" "Obstacle Avoided" Sent once each time obstacle warning triggers and is cleared

2. Objectives

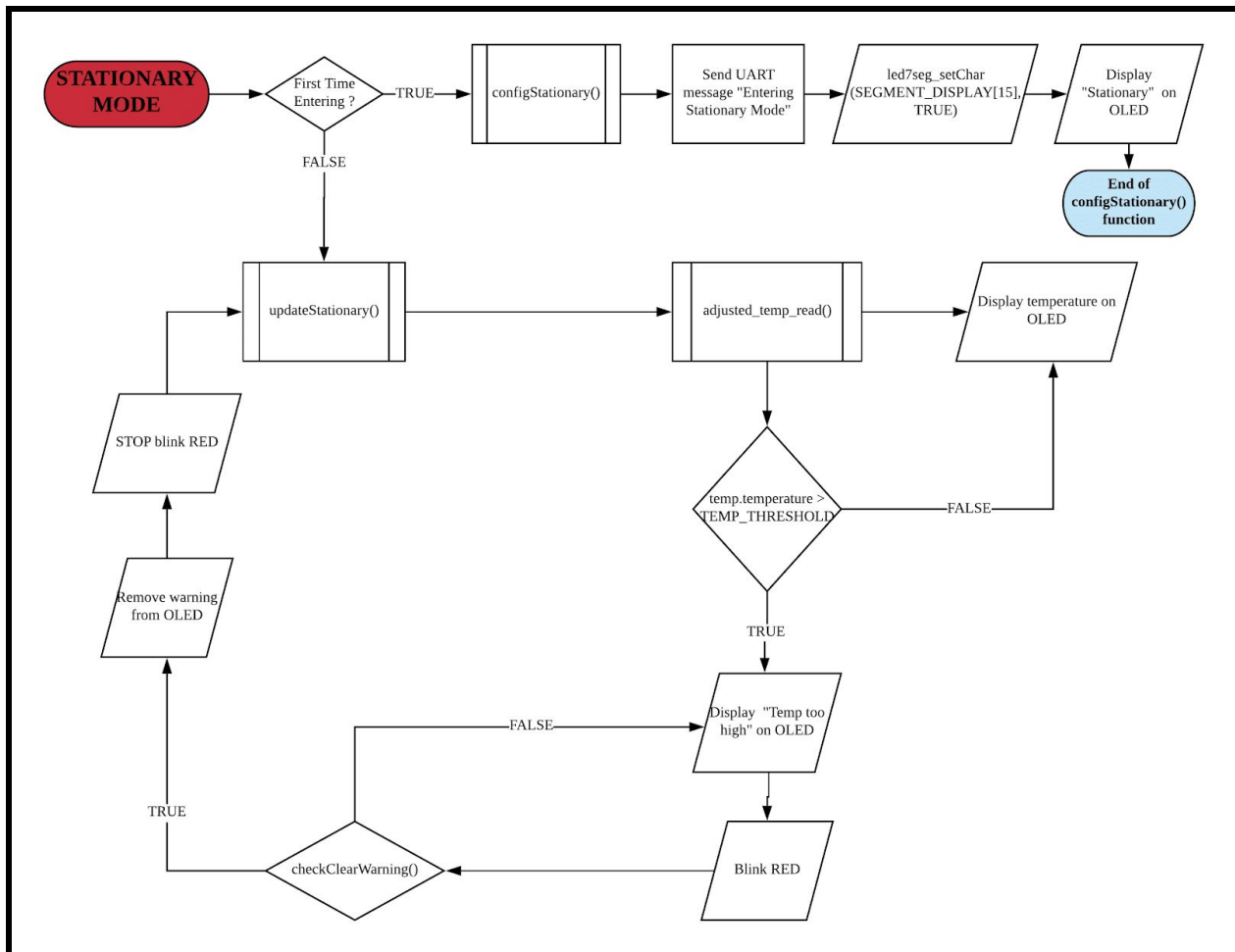
The objectives of the assignment for our group are as listed below:

1. Successfully implement the NUSpace devices used in the assignment required:

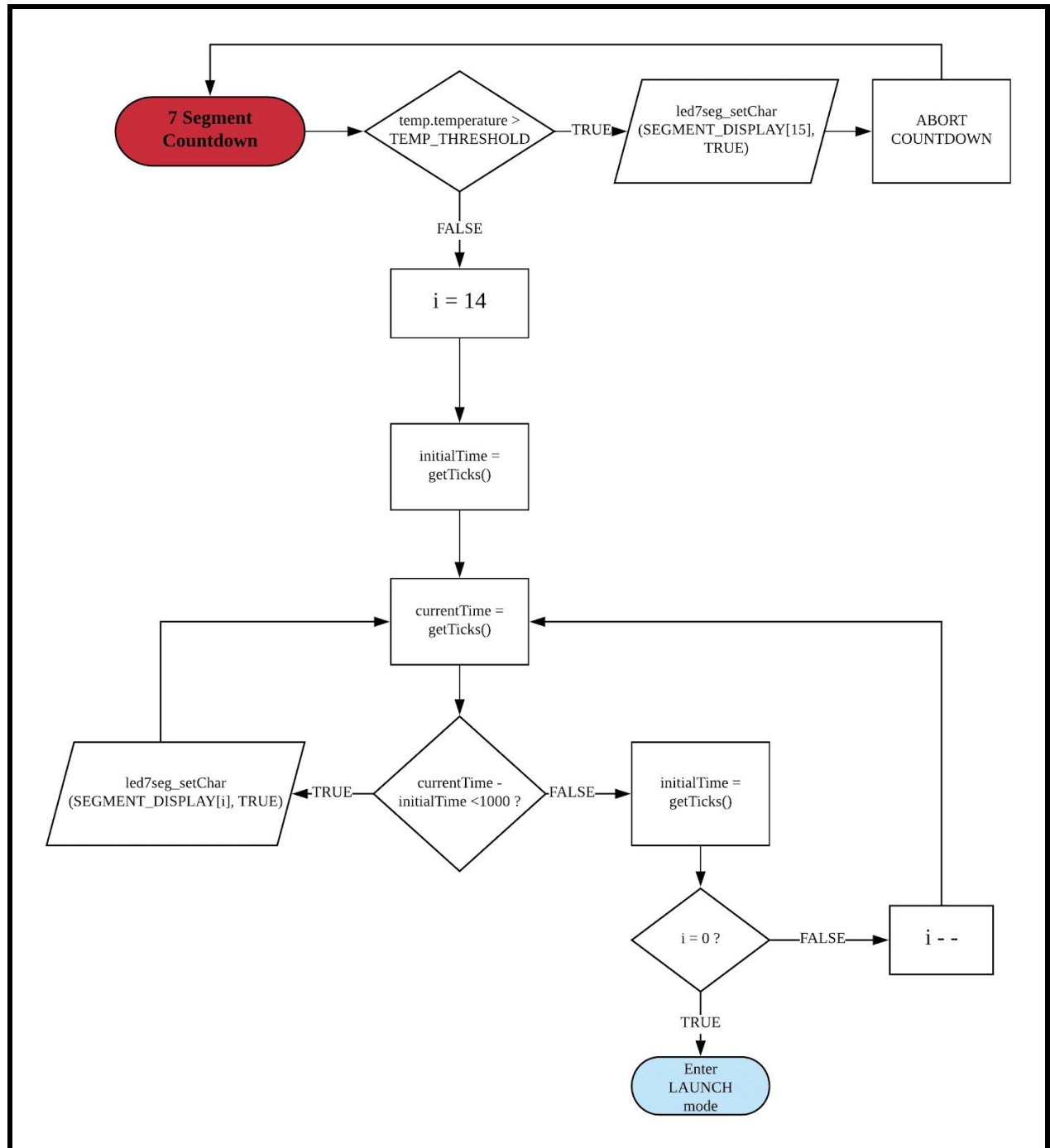
- a. Accelerometer (Rocket Orientation System)
 - b. Light Sensor (Obstacle Avoidance System)
 - c. Temperature Sensor (Fuel Tank Monitoring System)
 - d. LED Array Display
 - e. 7 Segment Display
 - f. OLED Graphics Display
 - g. RGB lights (red, blue only) [Warning indicators (temp, acc)]
 - h. SW3 (used as MODE TOGGLE)
 - i. SW4 (used as CLEAR WARNING)
 - j. UART Terminal program (NUSCloud) [Telemetry messages (data update) sent every 10 seconds]
 - k. Additional Peripherals such as rotary switch or joystick or the potentiometer
2. Implement NUSCloud system using UART medium for data transmissions (Rx and Tx) , both through wired USB and wirelessly via the XBee RF module.
3. Implement useful interrupts for device peripherals that have a significant impact on the system speed and optimisation compared to polling and manage the scheduling of these interrupts through the Nested Vector Interrupt Controller (NVIC) on LPC1769.
4. Introduce a meaningful improvement or enhancement to the system application that was not stated in the requirements that may positively impact the efficiency of the system and emulate the context of NUSpace to a larger degree.

3. System Designs: Flowcharts

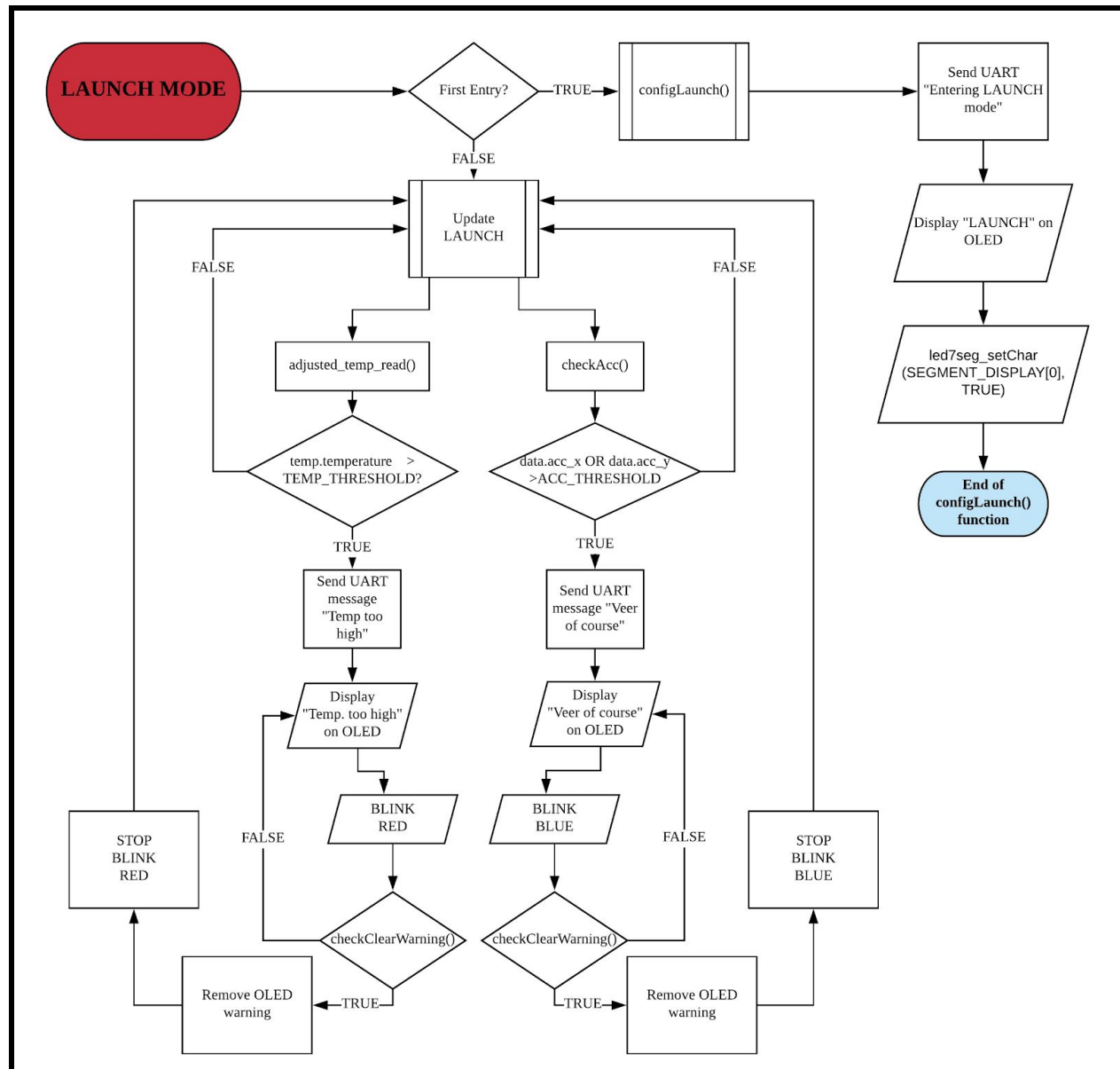
3. 1. Stationary Mode Flowchart



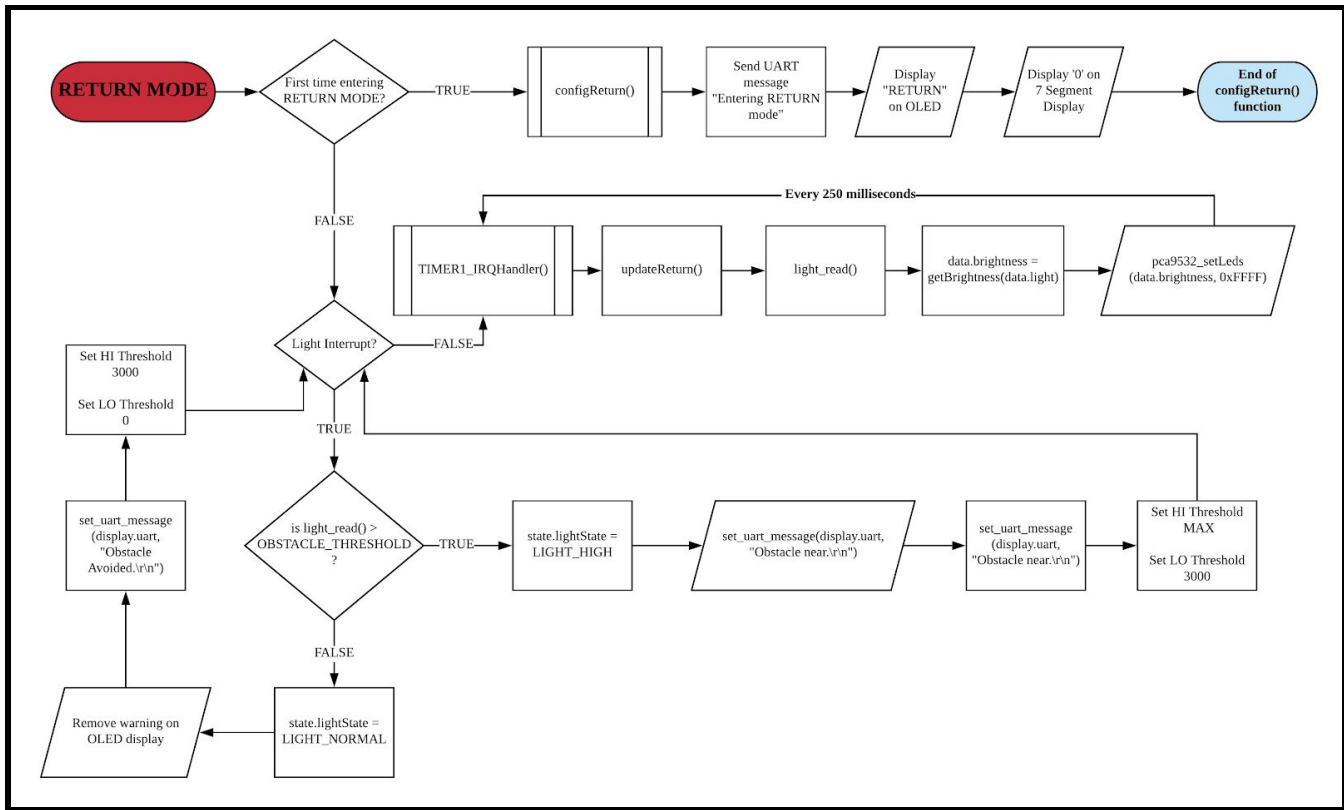
3. 2. Countdown Flowchart



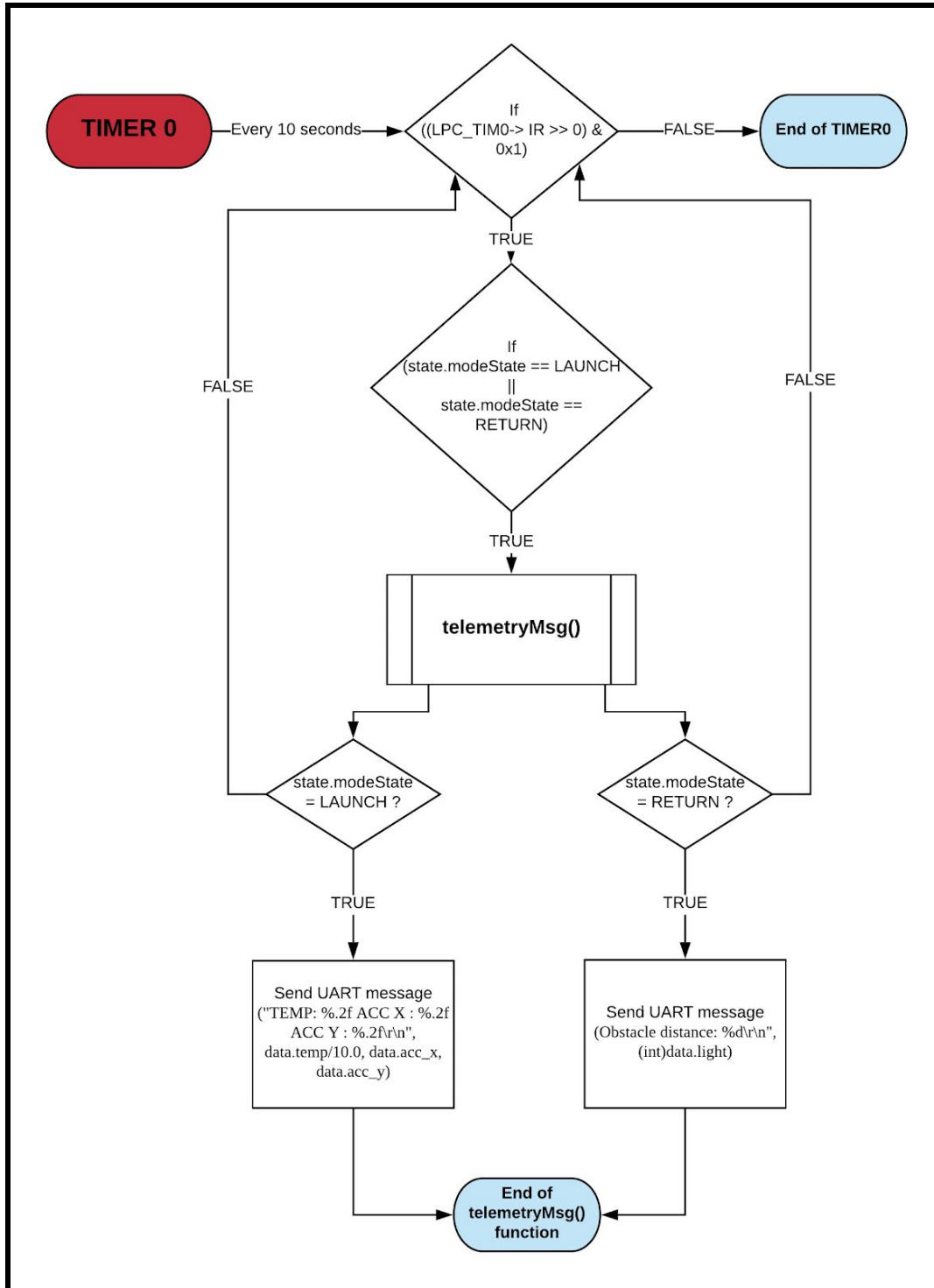
3. 3. Launch Mode Flowchart



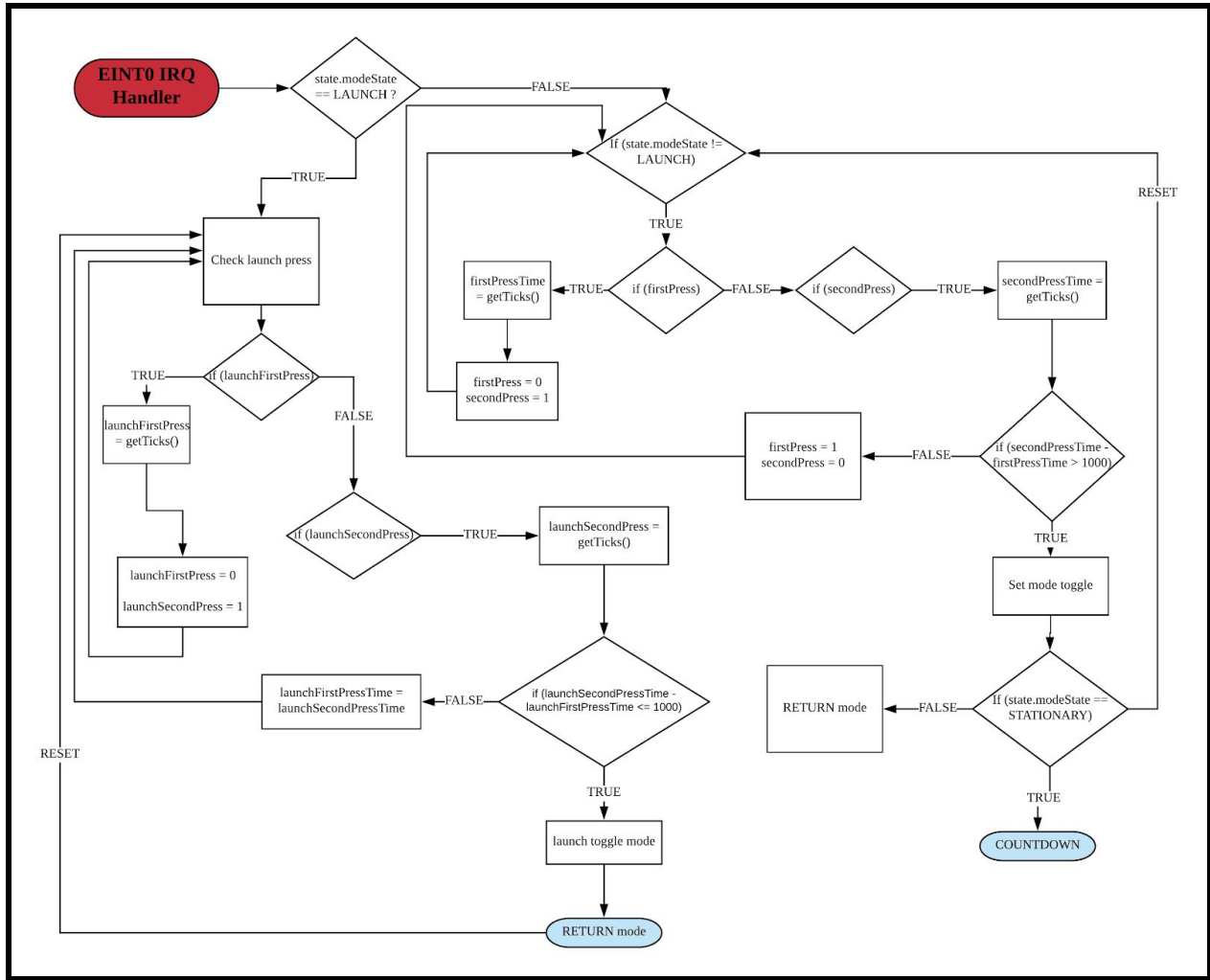
3. 4. Return Mode Flowchart



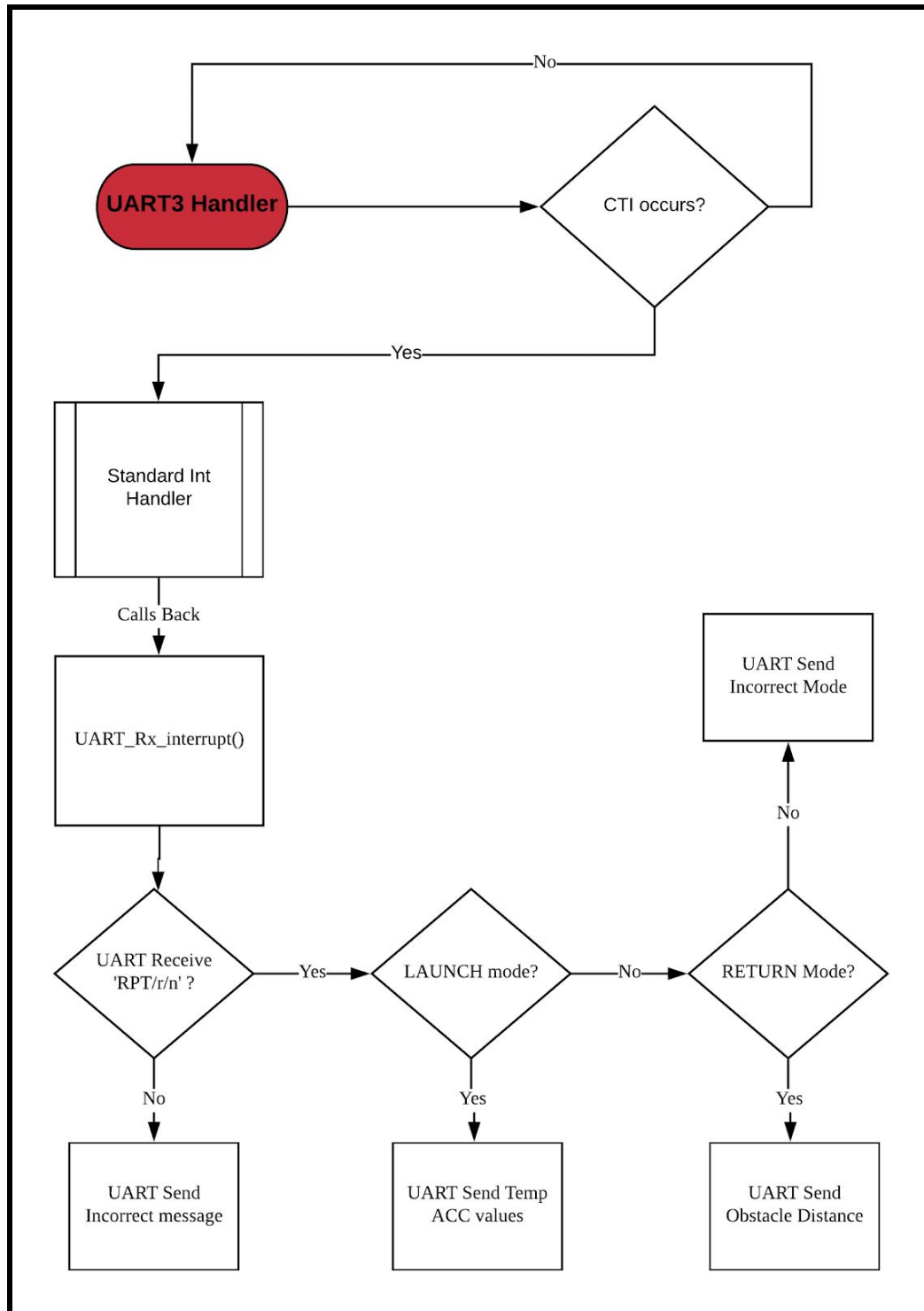
3. 5. TIMER 0 Flowchart



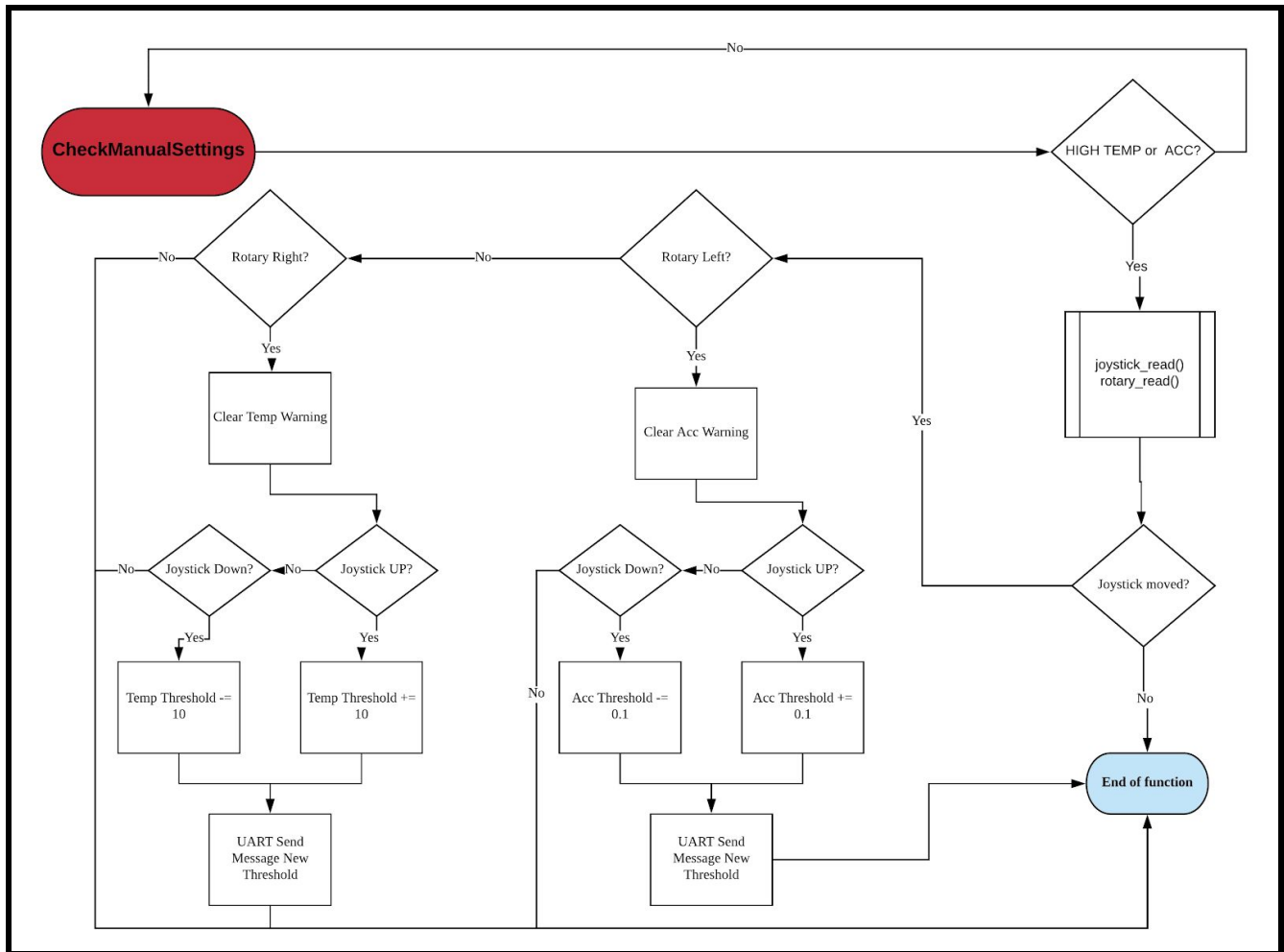
3. 6.EINT0 Flowchart



3. 7. UART3 Flowchart



3. 8. checkManualSettings (Enhancement) Flowchart



4. Implementation of code

The full code used in assignment 2 can be found in Appendix B. Before proceeding with the details in the subsections below, it is highly recommended to go through the full code to get an understanding of the flow of the code and any other functions which are not discussed below. If certain functions do need appear in the code extracts in the subsections below, it means they

are simple enough and do not require any detailed explanations but they can still be found in the Appendix.

4. 1. 1 Setup and initializations

Our setup (full code in Appendix) consists of defined thresholds, enum data types, that we have defined for each mode ,state of sensor, and information required as data, and lastly consists of optimisation flags (identified by lower caps flags with underscore ie; **temp_change**) that help optimise the code while serving as entry and exit flags in functions that only require to be executed upon an event or a trigger such as within the interrupt handler.

Code Extract 4.1.1

```
typedef enum {
    ACC_OFF,
    ACC_NORMAL,
    ACC_HIGH,
} ACC_STATE;

typedef enum {
    TEMP_OFF,
    TEMP_NORMAL,
    TEMP_HIGH,
} TEMP_STATE;

typedef enum {
    LIGHT_OFF,
    LIGHT_NORMAL,
    LIGHT_HIGH,
} LIGHT_STATE;

typedef enum {
    STATIONARY,
    LAUNCH,
    RETURN,
} MODE;

typedef struct {
    MODE modeState;
    ACC_STATE accState;
    TEMP_STATE tempState;
    LIGHT_STATE lightState;
} STATE;

typedef struct {
```

```

    int32_t temperature;
    int halfPeriods;
    uint32_t temperatureT1;
    uint32_t temperatureT2;
} TEMP;

typedef struct {
    float acc_x;
    float acc_y;
    int32_t temp;
    uint32_t light;
    uint16_t brightness;
} DATA;

static STATE state = {STATIONARY, ACC_OFF, TEMP_NORMAL, LIGHT_OFF};
static DATA data = {0, 0, 0, 0};

```

The **STATE** data type is a type-def comprising of 4 other type-defs; the **MODE**, **ACC_STATE**, **TEMP_STATE**, and **LIGHT_STATE** which each have their own enums (**HIGH**, **NORMAL**, and **OFF**) , these types of the sensors allow us to track the state of each sensor in the 3 modes effectively and follow up with functions to their respective necessity. The primary data type we are using is STATE that contains the necessary information of which mode the system is in and what the state of each sensor is in that mode. The first instance of STATE named state is initialized to be in stationary mode, with accelerometer off, temperature sensor on and light sensor off as per the requirements of stationary mode which will be discussed next. Lastly, The **DATA** enum holds the values that are obtained from the sensors and required by the system.

4. 1. 2 Main function and while(1) loop

Code Extract 4.1.2

```

int main (void) {
    SysTick_Config(SystemCoreClock/1000);
    init_all();
    LPC_TIM0->TCR = 1; //start timer0
    LPC_TIM1->TCR = 1; //start timer1
    configStationary();
    telemetry_change = 0;

    while (1)
    {
        telemetryMsg();
        checkManualSettings();
        if (mode_toggle) {
            modeToggle();
            mode_toggle = 0;
        }
    }
}

```



```

    }
    switch (state.modeState) {
    case STATIONARY :
        stationaryMode();
        break;

    case LAUNCH :
        launchMode();
        break;

    case RETURN :
        returnMode();
        break;
    }
}
}

```

In our main function, we kept the code short so as to clearly identify the flow of the code and our system. The sequence in the above code explained is as follows:

1. SysTick is configured to every 1 millisecond (SystemCoreClock = 1 second)
2. Initialize all peripherals necessary including PINSEL and PinConfigs and enable interrupts
3. Begin the timers as soon as code is launched, both TIMER0 and TIMER1
4. Configure system to be in stationary mode : turning off light and acc sensors and setting temp sensors as well as adjusted the OLED (more details explained in *4.3 Stationary Mode*)
5. Clearing the telemetry flag so no message is sent as soon as system startup but only every 10 seconds after the 0 second mark.
6. In the while loop before checking the modes, we check if the telemetry message is to be sent and if the MODE_TOGGLE has been pressed
7. Check which state system is in and execute the relevant functions (explained more the in subsections below)

4. 2 Stationary mode

The stationary mode emulates the rocket before takeoff or after return from space on Earth ground. The necessary code extracts are shown below.

Code Extract 4.2.1

```

void configStationary(void) {

    // Entering for first time into stationary from return mode
    state.lightState = LIGHT_OFF;
}

```

```

state.accState = ACC_OFF;
oled_clearScreen(OLED_COLOR_BLACK);
set_uart_message(display.uart, "Entering STATIONARY Mode\r\n");
UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);

pca9532_setLeds(0x0000, 0xFFFF);

// temp sensor enabled (GPIO to EINT3 interrupt and tempState = TEMP_NORMAL)
temp_enable();

oled_putString(0, 0, (uint8_t *) "STATIONARY", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
oled_putString(0, 10, (uint8_t *) "TEMP" , OLED_COLOR_WHITE, OLED_COLOR_BLACK);
led7seg_setChar(SEGMENT_DISPLAY[15], TRUE);
}

```

4. 2. 1 configStationary():

The system **modeState** is **STATIONARY** either when the system powers on or when the MODE_TOGGLE has been pressed from RETURN mode, upon either scenario, the **configStationary()** function is always called first and only once to configure the system as per stationary mode requirements such as turning off the light sensor, accelerometer and turning on the temperature sensor, it then sends the required message ("Entering STATIONARY Mode") to NUSCloud via UART and displays the necessary labels on the OLED Screen (refer to the code to get the exact labels). We attempted to optimize as far as possible which is why we put the 'TEMP' label on the OLED Screen in configStationary() as it will constantly be shown on screen so we do not need to always call the putString function for that, we only need to update the corresponding value which we did in the **stationaryMode()** function.

Code Extract 4.2.2

```

void stationaryMode() {

    checkClearWarning();

    updateStationary();

    if (message_received) { // Message sent in wrong mode
        set_uart_message(display.uart, "Report not enabled in this mode.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        message_received = 0;
        buf_count = 0;
    }
}

```

```

        if (state.tempState == TEMP_HIGH) {
            if (temp_change) {
                oled_putString(0, 40, (uint8_t *) "Temp. too high", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

                set_uart_message(display.uart, "Temp. too high.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
                temp_change = 0;
            }

            } else if (temp_change) {
                oled_putString(0, 40, (uint8_t *) "                ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
                temp_change = 0;
            }
        }

void updateStationary(void) {
    data.temp = temp.temperature;
    char s[16] = "";
    sprintf(s, "%.2f", data.temp/10.0);
    oled_putString(60, 10, (uint8_t *) s, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
}

```

4. 2. 2 stationaryMode():

After the system has been configured for stationary mode , the **stationaryMode()** is called repeatedly in the while loop until an external change has been made (such as power off or MODE_TOGGLE). The first action this function does is to check if the CLEAR_WARNING has been pressed (clearCheckWarning() can be found in the full code in the Appendix but it merely checks if the SW4 has been pressed and removes any warning state).

It then calls **updateStationary()** which reads the temperature through the EINT3 interrupt where **temp.temperature** is actually a pointer holder for the value from the temperature interrupt which is constantly monitored, if it is above the pre-defined threshold (in the demo we used 270), then it stores the **tempState** variable with a **TEMP_HIGH** and the **temp_change** flag is set to 1. A detailed explanation can be found in the EINT3 Subsection. By using **temp_change** flag we are able to optimize the code by executing the necessary changes as mentioned in this mode only once without any repeated and wasteful actions despite the function being called in the while loop, this increase the overall efficiency of the system. It then displays “Temp. too high” warning on the OLED and the red RGB light will also blink on repeatedly for 333ms. If the warning has been cleared, the **tempState** is set back to **TEMP_NORMAL** and the **temp_change** flag is triggered again to clear the OLED screen.

Lastly, in stationary mode , if anything is typed in NUSCloud Tera Term, then a return message is sent “Report not enabled in this mode”. This is because no transmissions are to occur in stationary mode according to the requirements but we decided to add this corner case message in case anyone does so by accident.

4.3 Launch Mode

Launch mode simulates a scenario of the space shuttle separating from the rocket out of Earth.

Code Extract 4.3.1

```
void countDown(void) {
    int i=14;
    uint32_t currentTime, initialTime;

    initialTime = getTicks();

    while (state.tempState != TEMP_HIGH) {
        currentTime = getTicks();
        if (currentTime - initialTime < 1000) {
            led7seg_setChar(SEGMENT_DISPLAY[i], TRUE);
        }
        else {
            initialTime = getTicks();
            if (i==0) {
                configLaunch();
                state.modeState = LAUNCH;
                break;
            }
            else {
                i--;
            }
        }
    }

    // Abort count down
    if (state.tempState == TEMP_HIGH) {
        led7seg_setChar(SEGMENT_DISPLAY[15], TRUE);
    }
}

}
```

4.3.1 countdown()

Launch mode always follow a countdown sequence first (simulating a real take off from stationary mode). This only occur when there is no temperature warning in stationary mode and the **MODE_TOGGLE** has been pressed. The **countDown()** function essentially displays 'F' to '0' on the 7 segment display for exactly one second. This is done by measuring the time difference between the first character set and the next and updating the **intialTime** and **counter i** variables in each iteration. When setting the character, the **led7seg_setChar()** is used with a self defined character array **SEGMENT_DISPLAY** (that contains only the necessary characters in fixed locations) and setting the second parameter to be **TRUE**(which allows for the raw character to be placed). Once the last character, '0' is set, the function sets the **modeState** datatype to **LAUNCH** enum and breaks from the while loop. However if at anytime the temperature sensor reading goes above the threshold, the sequence is aborted, "F" is redisplayed on the screen and the function exits.

Code Extract 4.3.2

```
void configLaunch(void) {  
    // Entering launch for first time from Stationary  
    set_uart_message(display.uart, "Entering LAUNCH Mode\r\n");  
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);  
    state.accState = ACC_NORMAL;  
  
    oled_clearScreen(OLED_COLOR_BLACK);  
    oled_putString(0, 0, (uint8_t *) "LAUNCH", OLED_COLOR_WHITE, OLED_COLOR_BLACK);  
    oled_putString(0, 10, (uint8_t *) "TEMP", OLED_COLOR_WHITE, OLED_COLOR_BLACK);  
    oled_putString(0, 20, (uint8_t *) "ACC", OLED_COLOR_WHITE, OLED_COLOR_BLACK);  
    led7seg_setChar(SEGMENT_DISPLAY[0], TRUE);  
}
```

4.3.2 configLaunch()

The **configLaunch()** function is always called first and only once to configure the system as per launch mode requirements such as turning on the accelerometer sensor (since temperature sensor is already on from stationary mode, it is not changed in the function) , it then sends the required message ("Entering LAUNCH Mode") to NUSCloud via UART and displays the necessary labels on the OLED Screen (refer to the code to get the exact labels). Similar to stationary mode, all relevant and unchanging labels have been inserted in the config for optimization which updating variables such as the values are in recurring **launchMode()** function. Lastly, it sets the 7 segment display to be '0'.

Code Extract 4.3.3

```
void launchMode() {

    checkAcc();
    updateLaunch();
    checkClearWarning();
    checkLaunchMsg();

    if (state.tempState == TEMP_HIGH) {
        if (temp_change) {
            oled_putString(0, 40, (uint8_t *) "Temp. too high", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

            set_uart_message(display.uart, "Temp. too high.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            temp_change = 0;
        }
    } else if (temp_change) {
        oled_putString(0, 40, (uint8_t *) "                ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
        temp_change = 0;
    }

    if (state.accState == ACC_HIGH) {
        if (acc_change) {
            oled_putString(0, 50, (uint8_t *) "Veer off course", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

            set_uart_message(display.uart, "Veer off course.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            acc_change = 0;
        }
    } else if (acc_change) {
        oled_putString(0, 50, (uint8_t *) "                ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
        acc_change = 0;
    }
}

void checkAcc(void) {
    acc_read(&x, &y, &z);
    x = x + xoff;
    y = y + yoff;

    data.acc_x = fabs (x / ACC_DIV_MEASURE) ;
    data.acc_y = fabs (y / ACC_DIV_MEASURE) ;

    if (state.accState == ACC_NORMAL) {
        if ((data.acc_x > ACC_THRESHOLD) || (data.acc_y > ACC_THRESHOLD)) {
            state.accState = ACC_HIGH;
            acc_change = 1;
        }
    }
}
```

```

    }
}

void updateLaunch(void) {

    data.temp = temp.temperature;
    char s[16] = "";
    sprintf(s, "%.2f", data.temp/10.0);
    oled_putString(60, 10, (uint8_t *) s, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    // after exiting measure, interrupt re-starts again

    char xy[16] = "";
    sprintf(xy, "%.2f, %.2f", data.acc_x, data.acc_y);
    oled_putString(30, 20, (uint8_t *) xy, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
}

void checkLaunchMsg(void) {

    char s[40] = "";
    if (state.tempState == TEMP_NORMAL && state.accState == ACC_NORMAL) {
        if (message_received) {
            message_received = 0;
            set_uart_string();
            if (strcmp(msg_buf, "RPT\r")==0) {
                sprintf(s,"TEMP: %.2f ACC X : %.2f ACC Y : %.2f\r\n",data.temp/10.0,
data.acc_x, data.acc_y );
                set_uart_message(display.uart, s);
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            } else {
                set_uart_message(display.uart, "Unknown message received.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            }
        }
    }
    }else if (message_received) {
        set_uart_message(display.uart, "Cannot report in warning state.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        message_received = 0;
        buf_count = 0;
    }
}

```

4. 3 .3 launchMode()

After the system has been configured for launch mode , the **launchMode()** is called repeatedly in the while loop until an external change has been made (such as power off or MODE_TOGGLE).

The first action executed by the function is **checkAcc()** which reads the accelerometer and accounts for the offset (calibrated in the initialization) and checks if either one of horizontal or vertical axes (x or y) is larger than the threshold. As the value can be negative as the centre from the offset is considered a 0 point and deviations can occur bi-directionally, the **fabs()** function from the math library is used to get the absolute value of the readings. If they are larger than the threshold, then the **accState** is set to **ACC_HIGH** and the **acc_change** optimisation flag is set to 1 which will be picked up in the main launchMode code and issue the relevant response (OLED , RGB blink blue and UART Send). The same is done for **tempState** in the main launchMode() function however there is no requirement for an external temperature reading check as in order to boost the efficiency of the system, the temp is a pointer that is constantly monitored and is already pre-configured from the stationary mode and does not need to be changed (refer to 4.2.2 for explanation of temperature warning and 4.5.2 for the EINT3 temperature interrupt). The values are then updated in the **updateLaunch()** function.

The next action this function does is to check if the CLEAR_WARNING has been pressed (clearCheckWarning() can be found in the full code in the Appendix but it merely checks if the SW4 has been pressed and removes any warning state). If this is triggered, **tempState**, **accState** are set back to **TEMP_NORMAL** and **ACC_NORMAL** respectively, and the **temp_change** , **acc_change** flags are set to 1 to clear the OLED screen.

Lastly, in launch mode , the **checkLaunchMsg()** is called to check if anything is typed in NUSCloud Tera Term, if the word “RPT” is sent in TeraTerm then an update report containing the temperature and accelerometer readings are to NUSCloud. It is important to note that in windows terminal, serial data is sent with the suffixes /r/n to end the string, so “RPT” typed and written is received as “RPT/r/n” in the UART FIFO buffer, more details as to how UART receive interrupt was conducted can be found in 4.5.3. This function also checks if anything aside “RPT” was sent in which case it will respond by sending an Unknown message signal to NUSCloud or if it any message was received in warning mode and the relevant corner case response is sent similarly.

4. 4 Return Mode

Return mode simulates a scenario of the space shuttle returning back to Earth from outer space and likely through the asteroid belt as it may face obstacles in its return path.

Code Extract 4.4.1

```
void configReturn(void) {  
    // Entering Return for first time from launch
```



```

set_uart_message(display.uart, "Entering RETURN Mode\r\n");
UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);

state.lightState = LIGHT_NORMAL;
state.accState = ACC_OFF;
temp_disable(); // Turns off temp sensor and sets tempState to TEMP_OFF

oled_clearScreen(OLED_COLOR_BLACK);
oled_putString(0, 0, (uint8_t *) "RETURN", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
// led7seg_setChar(SEGMENT_DISPLAY[0], TRUE); 7seg is the same from launch mode
}

```

4.4.1 configReturn()

The **configReturn()** function is always called first and only once to configure the system as per return mode requirements such as turning off the accelerometer sensor and the temperature sensor and turning on the light sensor, it then sends the required message ("Entering RETURN Mode") to NUSCloud via UART and displays the necessary the label on the OLED Screen.

Code Extract 4.4.2

```

void returnMode() {

    if (state.lightState == LIGHT_NORMAL) {
        if (obstacle_avoid) {
            obstacle_avoid = 0;
            oled_putString(0, 10, (uint8_t *) "          ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
            set_uart_message(display.uart, "Obstacle Avoided.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        }
    } else if (state.lightState == LIGHT_HIGH) {
        if (light_change) {
            light_change = 0;
            oled_putString(0, 10, (uint8_t *) "Obstacle near", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
            set_uart_message(display.uart, "Obstacle near.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        }
    }

    checkReturnMsg();
}

void checkReturnMsg(void) {

    char x[40] = "";

    if (state.lightState == LIGHT_NORMAL) {
        if (message_received) {
            message_received = 0;

```

```

        set_uart_string();
        if (strcmp(msg_buf, "RPT\r") == 0 ) {
            sprintf(x, "Obstacle distance: %d\r\n", (int)data.light);
            set_uart_message(display.uart, x);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
        } else {
            set_uart_message(display.uart, "Unknown message received.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
        }
    }
} else if (message_received) {
    set_uart_message(display.uart, "Cannot report in warning state.\r\n");
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
    message_received = 0;
    buf_count = 0;
}
}

```

4. 4. 2 returnMode()

After the system has been configured for return mode , the **returnMode()** is called repeatedly in the while loop until an external change has been made (such as power off or MODE_TOGGLE).

Similar to the other two modes, **returnMode()** merely checks if the **lightState** is **LIGHT_HIGH** and if the **light_change** or **obstacle_avoid** optimisation flags have been set to 1 and then issues the relevant warning messages accordingly. However unlike the other two modes, the function **updateReturn()** is not called in this function. This is because it is called in **TIMER1** interrupt instead which reads the light sensor and updates the 16 LED Array every 250 milliseconds. We did this so as to reduce amount of time spent by returnMode() to read light sensor and go through the selection statements in the 16 LED Array function **getBrightness()**. More information on this can be found in 4.5.5.

Lastly, like launch mode , in return mode; the **checkReturnMsg()** is called to check if anything is typed in NUSCloud Tera Term, if the word “**RPT**” is sent in TeraTerm then an update report containing the temperature and accelerometer readings are to NUSCloud. More details as to how UART receive interrupt was conducted can be found in 4.5.3. This function also checks if anything aside “**RPT**” was sent in which case it will respond by sending an Unknown message signal to NUSCloud or if it any message was received in warning mode and the relevant corner case response is sent similarly.

4. 5 Interrupts

In order to increase the efficiency and responsiveness of our system , as a space flight system should be, we have used 5 interrupts in our entire system that make significant improvements compared to the alternative that is polling in both time reduction and efficient resource allocation. Each of these 5 interrupts are explained more in detail in the subsections below.

4. 5. 1 EINT0 : MODE_TOGGLE

The MODE_TOGGLE (SW3) was read using an interrupt instead of polling and instead of EINT3 GPIO interrupt it was used in EINT0 as it was an available option which allowed us to configure EINT0 to be the highest preempt priority interrupt and this ensured that the responsiveness of MODE_TOGGLE was as maximum.

Code Extract 4.5.1.1

```
void EINT0_IRQHandler(void) {  
  
    // SW3 External Interrupt  
    if ((LPC_SC -> EXTINT) & 0x01) {  
  
        // Every other mode except in launch (1 press is mode toggle)  
        if (state.modeState != LAUNCH) {  
  
            if (firstPress) {  
                firstPressTime = getTicks(); // Record time when first press  
                firstPress = 0;  
                secondPress = 1;  
            } else if (secondPress) {  
                secondPressTime = getTicks();  
                secondPress = 0;  
                firstPress = 1;  
            }  
  
            // If the delay between first and second press is larger than 1 second  
            if (secondPressTime == 0) {  
                mode_toggle = 1;  
            } else if ((int)(secondPressTime - firstPressTime) > 1000) {  
                mode_toggle = 1;  
            } else { // Reset sw3 to prepare for next presses  
                firstPress = 1;  
                firstPressTime = 0;  
                secondPress = 0;  
                secondPressTime = 0;  
            }  
        }  
    }  
}
```

```

    }

    } else {

        if (launchFirstPress) {
            launchFirstPressTime = getTicks();
            launchFirstPress = 0;
            launchSecondPress = 1;
        } else if (launchSecondPress) {
            launchSecondPressTime = getTicks();
            launchSecondPress = 0;
            launchFirstPress = 1;
        }

        if (launchSecondPressTime == 0) {
            launch_toggle = 0;

        } else if (launchSecondPressTime - launchFirstPressTime <= 1000) {
            mode_toggle = 1;
            launch_toggle = 1;
        } else {
            launchFirstPressTime = launchSecondPressTime;
            launchSecondPress = 1;
            launchFirstPress = 0;
        }
    }

    // Reset sw3 to prepare for next presses
    if (launch_toggle) {
        launchFirstPress = 1;
        launchFirstPressTime = 0;
        launchSecondPress = 0;
        launchSecondPressTime = 0;
        firstPress = 1;
        firstPressTime = 0;
        secondPress = 0;
        secondPressTime = 0;
    }
}

// clear interrupt
LPC_SC->EXTINT |= (1<<0);
}

```

4. 5. 1. 1 EINT0_IRQHandler

EINT0_IRQHandler() is triggered each time the MODE_TOGGLE (SW3) has been pressed on the baseboard. Though the ISR looks lengthy, it is actually optimised with simple flags so as to debounce SW3 effectively ensuring a swift one trigger in one second (regardless of how many

times it was actually pressed within a second) and also meeting the launch to return requirements of only registering if **MODE_TOGGLE** was pressed twice within a second.

The first debounce of only press in a second was checked in every mode except **modeState LAUNCH** (due to the requirement as stated above) , it does by getting the time of the **firstPress** and the **secondPress** and alternating these flags between them. It then checks if the time difference between **secondPressTime** and **firstPressTime** is larger than a second before enabling the **mode_toggle** flag. If it is not, reset the switch parameters and prepare for next presses. (This allows for maximum effectiveness to SW3 in the sense that no matter how many times you press it within a second, it always resets without counting those number of presses as valid and the next press will **mode_toggle** if no other presses are made within the second).

In a similar fashion to the logic as stated above, in the launch mode, the second requirement was met but the difference being that instead of a larger than a second, the time difference between **launchSecondPressTime** and **launchFirstPressTime** (two different variables) now has to be lesser than a second to be valid and set the flag **launch_toggle** to 1. If not, instead of resetting the switch like in the previous method, set the second press as the first press. (ie **launchFirstPressTime = launchSecondPressTime**) . This improves the effectiveness and practicality of SW3 in launch mode as it eliminates the need for the user to press a third time to counteract the first two presses. Now if someone presses more than twice but the difference between the first launch press and second launch press is larger than one second and if the difference between the second press and third press is less than one second, the combination of the last two sets of presses will be registered as valid and the first press will be discarded, saving valuable time for a space flight system and improving the overall speeds of **MODE_TOGGLE**. The actual change of modes can be seen in the next subsection **modeToggle()**. Lastly the interrupt is cleared.

Code Extract 4.5.1.2

```
void modeToggle(void) {  
  
    if ((state.modeState == STATIONARY) && (state.tempState != TEMP_HIGH)) {  
        countDown();  
        // if second mode toggle is recorded and within 1 second of the first mode toggle  
    } else if ((state.modeState == LAUNCH) && (launch_toggle)) {  
        configReturn();  
        state.modeState = RETURN;  
        launch_toggle = 0;  
  
    } else if (state.modeState == RETURN) {  
        checkLightWarning();  
        configStationary();  
    }  
}
```

```

        state.modeState = STATIONARY;
    }
}

```

4. 5. 1. 2 modeToggle()

The **modeToggle()** function is checked every iteration in the main while loop (refer to 4.1.2) and if the **mode_toggle** flag is set to 1 from the EINT0 Handler (above subsection) , it is entered. This function changes the **modeState** of the system depending on the current modeState and the requirements of the system. It is also noteworthy that a transition from launch mode to return mode will only be made if the **launch_toggle** is also set in the EINT0 handler.

4. 5. 2 EINT3 Interrupt

This interrupt was used for the temperature sensor reading and light interrupt threshold (both high and low). It is the shared interrupt for the GPIO devices on the baseboard.

Code Extract 4.5.2

```

void EINT3_IRQHandler(void)
{
    // Light Sensor Interrupt (Obstacle detection)
    if ((LPC_GPIOINT->IO2IntStatF >> 5) & 0x1) {
        if (state.lightState == LIGHT_NORMAL) {
            state.lightState = LIGHT_HIGH;
            light_change = 1;
            light_setHiThreshold(MAX_LIGHT);
            light_setLoThreshold(OBSTACLE_THRESHOLD);
        }
        else if (state.lightState == LIGHT_HIGH){
            state.lightState = LIGHT_NORMAL;
            obstacle_avoid = 1;
            light_setHiThreshold(OBSTACLE_THRESHOLD);
            light_setLoThreshold(0);
        }

        light_clearIrqStatus();
        LPC_GPIOINT->IO2IntClr = (1 << 5);
    }

    // Temp sensor Interrupt (Monitor Fuel Tank)
    if ((LPC_GPIOINT->IO0IntStatF >> 2) & 0x1) {

```

```

        adjusted_temp_read(&temp);

        if ((state.tempState == TEMP_NORMAL) && (temp.temperature > TEMP_THRESHOLD)) {
            state.tempState = TEMP_HIGH;
            temp_change = 1;
        }
        LPC_GPIOINT->IO0IntClr |= (1 << 2);
    }
}

```

4. 5. 2. 1 Light Sensor Interrupt

The first action in the light sensor interrupt section of EINT3 is to check if the interrupt is indeed from the light sensor pin, as multiple interrupts could be configured in EINT3. This is done by checking if the interrupt is triggered by **Port 2 Pin 5** (light interrupt pin) by checking the interrupt status register using **IO2IntStatF >> 5** . This detects if a falling edge from the light interrupt pin has occurred. If this is the first time the interrupt has triggered the **lightState** of the system should be **LIGHT_NORMAL** in the return mode and it will enter the first if condition. It then enables the **light_change** flag, sets lightState to **LIGHT_HIGH** and proceeds to **light_setHiThreshold(MAX_LIGHT)** and **light_setLoThrehsold (OBSTACLE_THRESHOLD)**. What this is does is it configures the next light interrupt to happen only if the light sensor detects a reading above max light (we set it to 16000 so it close to impossible for this to trigger the interrupt) or if the light sensor readings drops below the 3000 Threshold (as soon as the light source is moved) which would indicate that the system is no longer in a warning and since the lightState is now in **LIGHT_HIGH** , only the second else if loop if the interrupt is triggered the next time, which in turn sets the lighState back to **LIGHT_NORMAL** and enables the obstacle_avoid flag and then sets the high and low thresholds (3000 and 0) of the interrupt back to the original values required to be checked for the obstacle warning state. The final step each time the interrupt is triggered is **light_clearIrqStatus()** which clears the interrupt within the light sensor itself and then we clear the EINT3 interrupt from the light sensor pin as well.

Code Extract 4.5.2.2

```

void temp_disable(void) {
    LPC_GPIOINT->IO0IntEnF &= ~(1<<2);
    LPC_GPIOINT->IO0IntClr |= (1 << 2);
    state.tempState = TEMP_OFF;
}

```

```

void temp_enable(void) {
    LPC_GPIOINT->I00IntEnF |= (1<<2);
    temp.halfPeriods = 0;
    temp.temperature = 0;
    temp.temperatureT1 = 0;
    temp.temperatureT2 = 0;
    state.tempState = TEMP_NORMAL;
}

void adjusted_temp_read(TEMP* temp) {
    if (!temp->temperatureT1 && !temp->temperatureT2) {
        temp->temperatureT1 = getTicks();
    } else if (temp->temperatureT1 && !temp->temperatureT2) {
        temp->halfPeriods++;
        if (temp->halfPeriods == TEMP_NUM_HALF_PERIODS/2) {
            temp->temperatureT2 = getTicks();
            if (temp->temperatureT2 > temp->temperatureT1) {
                temp->temperatureT2 = temp->temperatureT2 - temp->temperatureT1;
            }
            else {
                temp->temperatureT2 = (0xFFFFFFFF - temp->temperatureT1 + 1) +
temp->temperatureT2;
            }
            temp->temperature = ((2*1000*temp->temperatureT2) /
(TEMP_NUM_HALF_PERIODS*TEMP_SCALAR_DIV10) - 2731);
            temp->temperatureT1 = 0;
            temp->temperatureT2 = 0;
            temp->halfPeriods = 0;
        }
    }
}

```

4. 5. 2. 2 Temperature Sensor Interrupt

The temperature interrupt is detected in **EINT3_IRQHandler** by checking if a falling edge has been detected from the temperature sensor **Port 2 Pin 0**, every time this happens , the function **adjusted_temp_read()** is called with the pointer of data type **TEMP** , as mentioned in section 4. 1. 1 Setup , the TEMP data type is a global pointer that contains the necessary data for the temperature reading to be calculated in our adjusted_temp_read() function. The adjusted_temp_read() function essentially measures the difference in periods of **temperatureT1** and **temperatureT2** in the sampling period of 170 milliseconds. The temperature is then calculated by measuring the change in the periods of which corresponds to any external change in temperature by utilizing the defined formula in the temperature sensor library

($2*1000*temp \rightarrow temperatureT2 / TEMP_NUM_HALF_PERIODS*TEMP_SCALAR_DIV0$) - 2731)
 . This returns the value of temperature in degrees but with a multiple of 10, eg 28 degrees is returned as 280. Which is why **data.temp** is divided by 10.0 when comparing it to the **TEMP_THRESHOLD** value in the EINT3_IRQHandler, if it is larger than the threshold, and the **tempState** is **TEMP_NORMAL** (indicating a first temperature warning trigger), the temp_change flag is set to 1 and the tempState is assigned to **TEMP_HIGH**. Lastly, the temperature interrupt is cleared from EINT3 before exiting the ISR.

4. 5. 3 UART3 Interrupt

The UART3_IRQHandler() was used for receiving transmissions from NUSCloud, this was done to give an immediate reply to NUSCloud instead of depending on polling within the launch mode and return mode where messages are supposed to be received. Overall using an interrupt gave a more efficient and swift response to any message sent from NUSCloud to NUSpace

Code Extract 4.5.3.1

```
void UART3_IRQHandler(void) {
    UART3_StdIntHandler();
}

void UART3_Rx_Interrupt(void) {

    if(UART_Receive(LPC_UART3, &msg_buf[buf_count], 1, NONE_BLOCKING) == 1) {
        if(msg_buf[buf_count] == '\n'){
            message_received = 1;
        }

        buf_count++;

        if (buf_count == 64) {
            buf_count = 0;
        }
    }
}
```

4. 5. 3. 1 UART3_IRQHandler() & Call back function

In the **init_uart()**(refer to full code segment in Appendix B), we have enabled the FIFO configuration and a call back function each time Rx occurs. Hence in the **UART3_IRQHanlder()** only the given **UART3_StdIntHandler()** function is called which in turn will call back our self defined function **UART3_Rx_Interrupt()**.

UART3_Rx_Interrupt() checks if UART_Receive has occurred from the LPC_UART3 port to our message buffer ; **msg_buf[buf_count]** which takes in one character at a time with **NONE_BLOCKING** each time CTI occurs. If this happens, the buffer is checked for the terminating character '\n' from NUSCloud, indicating that a complete message has been sent and the **message_received** flag is set to 1. Lastly, the **buf_count** is increment for each character in the FIFO left to be received and if the limit of 64 of the buffer has been reached then the index of the buffer is reset to 1.

4. 5. 4 TIMER0 Interrupt

The TIMER0 interrupt was used for the periodic sending of data update report every 10s seconds to NUSCloud when the system is in launch or in return mode. By doing so, we eliminated the need for extra system time variables and checking their differences and we used the available timer configuration to exactly time transmissions.

Code Extract 4.5.4.1

```
void init_timer0(void) {  
    LPC_SC->PCONP |= (1<<1); //Power up TIMER0  
    LPC_SC->PCLKSEL0 |= (0b01 << 2);  
    LPC_TIM0->TCR = 2; // reset & hold TIMER0  
    LPC_TIM0->MR0 = SystemCoreClock*10; // Interruption every 10seconds  
    LPC_TIM0->MCR |= 1<<0; // Interruption on Match0 compare  
    LPC_TIM0->MCR |= 1<<1; // reset TIMER0 on match 0  
}
```

4. 5. 4 .1 Initialization of TIMER0 Interrupt

Following the given settings from the LPC17xx user manual we have configured timer0 in the **init_timer0()** such that its enabled and interrupt every **10 seconds** (SystemCoreClock is 1 second) and interrupts on **match0** comparison and once enabled we reset timer0 and hold it until the active command command is issues , which is in the main function (refer to 4.1.2).

Code Extract 4.5.4.2

```
void TIMER0_IRQHandler(void) {
    // check if TIMER0 M0 interrupt
    if ((LPC_TIMER0->IR >> 0) & 0x1) {
        if (state.modeState == LAUNCH || state.modeState == RETURN) {
            telemetry_change = 1; // every 10seconds flag for telemetry data to be sent
        }
        LPC_TIMER0->IR |= (1<<0); // clear TIMER0 interrupt
    }
}

void telemetryMsg(void) {
    char s[40] = "";
    if (telemetry_change) {
        if (state.modeState == LAUNCH) {
            sprintf(s, "TEMP: %.2f ACC X : %.2f ACC Y : %.2f\r\n", data.temp/10.0,
data.acc_x, data.acc_y);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            telemetry_change = 0;
        }

        if (state.modeState == RETURN) {
            sprintf(s, "Obstacle distance: %d\r\n", (int)data.light);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            telemetry_change = 0;
        }
    }
}
```

4. 5. 4 .2 TIMER0_IRQHandler() & telemetryMsg()

The **TIMER0_IRQHandler()** itself is kept minimum and simply checks if the system is in launch mode or return mode (the two modes in which periodic report update is required) and enables the **telemetry_change** flag if the system is in either of the modes every 10 seconds.

The **telemetry_change** flag is used to check **telemetryMsg()** in the while loop from the main function (refer to 4.1.2). **telemetryMsg()** checks if the flag is enabled and sends the relevant data to NUSCloud depending on the system mode. Finally it resets the **telemetry_change** flag back to 0 so as to ensure only 1 update is sent every 10 seconds.

4. 5. 5 TIMER1 Interrupt

The TIMER1 interrupt was used to update return mode values from the light sensor and get the brightness value to set the 16 LED array.

Code Extract 4.5.5.1

```
void init_timer1(void) {
    LPC_SC->PCONP |= (1 << 2); // power up TIMER1
    // PCLK_peripheral = SystemCoreClock
    LPC_SC->PCLKSEL0 |= (0b01 << 4);
    LPC_TIM1->TCR = 2; // reset & hold TIMER1
    LPC_TIM1->MR0 = (SystemCoreClock/4); // interrupt every 250 millisecond
    LPC_TIM1->MCR |= 1 << 0; // interrupt on Match1 compare
    LPC_TIM1->MCR |= 1 << 1; // reset TIMER1 on Match 1
}
```

4. 5. 5 .1 Initialization of TIMER1 Interrupt

The initialization of TIMER1 was done almost exactly like TIMER0 (refer to 4.5.4.1) but the only difference being that the interrupt rate was set to every 250 milliseconds (SystemCoreClock / 4).

Code Extract 4.5.5.2

```
void TIMER1_IRQHandler(void) {
    // check if TIMER0 M0 interrupt
    if ((LPC_TIM1->IR >> 0) & 0x1) {
        if (state.modeState == RETURN) {
            updateReturn(); // every 10seconds flag for telemetry data to be sent
        }
        LPC_TIM1->IR |= (1<<0); // clear TIMER0 interrupt
    }
}

void updateReturn(void) {
    data.light = light_read();
    data.brightness = getBrightness(data.light);
    pca9532_setLeds(data.brightness, 0xFFFF);
}

static uint16_t getBrightness(uint32_t light) {
    if (light < 242) {
        return 0x0000;
    } else if (light < 484) { // 1
        return 0x0003;
    }
}
```

```

    } else if (light < 726) { // 2
        return 0x0007;
    } else if (light < 968) { // 3
        return 0x000F;
    } else if (light < 1210) { // 4
        return 0x001F;
    } else if (light < 1452) { // 5
        return 0x003F;
    } else if (light < 1694) { // 6
        return 0x007F;
    } else if (light < 1936) { // 7
        return 0x00FF;
    } else if (light < 2178) { // 8
        return 0x01FF;
    } else if (light < 2420) { // 9
        return 0x03FF;
    } else if (light < 2662) { // 10
        return 0x07FF;
    } else if (light < 2904) { // 11
        return 0x0FFF;
    } else if (light < 3146) { // 12
        return 0x1FFF;
    } else if (light < 3388) { // 13
        return 0x3FFF;
    } else if (light < 3630) { // 14
        return 0x7FFF;
    } else {
        return 0xFFFF;
    }
}

```

4. 5. 4 .2 TIMER0_IRQHandler() & updateReturn()

TIMER1 ISR checks if the system mode is in return every 250 milliseconds and calls the the **updateReturn()** function if it is.

The updateReturn function reads from the light sensor and converts the uint32_t light intensity value to a corresponding uint16_t value, stored in **data.brightness** why the helper function **getBrightness()** ,which are the parameters for **pca9532_setLeds()**. By using an interrupt instead of continually polling in the **returnMode()** function, we are able to minimize the amount of time spent reading from light sensor and setting the 16 LED Array.

4. 5. 6 Interrupt Priority Setting

Code Extract 4.5.6

```
NVIC_SetPriorityGrouping(5);

NVIC_SetPriority(SysTick_IRQn, 0x00);
NVIC_SetPriority(EINT0_IRQn, 0x40);
NVIC_SetPriority(TIMER1_IRQn, 0x48);
NVIC_SetPriority(TIMER0_IRQn, 0x50);
NVIC_SetPriority(EINT3_IRQn, 0x80);
NVIC_SetPriority(UART3_IRQn, 0x88);

NVIC_ClearPendingIRQ(EINT0_IRQn);
NVIC_ClearPendingIRQ(TIMER0_IRQn);
NVIC_ClearPendingIRQ(TIMER1_IRQn);
NVIC_ClearPendingIRQ(EINT3_IRQn);
NVIC_ClearPendingIRQ(UART3_IRQn);

NVIC_EnableIRQ(EINT0_IRQn);
NVIC_EnableIRQ(TIMER1_IRQn);
NVIC_EnableIRQ(TIMER0_IRQn);
NVIC_EnableIRQ(EINT3_IRQn);
NVIC_EnableIRQ(UART3_IRQn);
```

Code Extract 4.5.6 can be found in the main **init_all()** from the full code segment (Appendix B) , here we used **NVIC_SetPriorityGrouping(5)** which allows us to use the first two MSB bits for preempt priority level setting and the next 3 bits for sub-priority level setting as LPC1769 only has 5 bits enabled and the remaining 3 LSB bits are unused. We give **SysTick_IRQn** the highest priority overall because we do not want any other interrupt interfering with the SysTick function as it is the most important function to keep track of the system timings and cannot afford any delays. Followed by the **EINT0_IRQn** who has the same preempt priority has **TIMER1_IRQn** but a high sub priority level , this is so that **MODE_TOGGLE** can be done effectively and have the fastest response in the system. It is also important to note that we gave TIMER1 a similar preempt priority as we realized there was a system clash between TIMER1 and EINT3 ; if the light sensor interrupt interrupted our system while the system was in the light_read() function, for an unknown reason, our board would hang which we suspect might be due to overriding I²C commands within the light sensor register. Next we gave **EINT3_IRQn** the second last priority as this was the one ISR that would delay most of the ISRs (due to temperature interrupt and light interrupt) and the system was still very responsive in terms of updating the temperature and light interrupts as necessary , hence we found this to be a

balanced tradeoff between performance and functionality. Lastly, we gave **UART3_IRQn** the lowest priority as we felt it is was not as necessary as the other interrupts and we were given a time range of 1 second to complete UART3 transmission which was enough time for all the other ISRs to complete in the scenario where UART3 was interrupted, furthermore the UART3_IRQn used a callback function with the FIFO structure when responding to CTI with each trailing character and the **NONE_BLOCKING** mode so we felt it would serve no purpose to give UART3 ISR any higher priority setting as it would have completed in approximately the same amount of time.

4. 6 Application Logic Enhancement

Our enhancement is in close relation to the objective that we set out for ourselves (refer to 2. Objectives) , as we feel this enhancement boosts the context of NUSpace and is has a value to the system. The enhancement is a custom control function, **checkManualSettings()**, to the pilot of NUSpace whereby he/she can adjust the thresholds of the fuel tank and the object orientation in real time if he/she feels confident enough that can manually control the aircraft better than the system. It uses a combination of **rotary switch** and **joystick** to control the thresholds , more details are explained below (4.6.1 and 4.6.2) .

Even without any hypothetical imagination, our enhancement also has a very practical purpose. For example, if the temperature threshold was set to 27 degrees but the environment you are in is constantly above 27, like 29 , one would have to resort to changing the threshold in the code in LPCXpresso and reuploading the code , but with our enhancement you can clear the warning and use the rotary switch to increase or decrease threshold real time such that now you can make it 30 degrees so the temperature warning will not constantly trigger. How this is done will be explained in the code and baseboard analysis below.

Code Extract 4.6.1

```
void checkManualSettings(void) {

    uint8_t rState = 0; //rotary state
    uint8_t jState = 0; //joystick state

    rState = rotary_read();
    jState = joystick_read();

    char s[40] = "";

    if (state.tempState == TEMP_HIGH || state.accState == ACC_HIGH) {

        if (jState) {
            if(rState == ROTARY_LEFT) {
                if (state.accState == ACC_HIGH) {
                    state.accState = ACC_NORMAL;
                    acc_change = 1;
                }
            }
        }
    }
}
```

```

    }
    if (jState == JOYSTICK_UP) {
        if (ACC_THRESHOLD <= 0.7) {
            ACC_THRESHOLD += 0.1;
        }
    } else if (jState == JOYSTICK_DOWN) {
        if (ACC_THRESHOLD >= 0.3) {
            ACC_THRESHOLD -= 0.1;
        }
    }
    sprintf(s, "New ACC_THRESHOLD = %.2f\r\n", ACC_THRESHOLD);
    set_uart_message(display.uart, s);
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
} else if (rState == ROTARY_RIGHT) {

    if (state.tempState == TEMP_HIGH) {
        state.tempState = TEMP_NORMAL;
        temp_change = 1;
    }

    if (jState == JOYSTICK_UP) {
        if (TEMP_THRESHOLD <= 310) {
            TEMP_THRESHOLD += 10;
        }
    } else if (jState == JOYSTICK_DOWN) {
        if (TEMP_THRESHOLD >= 260) {
            TEMP_THRESHOLD -= 10;
        }
    }
    sprintf(s, "New TEMP_THRESHOLD = %d\r\n", TEMP_THRESHOLD);
    set_uart_message(display.uart, s);
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
}
}
}
}

```

4. 6. 1 checkManualSettings()

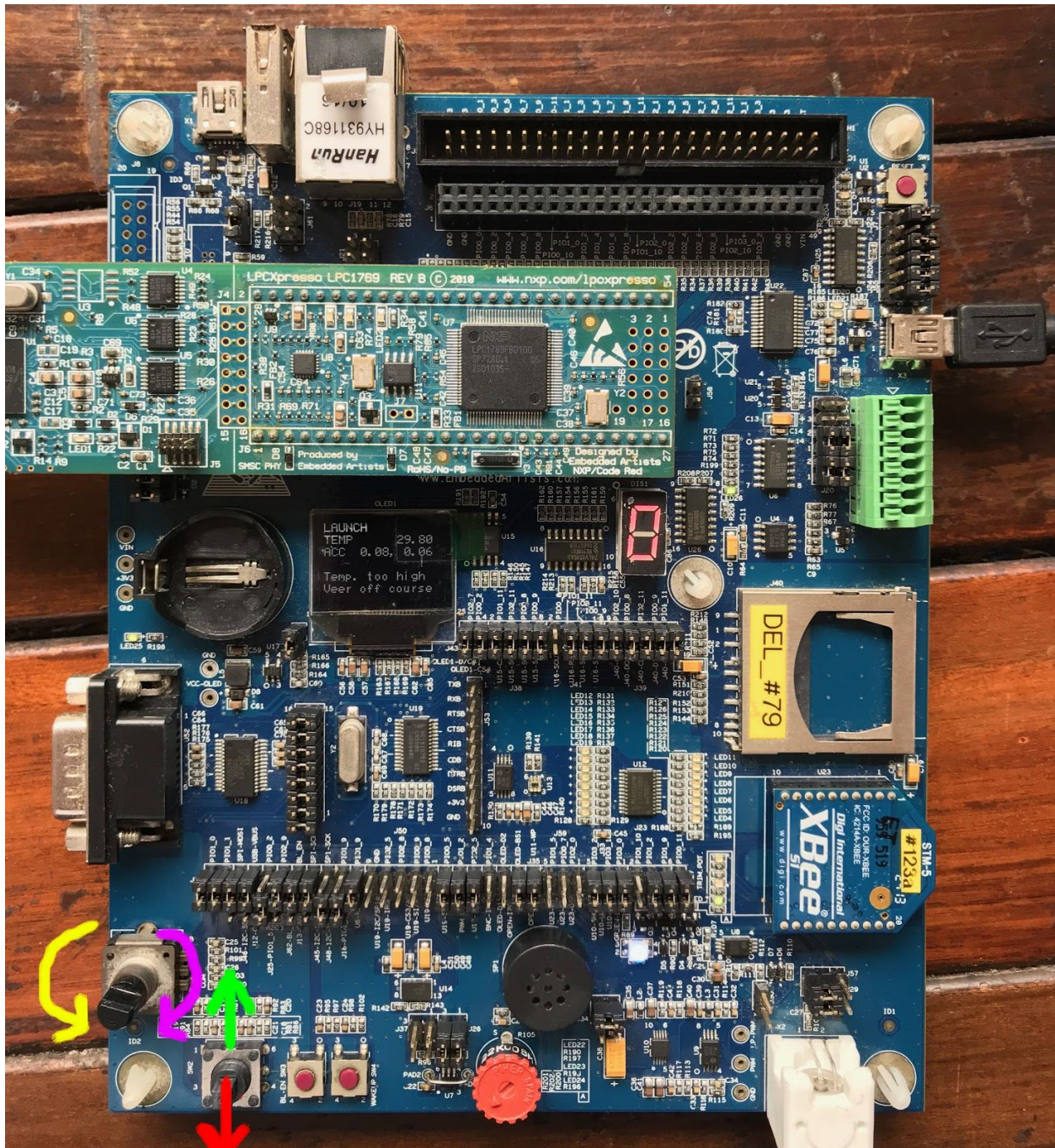
checkManualSettings() is constantly checking the while(1) loop from the main function (refer to section 4.1.2 for main function code). It can only be executed in the warning state: either temperature or accelerometer warnings , it cannot be triggered from the obstacle warning state as we felt the light interrupt already controls the clearing of warning and setting the thresholds hence it would be redundant. By using a combination of the rotary switch and joystick we are able to execute the command exactly once without the worry of debouncing the joystick.

Furthermore it also adds an element of play for the user as they will have coordinate the two peripherals correctly, somewhat emulating the difficulty in piloting an aircraft (albeit much simpler).

The function always check to see if the joystick has been moved at all first which is the first parameter , ie, **if (jState)** where jState is the reading from the joystick and is a numerical value as long as the joystick is in the centre or unmoved. The joystick only has 2 parameters in our function, it is set to be either **JOYSTICK_UP** or **JOYSTICK_DOWN** which represent an increase or decrease in threshold respectively. Once the joystick movement has been recognized the next parameter that determines whether the temperature or accelerometer threshold is to be altered is the rotary switch reading in the **rState** variable. rState also only has two parameters ; either **ROTARY_LEFT** or **ROTARY_RIGHT**, whereby left indicates in a request to change accelerometer warning threshold and right indicates a request to change temperature threshold.

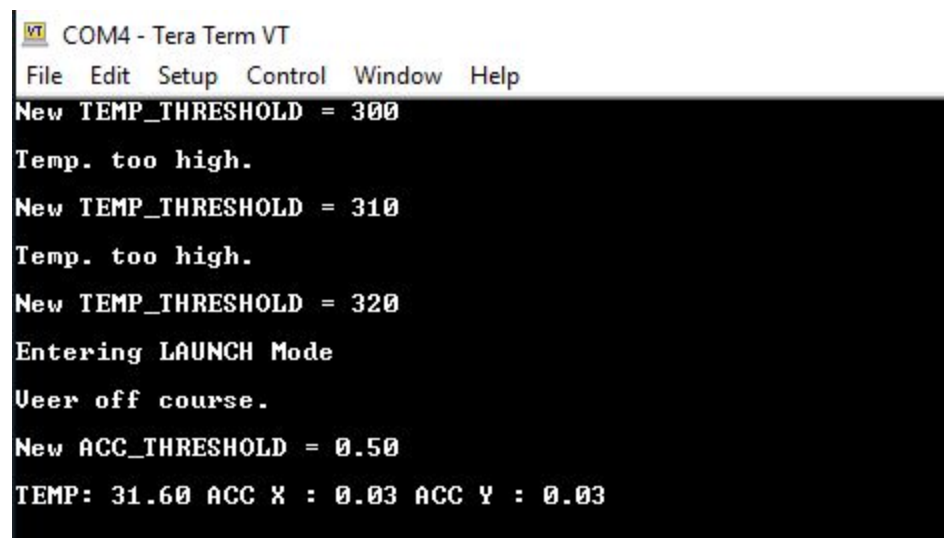
Each time the checkManualSettings() is in use (only in warning state) and a combination of joystick and rotary switch is externally controlled, the first step the function does is to clear the warning (set **accState** to **ACC_NORMAL** or set **tempState** to **TEMP_NORMAL**) depending on the combination of joystick and rotary used as described above (also refer to subsection below for visuals). This gives the pilot advanced control to clear either both the warnings or one at time, something which SW4 **CLEAR_WARNING** cannot do as it clears all warnings when it is pressed. Next depending on the settings, the threshold is adjusted, if accelerometer threshold is to be adjusted , it is incremented or decremented by **0.1g** and capped at a max of **0.7g** and min of **0.3g**, and if the temperature is to be adjusted, it is incremented or decremented by 10 units (which is **1 degree celsius**) and maxed at 31 degrees celsius and min at 26 degrees celsius. Lastly, a message is sent to NUSCloud (refer to screenshot in below subsection) to indicate the new threshold value via the LPC_UART3 port.

4. 6. 2 Control movements on Baseboard and NUSCloud display



The above diagram the control combinations necessary to use our enhancement. A **downward** movement or **RED** on the **joystick** would be **decrease** the threshold value and an **upward GREEN** would be to **increase** it. Doing so in conjunction with **rotary switch** turned to the **left** or **YELLOW** would control the **accelerometer** threshold and **right** or **PINK** would control the **temperature** threshold.

So for example, if the temperature threshold is desired to be increase in the warning state, the order to do so would be to pull the joystick upwards with on hand and move the rotary switch to the right with another **simultaneously**. The motion is detected by the board, and the warning will be cleared with an increase in threshold that can be confirmed in the TeraTerm software like in the screenshot below.



```
VT COM4 - Tera Term VT
File Edit Setup Control Window Help
New TEMP_THRESHOLD = 300
Temp. too high.
New TEMP_THRESHOLD = 310
Temp. too high.
New TEMP_THRESHOLD = 320
Entering LAUNCH Mode
Ueer off course.
New ACC_THRESHOLD = 0.50
TEMP: 31.60 ACC X : 0.03 ACC Y : 0.03
```

5. Problems encountered and solutions proposed

The most significant problem faced was highlighted in 4.5.6 , where our light interrupt would clash with the **light_read()** function in **returnMode()** causing the baseboard to completely freeze and no action could be detected or output. This proved to be quite a hindrance to our assignment as we could not effectively the cause of the this at first, but after using the debugging tools and breakpoints at various points in our codes , we realized that this would happen each time the **light_read()** is interrupted by the light sensor interrupt. This leads us to believe perhaps some registers of the light sensor were being overridden or accessed too quickly causing either an undetected hard fault or it being stuck bouncing between two functions calling each other infinitely.

We resorted to using the `light_read()` in the `updateReturn()` function to be called in the **TIMER1** interrupt and we assigned this interrupt a **higher preempt priority** than **EINT3_IRQn** as this allowed the **light_read()** to be completed first from the TIMER1 ISR and then the light sensor interrupt to occur , ensuring that the light sensor interrupt does not preempt `light_read()` and this managed to not only solve the problem but also give us a very responsive `light_read()` and 16 LED array calibration.

6. Improvement suggestions

Overall, we felt the lab sessions were paced well and delivered the necessary information effectively. We appreciate the efforts of the lab GAs and their assistance in this assignment was paramount for us to get a good understanding on how several concepts work in conjunction with the detailed lectures from Dr Panicker and the tutorial lessons. One improvement would be to start assignment 2 before recess week or perhaps schedule it be as soon as assignment 1 so that the one extra week could come in useful to absorb the concepts and requirements for the lab beforehand.

7. Conclusion

In conclusion, assignment 2 is challenging but rewarding in the sense that we get to hands on experience with the concepts we learn in class (I^2C , SPI and UART) and we feel we have fulfilled the objectives that were set for assignment 2 , both by the requirements and our own objectives as well.

APPENDIX A

PROGRESS LOG

<u>TIMELINE</u>	<u>Items completed</u>	<u>Sections of code implemented</u>
WEEK 7	<ul style="list-style-type: none">• Familiarisation with the Baseboard.• Using the SW3 to play music	<ul style="list-style-type: none">• Played music through the baseboard's speaker
WEEK 8	<ul style="list-style-type: none">• Used sensors (temperature, accelerometer)• Applied previous lab's GPIO knowledge to implement RGB LED (BLUE and RED)• Designed flowchart for 7-segment display• Used 7-segment to display countdown• Added use of SW4 to clear warnings• Used SW3 for toggling between necessary modes.	<ul style="list-style-type: none">• Managed to obtain readings from the sensors individually (only the part to obtain sensor readings using systicks).• Successfully implemented RGB LED, and made it blink simultaneously.• Implemented the countdown feature.• Added a function to clear warnings that have been made• Implemented SW3 as EINT0 interrupt.
WEEK 9	<ul style="list-style-type: none">• Reading the necessary sensor values, and displaying them on the OLED.• Implemented the required thresholds.• Designed flowchart for the STATIONARY LAUNCH and RETURN.	<ul style="list-style-type: none">• Read the accelerometer (I²C) using interrupt, light sensor, temperature sensor (GPIO) in polling mode.• Display text on OLED.• Started implementing the necessary thresholds and warnings to be displayed on OLED.• Created necessary functions for STATIONARY mode.• Added enums for the different sensor readings and modes (TEMP_STATE, ACC_STATE, LIGHT_STATE, MODE, STATE).

WEEK 10	<ul style="list-style-type: none"> Designed flowchart for the EINT0 (mode toggle) and EINT3 (GPIO Light, temp interrupts) Added in RGB LED to blink if sensor readings exceed threshold values. 	<ul style="list-style-type: none"> Made a skeleton of the LAUNCH and RETURN mode's functions. Added logic for the different states (accSTATE, tempSTATE, lightSTATE) for the reading values. Added debouncing for SW3. Implemented toggle function (toggling between the different operation modes using SW3). Dropped accelerometer interrupt via I²C as it proved to be cumbersome and unfruitful Used interrupts for light sensor and temperature sensors
WEEK 11	<ul style="list-style-type: none"> Designed flowcharts for UART3 interrupt Integrated UART into the program. 	<ul style="list-style-type: none"> Added in the necessary UART handlers and interrupts. Added necessary functions calls to set UART messages for the sensor readings and warnings. Displayed the messages on the teraTerm console using wired UART. Added TIMER0 and TIMER1 interrupt
WEEK 12	<ul style="list-style-type: none"> Designed the special feature for our baseboard. Started working on Lab Report. 	
WEEK 13	<ul style="list-style-type: none"> Added final refactoring and tidy up the code. Prepare for assessment and submission 	

APPENDIX B

COMPLETE CODE

```
/*
 * A demo example using several of the peripherals on the base board
 *
 * Copyright(C) 2011, EE2024
 * All rights reserved.
 */

#include "lpc17xx_pinsel.h"
#include "lpc17xx_gpio.h"
#include "lpc17xx_i2c.h"
#include "lpc17xx_ssp.h"
#include "lpc17xx_timer.h"
#include "lpc17xx_uart.h"

#include "joystick.h"
#include "rotary.h"
#include "pca9532.h"
#include "acc.h"
#include "oled.h"
#include "rgb.h"
#include "temp.h"
#include "led7seg.h"
#include "light.h"

#include <string.h>
#include <stdio.h>
#include <math.h>

#define TEMP_SCALAR_DIV10 1
#define TEMP_NUM_HALF_PERIODS 340
#define MAX_LIGHT 16000
#define MIN_LIGHT 0
#define ACC_DIV_MEASURE 64.0

static int TEMP_THRESHOLD = 290;
static int OBSTACLE_THRESHOLD = 3000;
static float ACC_THRESHOLD = 0.4;

static const uint8_t SEGMENT_DISPLAY[16] =
{
    //digits 0-9
    0x24, 0xAF, 0xE0, 0xA2, 0x2B, 0x32, 0x30, 0xA7, 0x20, 0x22,
    // A-F

```

```

        0x21, 0x38, 0x74, 0xA8, 0x70, 0x71,
};

typedef enum {
    ACC_OFF,
    ACC_NORMAL,
    ACC_HIGH,
} ACC_STATE;

typedef enum {
    TEMP_OFF,
    TEMP_NORMAL,
    TEMP_HIGH,
} TEMP_STATE;

typedef enum {
    LIGHT_OFF,
    LIGHT_NORMAL,
    LIGHT_HIGH,
} LIGHT_STATE;

typedef enum {
    STATIONARY,
    LAUNCH,
    RETURN,
} MODE;

typedef struct {
    MODE modeState;
    ACC_STATE accState;
    TEMP_STATE tempState;
    LIGHT_STATE lightState;
} STATE;

typedef struct {
    int32_t temperature;
    int halfPeriods;
    uint32_t temperatureT1;
    uint32_t temperatureT2;
} TEMP;

typedef struct {
    float acc_x;
    float acc_y;
    int32_t temp;
    uint32_t light;
    uint16_t brightness;
} DATA;

typedef struct {
    int8_t acc_x[6];
    int8_t acc_y[6];
    uint8_t temp[6];
    uint8_t light[6];

```



```

        uint8_t uart[100];
    } DISPLAY;

    int8_t xoff, yoff;
    int8_t x, y, z;
    uint8_t msg_buf[64];
    uint32_t buf_count = 0;

    static uint8_t clear_warning = 0;

    // Optimization flags
    static int temp_change = 1;
    static int acc_change = 0;
    static int light_change = 0;
    static int telemetry_change = 0;
    static int obstacle_avoid = 0;
    static int message_received = 0;
    static int launch_toggle = 0;
    static int mode_toggle = 0;

    // De-bouncing flags
    static int firstPress = 1;
    static int secondPress = 0;
    static int launchFirstPress = 1;
    static int launchSecondPress = 0;

    uint32_t launchFirstPressTime;
    uint32_t launchSecondPressTime = 0;
    uint32_t firstPressTime;
    uint32_t secondPressTime = 0;
    volatile uint32_t msTicks;

    static DATA data = {0, 0, 0, 0};
    static DISPLAY display = {"", "", "", "", ""};
    static TEMP temp = {0, 0, 0, 0};

    static STATE state = {STATIONARY, ACC_OFF, TEMP_NORMAL, LIGHT_OFF};

    // ***** FUNCTION PROTOTYPES *****
    ////////// Init functions //////////
    void init_all(void);
    static void init_ssp(void);
    static void init_i2c(void);
    static void init_uart(void);
    static void init_sw3(void);
    static void init_sw4(void);
    void init_light_sensor(void);
    void init_temp_sensor(void);
    void init_timer0(void);

    ////////// Main Functions //////////
    void warn_blink(void);
    void countDown(void);
    void configStationary(void);

```

```

void updateStationary(void);
void configLaunch(void);
void updateLaunch(void);
void configReturn(void);
void updateReturn(void);
void modeToggle(void);
void telemetryMsg(void);
void checkLaunchMsg(void);
void checkReturnMsg(void);

////////// Temp Functions //////////
void temp_enable(void);
void temp_disable(void);
void adjusted_temp_read(TEMP*);
void checkTemp(void);

////////// Acc Functions //////////
void checkAcc(void);

////////// Light Functions //////////
void light_sensor_enable(void);
void light_sensor_disable(void);
static uint16_t getBrightness(uint32_t);

////////// UART Functions //////////
void UART3_Rx_Interrupt(void);
void set_uart_message(uint8_t* , char* );
void set_uart_string(void);

////////// Peripheral Functions //////////
void checkManualSettings(void);

void SysTick_Handler(void) {
    msTicks++;
    warn_blink();
}
uint32_t getTicks(void) {
    return msTicks;
}

void warn_blink(void) {
    if (state.accState == ACC_HIGH && state.tempState == TEMP_HIGH) {
        if (msTicks % 333 == 0) {
            rgb_setLeds(RGB_BLUE);
        }
        if ((msTicks + 167) % 333 == 0) {
            rgb_setLeds(RGB_RED);
        }
    } else if (state.accState == ACC_HIGH) {
        if (msTicks % 333 == 0) {
            rgb_setLeds(RGB_BLUE);
        }
        if ((msTicks + 167) % 333 == 0) {
            rgb_setLeds(0);
        }
    }
}

```

```

    }
} else if (state.tempState == TEMP_HIGH) {
    if (msTicks % 333 == 0) {
        rgb_setLeds(RED);
    }
    if ((msTicks + 167) % 333 == 0) {
        rgb_setLeds(0);
    }
} else {
    rgb_setLeds(0);
}
}

// ***** EINT0 MODE TOGGLE INTERRUPT AND FUNCTIONS *****
void EINT0_IRQHandler(void) {

    // SW3 External Interrupt
    if ((LPC_SC -> EXTINT) & 0x01) {

        // Every other mode except in launch (1 press is mode toggle)
        if (state.modeState != LAUNCH) {

            if (firstPress) {
                firstPressTime = getTicks(); // Record time when first press
                firstPress = 0;
                secondPress = 1;
            } else if (secondPress) {
                secondPressTime = getTicks();
                secondPress = 0;
                firstPress = 1;
            }

            // If the delay between first and second press is larger than 1 second
            if (secondPressTime == 0) {
                mode_toggle = 1;
            } else if ((int)(secondPressTime - firstPressTime) > 1000) {
                mode_toggle = 1;
            } else { // Reset sw3 to prepare for next presses
                firstPress = 1;
                firstPressTime = 0;
                secondPress = 0;
                secondPressTime = 0;
            }

        } else {

            if (launchFirstPress) {
                launchFirstPressTime = getTicks();
                launchFirstPress = 0;
                launchSecondPress = 1;
            } else if (launchSecondPress) {
                launchSecondPressTime = getTicks();
                launchSecondPress = 0;
            }
        }
    }
}

```

```

        launchFirstPress = 1;
    }

    if (launchSecondPressTime == 0) {
        launch_toggle = 0;

    } else if (launchSecondPressTime - launchFirstPressTime <= 1000) {
        mode_toggle = 1;
        launch_toggle = 1;
    } else {
        launchFirstPressTime = launchSecondPressTime;
        launchSecondPress = 1;
        launchFirstPress = 0;
    }
}

// Reset sw3 to prepare for next presses
if (launch_toggle) {
    launchFirstPress = 1;
    launchFirstPressTime = 0;
    launchSecondPress = 0;
    launchSecondPressTime = 0;
    firstPress = 1;
    firstPressTime = 0;
    secondPress = 0;
    secondPressTime = 0;
}

}

// clear interrupt
LPC_SC->EXTINT |= (1<<0);
}

void modeToggle(void) {

    if ((state.modeState == STATIONARY) && (state.tempState != TEMP_HIGH)) {
        countDown();
        // if second mode toggle is recorded and within 1 second of the first mode toggle
    } else if ((state.modeState == LAUNCH) && (launch_toggle)) {
        configReturn();
        state.modeState = RETURN;
        launch_toggle = 0;

    } else if (state.modeState == RETURN) {
        checkLightWarning();
        configStationary();
        state.modeState = STATIONARY;
    }
}

void countDown(void) {
    int i=14;
    uint32_t currentTime, initialTime;

    initialTime = getTicks();

```

```

while (state.tempState != TEMP_HIGH) {
    currentTime = getTicks();
    if (currentTime - initialTime < 1000) {
        led7seg_setChar(SEGMENT_DISPLAY[i], TRUE);
    }
    else {
        initialTime = getTicks();
        if (i==0) {
            configLaunch();
            state.modeState = LAUNCH;
            break;
        }
        else {
            i--;
        }
    }
}

// Abort count down
if (state.tempState == TEMP_HIGH) {
    led7seg_setChar(SEGMENT_DISPLAY[15], TRUE);
}
}

// ***** EINT3 GPIO LIGHT, TEMP INTERRUPTS *****
void EINT3_IRQHandler(void)
{
    // Light Sensor Interrupt (Obstacle detection)
    if ((LPC_GPIOINT->IO2IntStatF >> 5) & 0x1) {
        if (state.lightState == LIGHT_NORMAL) {
            state.lightState = LIGHT_HIGH;
            light_change = 1;
            light_setHiThreshold(MAX_LIGHT);
            light_setLoThreshold(OBSTACLE_THRESHOLD);
        }
        else if (state.lightState == LIGHT_HIGH){
            state.lightState = LIGHT_NORMAL;
            obstacle_avoid = 1;
            light_setHiThreshold(OBSTACLE_THRESHOLD);
            light_setLoThreshold(0);
        }

        light_clearIrqStatus();
        LPC_GPIOINT->IO2IntClr = (1 << 5);
    }

    // Temp sensor Interrupt (Monitor Fuel Tank)
    if ((LPC_GPIOINT->IO0IntStatF >> 2) & 0x1) {

        adjusted_temp_read(&temp);

        if ((state.tempState == TEMP_NORMAL) && (temp.temperature > TEMP_THRESHOLD)) {
            state.tempState = TEMP_HIGH;

```

```

        temp_change = 1;
    }
    LPC_GPIOINT->IO0IntClr |= (1 << 2);
}

// ***** LIGHT SENSOR FUNCTIONS *****

void light_sensor_enable(void) {
    state.lightState = LIGHT_NORMAL;
    LPC_GPIOINT->IO2IntEnF |= 1<<5;
}

void light_sensor_disable(void) {
    light_shutdown();
    LPC_GPIOINT->IO2IntEnF &= ~(1<<5);
    state.lightState = LIGHT_OFF;
}

// ***** TEMPERATURE SENSOR FUNCTIONS *****

void temp_disable(void) {
    LPC_GPIOINT->IO0IntEnF &= ~(1<<2);
    LPC_GPIOINT->IO0IntClr |= (1 << 2);
    state.tempState = TEMP_OFF;
}

void temp_enable(void) {
    LPC_GPIOINT->IO0IntEnF |= (1<<2);
    temp.halfPeriods = 0;
    temp.temperature = 0;
    temp.temperatureT1 = 0;
    temp.temperatureT2 = 0;
    state.tempState = TEMP_NORMAL;
}

void adjusted_temp_read(TEMP* temp) {
    if (!temp->temperatureT1 && !temp->temperatureT2) {
        temp->temperatureT1 = getTicks();
    } else if (temp->temperatureT1 && !temp->temperatureT2) {
        temp->halfPeriods++;
        if (temp->halfPeriods == TEMP_NUM_HALF_PERIODS/2) {
            temp->temperatureT2 = getTicks();
            if (temp->temperatureT2 > temp->temperatureT1) {
                temp->temperatureT2 = temp->temperatureT2 - temp->temperatureT1;
            }
            else {
                temp->temperatureT2 = (0xFFFFFFFF - temp->temperatureT1 + 1) +
temp->temperatureT2;
            }
            temp->temperature = ((2*1000*temp->temperatureT2) /
(TEMP_NUM_HALF_PERIODS*TEMP_SCALAR_DIV10) - 2731);
            temp->temperatureT1 = 0;
            temp->temperatureT2 = 0;
            temp->halfPeriods = 0;

```

```

    }
}

// ***** UART 3 INTERRUPT *****
void UART3_IRQHandler(void) {
    UART3_StdIntHandler();
}

void UART3_Rx_Interrupt(void) {

    if(UART_Receive(LPC_UART3, &msg_buf[buf_count], 1, NONE_BLOCKING) == 1) {
        if(msg_buf[buf_count] == '\n'){
            message_received = 1;
        }
        buf_count++;

        if (buf_count == 64) {
            buf_count = 0;
        }
    }
}

void set_uart_message(uint8_t* UART_DISPLAY, char* string) {
    strcpy((char*)UART_DISPLAY, "");
    strcat((char*)UART_DISPLAY, string);
}

void set_uart_string(){
    msg_buf[buf_count-1] = '\0';
    buf_count = 0;
}

// ***** TIMER 0 INTERRUPT *****
void TIMER0_IRQHandler(void) {
    // check if TIMER0 M0 interrupt
    if ((LPC_TIM0->IR >> 0) & 0x1) {
        if (state.modeState == LAUNCH || state.modeState == RETURN) {
            telemetry_change = 1; // every 10seconds flag for telemetry data to be sent
        }
        LPC_TIM0->IR |= (1<<0); // clear TIMER0 interrupt
    }
}

void telemetryMsg(void) {
    char s[40] = "";
    if (telemetry_change) {
        if (state.modeState == LAUNCH) {
            sprintf(s, "TEMP: %.2f ACC X : %.2f ACC Y : %.2f\r\n", data.temp/10.0,
data.acc_x, data.acc_y);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            telemetry_change = 0;
        }
    }
}

```

```

        if (state.modeState == RETURN) {
            sprintf(s, "Obstacle distance: %d\r\n", (int)data.light);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            telemetry_change = 0;
        }
    }
}

// ***** TIMER 1 INTERRUPT *****
void TIMER1_IRQHandler(void) {
    // check if TIMER0 M0 interrupt
    if ((LPC_TIM1->IR >> 0) & 0x1) {
        if (state.modeState == RETURN) {
            updateReturn(); // every 10seconds flag for telemetry data to be sent
        }
        LPC_TIM1->IR |= (1<<0); // clear TIMER0 interrupt
    }
}

// ***** SW4 CLEAR_WARNING FUNCTION *****
void checkClearWarning(void) {
    clear_warning = (GPIO_ReadValue(1) >> 31) & 0x01;

    if (clear_warning == 0) {
        if (state.tempState == TEMP_HIGH) {
            state.tempState = TEMP_NORMAL;
            temp_change = 1;
        }
        if (state.accState == ACC_HIGH) {
            state.accState = ACC_NORMAL;
            acc_change = 1;
        }
    }
}

void checkLightWarning(void) {
    if (state.lightState == LIGHT_HIGH) {
        state.lightState = LIGHT_NORMAL;
        light_setHiThreshold(OBSTACLE_THRESHOLD);
        light_setLoThreshold(0);
        set_uart_message(display.uart, "Obstacle Avoided.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
    }
}

// ***** OTHER PERIPHERAL FUNCTIONS *****
void checkManualSettings(void) {

    uint8_t rState = 0; //rotary state

```



```

uint8_t jState = 0; //joystick state

rState = rotary_read();
jState = joystick_read();

char s[40] = "";

if (state.tempState == TEMP_HIGH || state.accState == ACC_HIGH) {

    if (jState) {
        if(rState == ROTARY_LEFT) {
            if (state.accState == ACC_HIGH) {
                state.accState = ACC_NORMAL;
                acc_change = 1;
            }
            if (jState == JOYSTICK_UP) {
                if (ACC_THRESHOLD <= 0.7) {
                    ACC_THRESHOLD += 0.1;
                }
            } else if (jState == JOYSTICK_DOWN) {
                if (ACC_THRESHOLD >= 0.3) {
                    ACC_THRESHOLD -= 0.1;
                }
            }
            sprintf(s, "New ACC_THRESHOLD = %.2f\r\n", ACC_THRESHOLD);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
        } else if (rState == ROTARY_RIGHT) {

            if (state.tempState == TEMP_HIGH) {
                state.tempState = TEMP_NORMAL;
                temp_change = 1;
            }

            if (jState == JOYSTICK_UP) {
                if (TEMP_THRESHOLD <= 310) {
                    TEMP_THRESHOLD += 10;
                }
            } else if (jState == JOYSTICK_DOWN) {
                if (TEMP_THRESHOLD >= 260) {
                    TEMP_THRESHOLD -= 10;
                }
            }
            sprintf(s, "New TEMP_THRESHOLD = %d\r\n", TEMP_THRESHOLD);
            set_uart_message(display.uart, s);
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
        }
    }
}
}

```

```
// ***** WITHIN MODE FUNCTIONS *****
*****

void configStationary(void) {

    // Entering for first time into stationary from return mode
    state.lightState = LIGHT_OFF;
    state.accState = ACC_OFF;
    oled_clearScreen(OLED_COLOR_BLACK);
    set_uart_message(display.uart, "Entering STATIONARY Mode\r\n");
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);

    pca9532_setLeds(0x0000, 0xFFFF);

    // temp sensor enabled
    temp_enable();
    oled_putString(0, 0, (uint8_t *) "STATIONARY", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    oled_putString(0, 10, (uint8_t *) "TEMP" , OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    led7seg_setChar(SEGMENT_DISPLAY[15], TRUE);
}

void stationaryMode() {

    checkClearWarning();

    updateStationary();

    if (message_received) { // Message sent in wrong mode
        set_uart_message(display.uart, "Report not enabled in this mode.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        message_received = 0;
        buf_count = 0;
    }

    if (state.tempState == TEMP_HIGH) {
        if (temp_change) {
            oled_putString(0, 40, (uint8_t *) "Temp. too high", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

            set_uart_message(display.uart, "Temp. too high.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
            temp_change = 0;
        }

        } else if (temp_change) {
            oled_putString(0, 40, (uint8_t *) "          ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
            temp_change = 0;
        }
    }

    void updateStationary(void) {
        data.temp = temp.temperature;
        char s[16] = "";
        sprintf(s, "%2.2f", data.temp/10.0);
    }
}

```

```

        oled_putString(60, 10, (uint8_t *) s, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    }

    void configLaunch(void) {
        // Entering launch for first time from Stationary
        set_uart_message(display.uart, "Entering LAUNCH Mode\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        state.accState = ACC_NORMAL;

        oled_clearScreen(OLED_COLOR_BLACK);
        oled_putString(0, 0, (uint8_t *) "LAUNCH", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
        oled_putString(0, 10, (uint8_t *) "TEMP", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
        oled_putString(0, 20, (uint8_t *) "ACC", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
        led7seg_setChar(SEGMENT_DISPLAY[0], TRUE);
    }

    void launchMode() {

        checkAcc();
        updateLaunch();
        checkClearWarning();
        checkLaunchMsg();

        if (state.tempState == TEMP_HIGH) {
            if (temp_change) {
                oled_putString(0, 40, (uint8_t *) "Temp. too high", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

                set_uart_message(display.uart, "Temp. too high.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
                temp_change = 0;
            }
        } else if (temp_change) {
            oled_putString(0, 40, (uint8_t *) "                ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
            temp_change = 0;
        }

        if (state.accState == ACC_HIGH) {
            if (acc_change) {
                oled_putString(0, 50, (uint8_t *) "Veer off course", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

                set_uart_message(display.uart, "Veer off course.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
                acc_change = 0;
            }
        } else if (acc_change) {
            oled_putString(0, 50, (uint8_t *) "                ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);
            acc_change = 0;
        }
    }

    void checkAcc(void) {
        acc_read(&x, &y, &z);
    }

```

```

x = x + xoff;
y = y + yoff;

data.acc_x = fabs (x / ACC_DIV_MEASURE) ;
data.acc_y = fabs (y / ACC_DIV_MEASURE) ;

if (state.accState == ACC_NORMAL) {
    if ((data.acc_x > ACC_THRESHOLD) || (data.acc_y > ACC_THRESHOLD)) {
        state.accState = ACC_HIGH;
        acc_change = 1;
    }
}

}

void updateLaunch(void) {

    data.temp = temp.temperature;
    char s[16] = "";
    sprintf(s, "%.2f", data.temp/10.0);
    oled_putString(60, 10, (uint8_t *) s, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    // after exiting measure, interrupt re-starts again

    char xy[16] = "";
    sprintf(xy, "%.2f, %.2f", data.acc_x, data.acc_y);
    oled_putString(30, 20, (uint8_t *) xy, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
}

void checkLaunchMsg(void) {

    char s[40] = "";
    if (state.tempState == TEMP_NORMAL && state.accState == ACC_NORMAL) {
        if (message_received) {
            message_received = 0;
            set_uart_string();
            if (strcmp(msg_buf, "RPT\r")==0) {
                sprintf(s, "TEMP: %.2f ACC X : %.2f ACC Y : %.2f\r\n", data.temp/10.0,
data.acc_x, data.acc_y );
                set_uart_message(display.uart, s);
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            } else {
                set_uart_message(display.uart, "Unknown message received.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            }
        }
    }
    }else if (message_received) {
        set_uart_message(display.uart, "Cannot report in warning state.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        message_received = 0;
        buf_count = 0;
    }
}

```

```

void configReturn(void) {
    // Entering Return for first time from launch
    set_uart_message(display.uart, "Entering RETURN Mode\r\n");
    UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);

    state.lightState = LIGHT_NORMAL;
    state.accState = ACC_OFF;
    temp_disable();

    oled_clearScreen(OLED_COLOR_BLACK);
    oled_putString(0, 0, (uint8_t *) "RETURN", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
    // led7seg_setChar(SEGMENT_DISPLAY[0], TRUE); 7seg is the same from launch mode
}

void updateReturn(void) {
    data.light = light_read();
    data.brightness = getBrightness(data.light);
    pca9532_setLeds(data.brightness, 0xFFFF);
}

static uint16_t getBrightness(uint32_t light) {
    if (light < 242) {
        return 0x0000;
    } else if (light < 484) { // 1
        return 0x0003;
    } else if (light < 726) { // 2
        return 0x0007;
    } else if (light < 968) { // 3
        return 0x000F;
    } else if (light < 1210) { // 4
        return 0x001F;
    } else if (light < 1452) { // 5
        return 0x003F;
    } else if (light < 1694) { // 6
        return 0x007F;
    } else if (light < 1936) { // 7
        return 0x00FF;
    } else if (light < 2178) { // 8
        return 0x01FF;
    } else if (light < 2420) { // 9
        return 0x03FF;
    } else if (light < 2662) { // 10
        return 0x07FF;
    } else if (light < 2904) { // 11
        return 0x0FFF;
    } else if (light < 3146) { // 12
        return 0x1FFF;
    } else if (light < 3388) { // 13
        return 0x3FFF;
    } else if (light < 3630) { // 14
        return 0x7FFF;
    } else {
        return 0xFFFF;
    }
}

```

```

}

void returnMode() {

    if (state.lightState == LIGHT_NORMAL) {
        if (obstacle_avoid) {
            obstacle_avoid = 0;
            oled_putString(0, 10, (uint8_t *) "          ", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

            set_uart_message(display.uart, "Obstacle Avoided.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        }
    } else if (state.lightState == LIGHT_HIGH) {
        if (light_change) {
            light_change = 0;
            oled_putString(0, 10, (uint8_t *) "Obstacle near", OLED_COLOR_WHITE,
OLED_COLOR_BLACK);

            set_uart_message(display.uart, "Obstacle near.\r\n");
            UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        }
    }

    checkReturnMsg();
}

void checkReturnMsg(void) {

    char x[40] = "";

    if (state.lightState == LIGHT_NORMAL) {
        if (message_received) {
            message_received = 0;
            set_uart_string();
            if (strcmp(msg_buf, "RPT\r") == 0 ) {
                sprintf(x, "Obstacle distance: %d\r\n", (int)data.light);
                set_uart_message(display.uart, x);
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            } else {
                set_uart_message(display.uart, "Unknown message received.\r\n");
                UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart),
BLOCKING);
            }
        }
    } else if (message_received) {
        set_uart_message(display.uart, "Cannot report in warning state.\r\n");
        UART_Send(LPC_UART3, display.uart, strlen((char*)display.uart), BLOCKING);
        message_received = 0;
        buf_count = 0;
    }
}

// ***** ALL INITIALIZATIONS *****

```

```

static void init_i2c(void)
{
    PINSEL_CFG_Type PinCfg;

    /* Initialize I2C2 pin connect */
    PinCfg.Funcnum = 2;
    PinCfg.Pinnum = 10;
    PinCfg.Portnum = 0;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 11;
    PINSEL_ConfigPin(&PinCfg);

    // Initialize I2C peripheral
    I2C_Init(LPC_I2C2, 100000);

    /* Enable I2C1 operation */
    I2C_Cmd(LPC_I2C2, ENABLE);
}

static void init_ssp(void)
{
    SSP_CFG_Type SSP_ConfigStruct;
    PINSEL_CFG_Type PinCfg;

    /*
     * Initialize SPI pin connect
     * P0.7 - SCK;
     * P0.8 - MISO
     * P0.9 - MOSI
     * P2.2 - SSEL - used as GPIO
     */
    PinCfg.Funcnum = 2;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;
    PinCfg.Portnum = 0;
    PinCfg.Pinnum = 7;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 8;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 9;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Funcnum = 0;
    PinCfg.Portnum = 2;
    PinCfg.Pinnum = 2;
    PINSEL_ConfigPin(&PinCfg);

    SSP_ConfigStructInit(&SSP_ConfigStruct);

    // Initialize SSP peripheral with parameter given in structure above
    SSP_Init(LPC_SSP1, &SSP_ConfigStruct);

    // Enable SSP peripheral
    SSP_Cmd(LPC_SSP1, ENABLE);
}

```

```

}

void init_uart(void) {
    UART_CFG_Type uartCfg;

    UART_FIFO_CFG_Type fifoCfg;

    //uart pinssel
    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 2;
    PinCfg.Pinnum = 0;
    PinCfg.Portnum = 0;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 1;
    PINSEL_ConfigPin(&PinCfg);

    // init default UART config , BR 9600
    UART_ConfigStructInit(&uartCfg);
    // change baudrate to 115200
    uartCfg.Baud_rate = 115200;
    UART_Init(LPC_UART3, &uartCfg);

    UART_FIFOConfigStructInit(&fifoCfg);
    UART_FIFOConfig(LPC_UART3, &fifoCfg);

    UART_SetupCbs(LPC_UART3, 0, UART3_Rx_Interrupt); //Call back function
    UART_TxCmd(LPC_UART3, ENABLE);
    UART_IntConfig(LPC_UART3, UART_INTCFG_RBR, ENABLE);
}

void init_light_sensor(void) {
    light_enable();
    light_setRange(LIGHT_RANGE_16000);
    LPC_GPIOINT->IO2IntEnF |= (1<<5);
    light_setHiThreshold(OBSTACLE_THRESHOLD);
    light_clearIrqStatus();
}

void init_temp_sensor(void) {
    PINSEL_CFG_Type PinCfg;

    PinCfg.Funcnum = 0;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;
    PinCfg.Portnum = 0;
    PinCfg.Pinnum = 2;
    PINSEL_ConfigPin(&PinCfg);

    GPIO_SetDir(0, (1<<2), 0);
}

static void init_sw3(void) {
    //SW3 MODE_TOGGLE

```



```

    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 1;
    PinCfg.Portnum = 2;
    PinCfg.Pinnum = 10;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;
    PINSEL_ConfigPin(&PinCfg);

    // Enable EINT0 Interrupt
    LPC_SC->EXTMODE |= 0x01;
    LPC_SC->EXTPOLAR &= 0xFFFFE; // set EINT0 to trigger on falling edge
}

static void init_sw4(void) {
    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 0;
    PinCfg.OpenDrain = 0;
    PinCfg.Pinmode = 0;

    //SW4 CLEAR_WARNING
    PinCfg.Portnum = 1;
    PinCfg.Pinnum = 31;
    PINSEL_ConfigPin(&PinCfg);
    // (portnum, pinnum, in/out (0/1) )
    GPIO_SetDir(0, 1<<31, 0); // SW4
}

void init_timer0(void) {
    LPC_SC->PCONP |= (1<<1); //Power up TIMER0
    LPC_SC->PCLKSEL0 |= (0b01 << 2);
    LPC_TIM0->TCR = 2; // reset & hold TIMER0
    LPC_TIM0->MR0 = SystemCoreClock*10; // Interrupt every 10seconds
    LPC_TIM0->MCR |= 1<<0; // Interrupt on Match0 compare
    LPC_TIM0->MCR |= 1<<1; // reset TIMER0 on match 0
}

void init_timer1(void) {
    LPC_SC->PCONP |= (1 << 2); // power up TIMER1
    // PCLK_peripheral = SystemCoreClock
    LPC_SC->PCLKSEL0 |= (0b01 << 4);
    LPC_TIM1->TCR = 2; // reset & hold TIMER1
    LPC_TIM1->MR0 = (SystemCoreClock/4); // interrupt every 250 millisecond
    LPC_TIM1->MCR |= 1 << 0; // interrupt on Match1 compare
    LPC_TIM1->MCR |= 1 << 1; // reset TIMER1 on Match 1
}

void init_all() {
    init_i2c();
    init_ssp();
    init_uart();

    led7seg_init();
    rotary_init();
}

```

```

        joystick_init();
pca9532_init();
pca9532_setLeds(0x0000, 0xffff);
    rgb_init();
    oled_init();
    acc_init();
    acc_read(&x, &y, &z);
    xoff = 0 - x;
    yoff = 0 - y;
    oled_clearScreen(OLED_COLOR_BLACK);

    //EINT3_GPIO
    init_light_sensor();

//EINT3_GPIO
    init_temp_sensor();

    //EINT0 MODE_TOGGLE
init_sw3();

    //TIMER0
    init_timer0();

    //TIMER1
    init_timer1();

// CLEAR_WARNING
    init_sw4();

    NVIC_SetPriorityGrouping(5);

    NVIC_SetPriority(SysTick_IRQn, 0x00);
    NVIC_SetPriority(EINT0_IRQn, 0x40);
    NVIC_SetPriority(TIMER1_IRQn, 0x48);
    NVIC_SetPriority(TIMER0_IRQn, 0x50);
    NVIC_SetPriority(EINT3_IRQn, 0x80);
    NVIC_SetPriority(UART3_IRQn, 0x88);

    NVIC_ClearPendingIRQ(EINT0_IRQn);
    NVIC_ClearPendingIRQ(TIMER0_IRQn);
    NVIC_ClearPendingIRQ(TIMER1_IRQn);
    NVIC_ClearPendingIRQ(EINT3_IRQn);
    NVIC_ClearPendingIRQ(UART3_IRQn);

    NVIC_EnableIRQ(EINT0_IRQn);
    NVIC_EnableIRQ(TIMER1_IRQn);
    NVIC_EnableIRQ(TIMER0_IRQn);
    NVIC_EnableIRQ(EINT3_IRQn);
    NVIC_EnableIRQ(UART3_IRQn);
}

int main (void) {
    SysTick_Config(SystemCoreClock/1000);

```

```
init_all();
LPC_TIM0->TCR = 1; //start timer0
LPC_TIM1->TCR = 1; //start timer1
configStationary();
telemetry_change = 0;

while (1)
{
    telemetryMsg();
    checkManualSettings();
    if (mode_toggle) {
        modeToggle();
        mode_toggle = 0;
    }
    switch (state.modeState) {
    case STATIONARY :
        stationaryMode();
        break;

    case LAUNCH :
        launchMode();
        break;

    case RETURN :
        returnMode();
        break;
    }
}
```