EE2024 Project

Assignment 2:

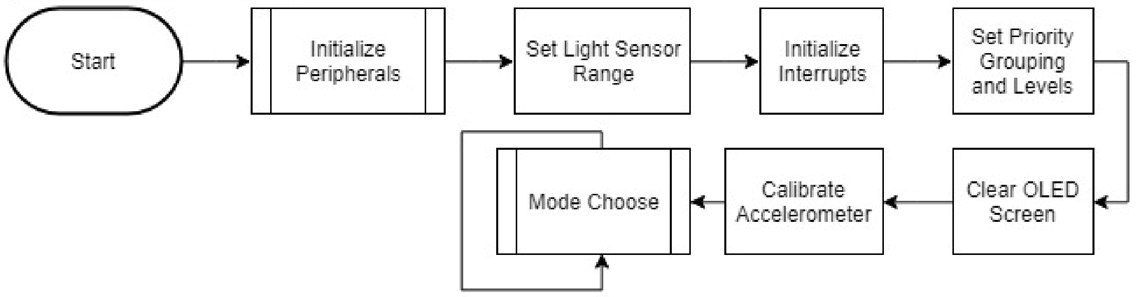
NUSpace

Xie Yuheng (A0149868E) & Zechariah Gerard Tan Jia Le (A0155361J)

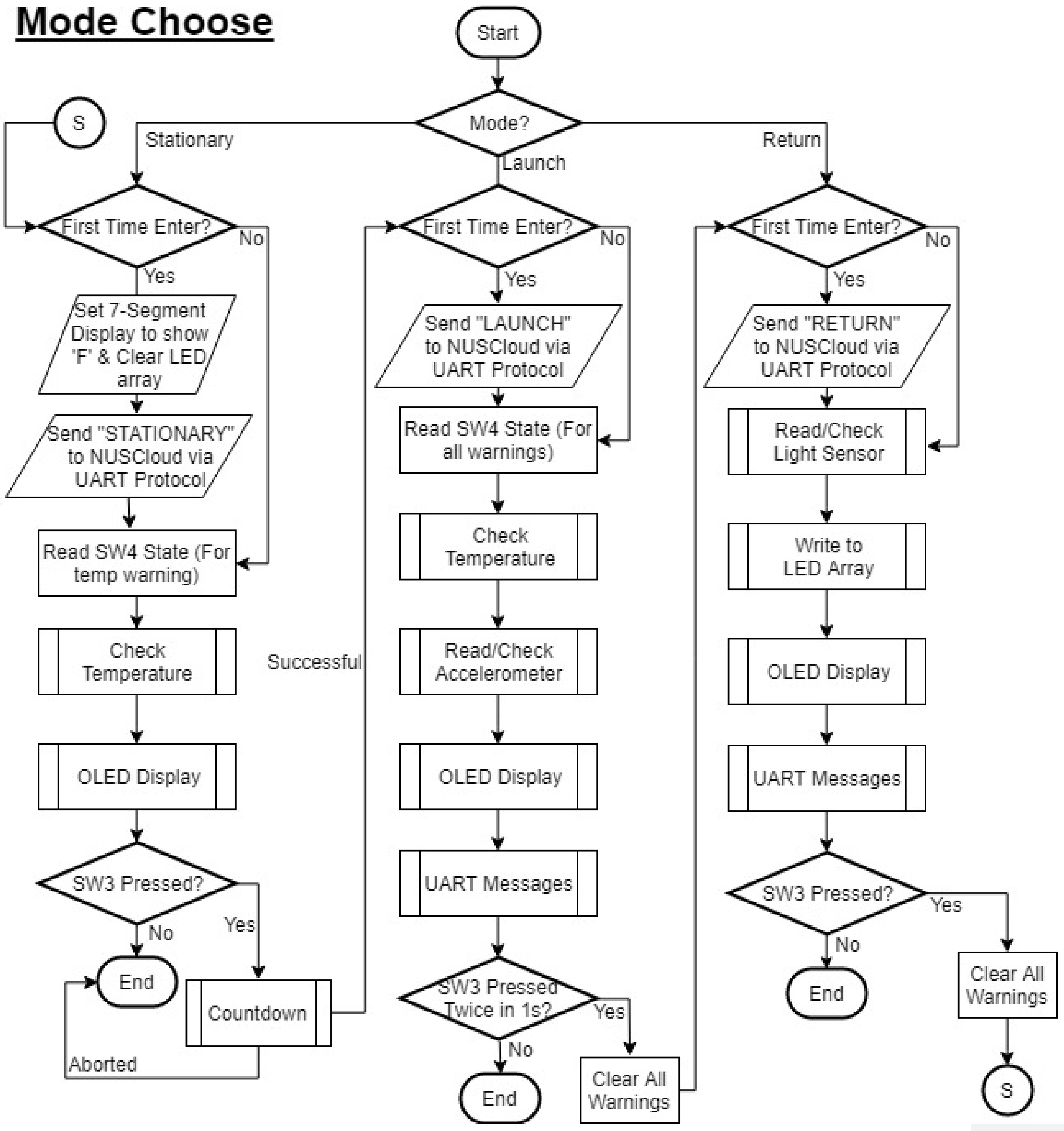
Wednesday [W1]

Table of Contents

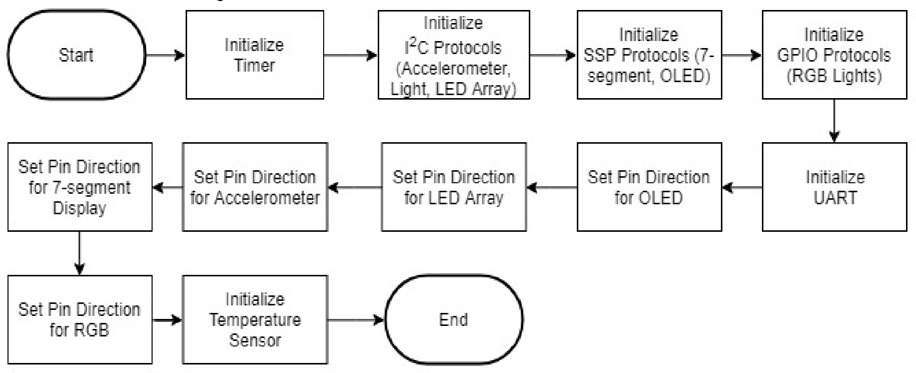
Main Function



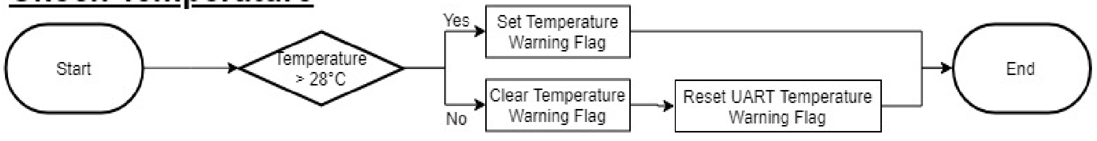
Mode Choose



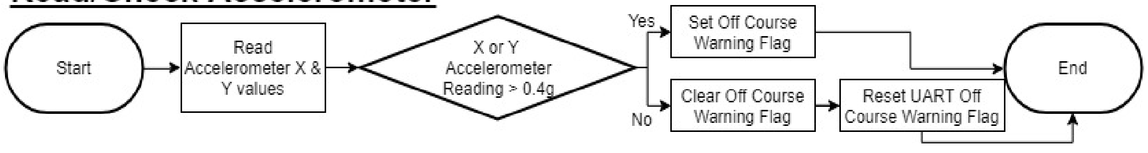
Initialize Peripherals


Start
Initialize Timer
Initialize I²C Protocols (Accelerometer, Light, LED Array)
Initialize SSP Protocols (7-segment, OLED)
Initialize GPIO Protocols (RGB Lights)
Initialize UART
Set Pin Direction for OLED
Set Pin Direction for LED Array
Set Pin Direction for Accelerometer
Set Pin Direction for 7-segment Display
Set Pin Direction for RGB
Initialize Temperature Sensor
End

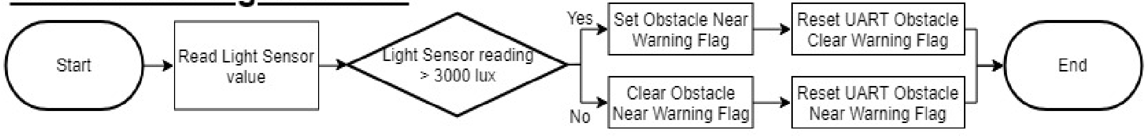
Check Temperature


Start
Temperature > 28°C
Yes
Set Temperature Warning Flag
No
Clear Temperature Warning Flag
Reset UART Temperature Warning Flag
End

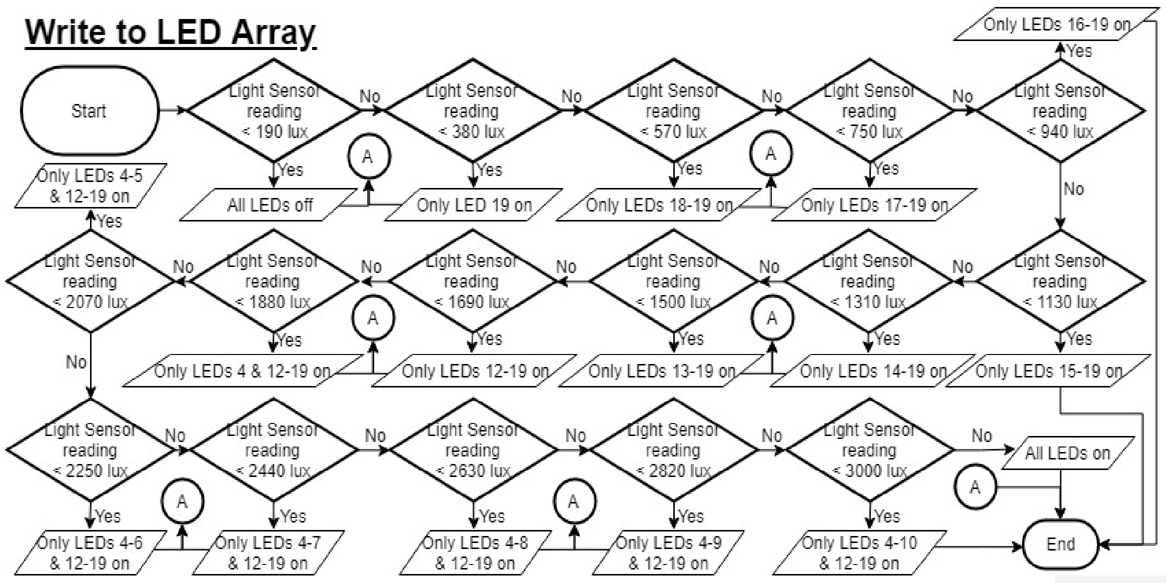
Read/Check Accelerometer


Start
Read Accelerometer X & Y values
X or Y Accelerometer Reading > 0.4g
Yes
Set Off Course Warning Flag
No
Clear Off Course Warning Flag
Reset UART Off Course Warning Flag
End

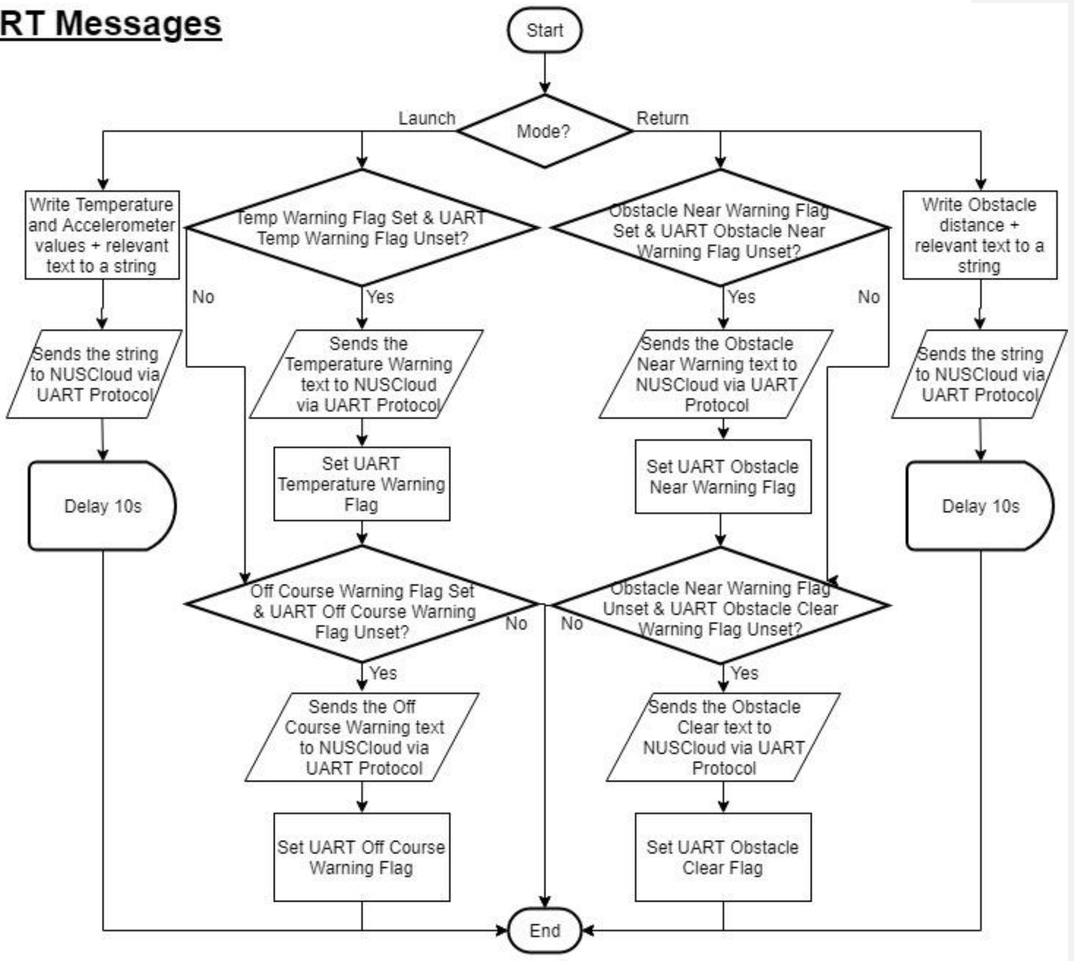
Read/Check Light Sensor


Start
Read Light Sensor value
Light Sensor reading > 3000 lux
Yes
Set Obstacle Near Warning Flag
Reset UART Obstacle Clear Warning Flag
No
Clear Obstacle Near Warning Flag
Reset UART Obstacle Near Warning Flag
End

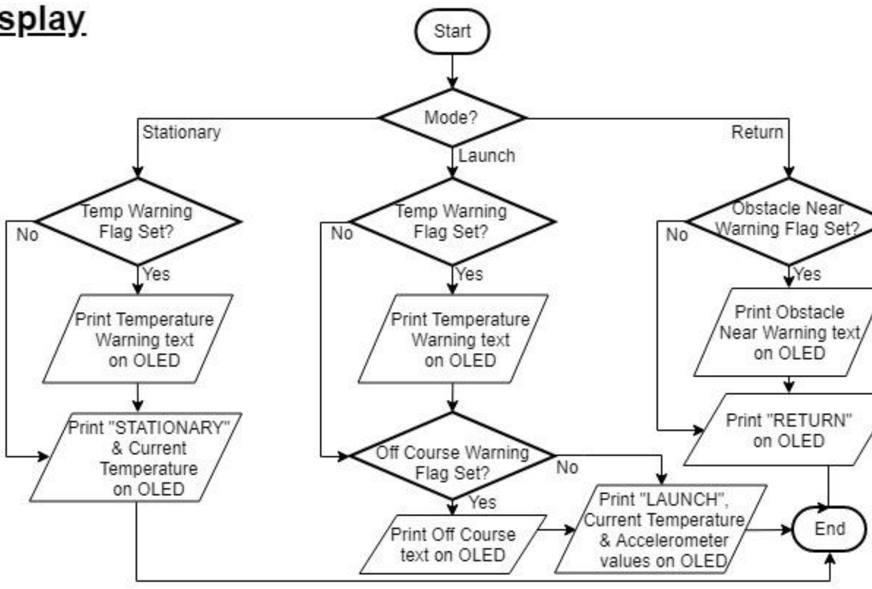
Write to LED Array


Start
Light Sensor reading < 190 lux
Yes
All LEDs off
No
Light Sensor reading < 380 lux
Yes
Only LED 19 on
No
Light Sensor reading < 570 lux
Yes
Only LEDs 18-19 on
No
Light Sensor reading < 750 lux
Yes
Only LEDs 17-19 on
No
Light Sensor reading < 940 lux
Yes
Only LEDs 16-19 on
No
Light Sensor reading < 1130 lux
Yes
Only LEDs 15-19 on
No
Light Sensor reading < 1310 lux
Yes
Only LEDs 14-19 on
No
Light Sensor reading < 1500 lux
Yes
Only LEDs 13-19 on
No
Light Sensor reading < 1690 lux
Yes
Only LEDs 12-19 on
No
Light Sensor reading < 1880 lux
Yes
Only LEDs 4 & 12-19 on
No
Light Sensor reading < 2070 lux
Yes
Only LEDs 4-5 & 12-19 on
No
Light Sensor reading < 2250 lux
Yes
Only LEDs 4-6 & 12-19 on
No
Light Sensor reading < 2440 lux
Yes
Only LEDs 4-7 & 12-19 on
No
Light Sensor reading < 2630 lux
Yes
Only LEDs 4-8 & 12-19 on
No
Light Sensor reading < 2820 lux
Yes
Only LEDs 4-9 & 12-19 on
No
Light Sensor reading < 3000 lux
Yes
Only LEDs 4-10 & 12-19 on
No
All LEDs on
A
End

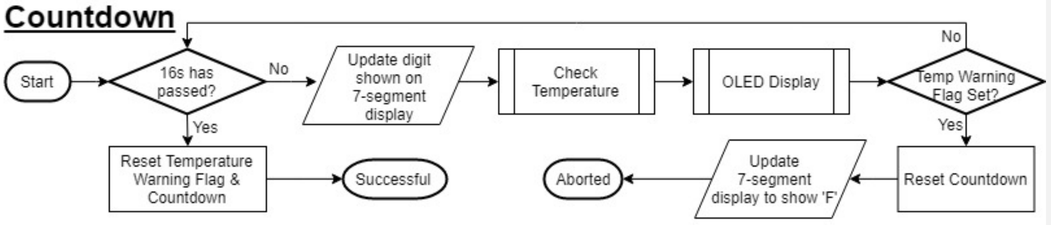
UART Messages


Start
Mode?
Launch
Return
Write Temperature and Accelerometer values + relevant text to a string
Temp Warning Flag Set & UART Temp Warning Flag Unset?
Obstacle Near Warning Flag Set & UART Obstacle Near Warning Flag Unset?
Write Obstacle distance + relevant text to a string
No
Yes
Yes
No
Sends the string to NUSCloud via UART Protocol
Sends the Temperature Warning text to NUSCloud via UART Protocol
Sends the Obstacle Near Warning text to NUSCloud via UART Protocol
Sends the string to NUSCloud via UART Protocol
Set UART Temperature Warning Flag
Set UART Obstacle Near Warning Flag
Delay 10s
Delay 10s
Off Course Warning Flag Set & UART Off Course Warning Flag Unset?
No
No
Obstacle Near Warning Flag Unset & UART Obstacle Clear Warning Flag Unset?
Yes
Yes
Sends the Off Course Warning text to NUSCloud via UART Protocol
Sends the Obstacle Clear text to NUSCloud via UART Protocol
Set UART Off Course Warning Flag
Set UART Obstacle Clear Flag
End

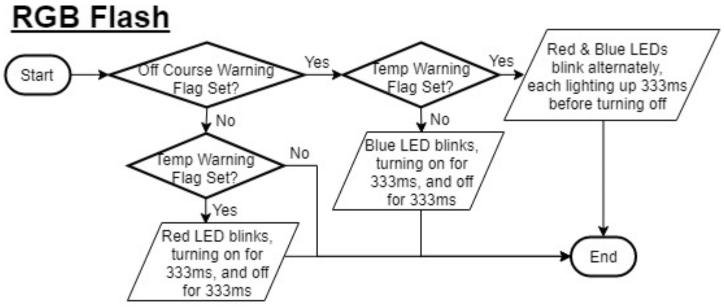
OLED Display


Start
Mode?
Stationary
Launch
Return
Temp Warning Flag Set?
Temp Warning Flag Set?
Obstacle Near Warning Flag Set?
No
Yes
No
Yes
No
Yes
Print Temperature Warning text on OLED
Print Temperature Warning text on OLED
Print Obstacle Near Warning text on OLED
Print "STATIONARY" & Current Temperature on OLED
Off Course Warning Flag Set?
No
Yes
Print "RETURN" on OLED
Print Off Course text on OLED
Print "LAUNCH", Current Temperature & Accelerometer values on OLED
End

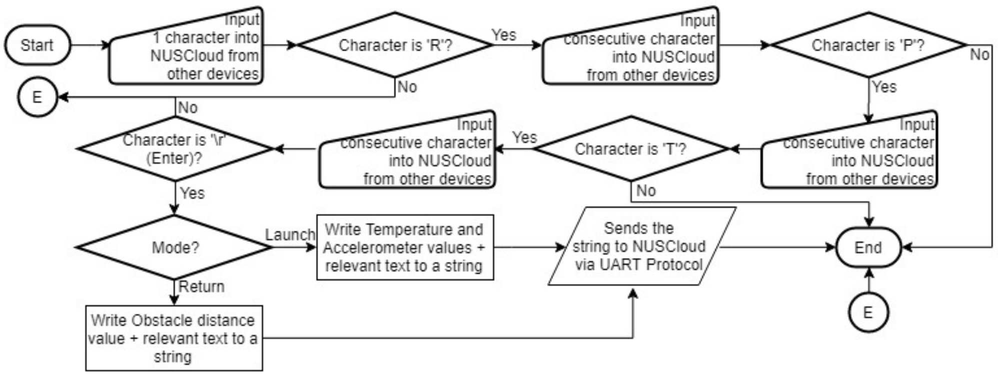
Countdown


Start
16s has passed?
No
Yes
Update digit shown on 7-segment display
Check Temperature
OLED Display
Temp Warning Flag Set?
No
Yes
Reset Temperature Warning Flag & Countdown
Successful
Aborted
Update 7-segment display to show 'F'
Reset Countdown

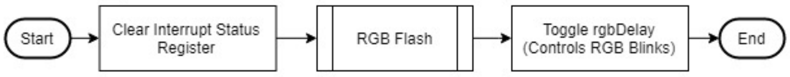
RGB Flash


Start
Off Course Warning Flag Set?
Yes
Temp Warning Flag Set?
Yes
Red & Blue LEDs blink alternately, each lighting up 333ms before turning off
No
Temp Warning Flag Set?
No
Blue LED blinks, turning on for 333ms, and off for 333ms
No
Yes
Red LED blinks, turning on for 333ms, and off for 333ms
End

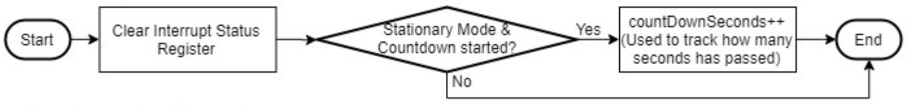
UART Callback


Start
Input 1 character into NUSCloud from other devices
Character is 'R'?
Yes
Input consecutive character into NUSCloud from other devices
Character is 'P'?
No
E
No
Yes
No
Character is '\r' (Enter)?
Input consecutive character into NUSCloud from other devices
Yes
Character is 'T'?
Input consecutive character into NUSCloud from other devices
No
Yes
Mode?
Launch
Write Temperature and Accelerometer values + relevant text to a string
Sends the string to NUSCloud via UART Protocol
End
Return
E
Write Obstacle distance value + relevant text to a string

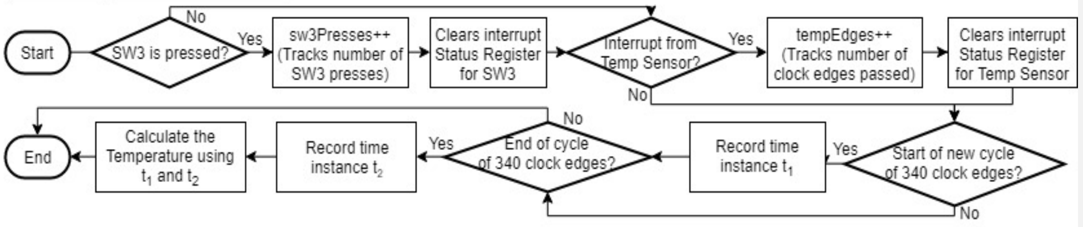
Timer0_IRQHandler


Start
Clear Interrupt Status Register
RGB Flash
Toggle rgbDelay (Controls RGB Blinks)
End

Timer1_IRQHandler


Start
Clear Interrupt Status Register
Stationary Mode & Countdown started?
Yes
countDownSeconds++ (Used to track how many seconds has passed)
End
No

EINT3_IRQHandler


No
Start
SW3 is pressed?
Yes
sw3Presses++ (Tracks number of SW3 presses)
Clears interrupt Status Register for SW3
Interrupt from Temp Sensor?
Yes
tempEdges++ (Tracks number of clock edges passed)
Clears interrupt Status Register for Temp Sensor
No
No
End
Calculate the Temperature using t1 and t2
Record time instance t2
Yes
End of cycle of 340 clock edges?
Record time instance t1
Yes
Start of new cycle of 340 clock edges?
No

1. Introduction:

Assignment 2 requires us to simulate a rocket that goes to and returns from space using the LPC1769 microprocessor (by NXP Semiconductors) to program the LPCXpresso Base Board developed by *Embedded Artists.*

UART Communication is also possible between the Base Board and a personal computer using another attached peripheral. This peripheral is the XBee Series 1 Radio-Frequency (RF) Module developed by *Digi International*, based on the IEEE 802.15.4 standard for physical layer and MAC control specifications targeting low-cost, low-speed and low-power communication devices. The indoor communication range is up to 30 meters.

Collectively, the Base Board, microprocessor and the XBee Module forms the prototype known as NUSpace. NUSpace uses the following peripherals: MMA7455L Three Axis Low-g Digital Output Accelerometer, MAXIM MAX6576 Temperature Sensor, intersil ISL29003 Light Sensor, 7-Segment LED Display, Univision 96x64 White Organic Light-Emitting Diode (OLED), Red-Green-Blue (RGB) LED, PCA9532 16-bit LED Array, 2 switches (SW3 & SW4) and the Xbee RF module.

NUSpace has sensors installed which help pilots take the appropriate action where required. The accelerometer on board senses whether the rocket is going off course in the X & Y direction (The Z direction is the upward direction of the rocket). The temperature sensor monitors the temperature of its fuel tank such that any overheating can be resolved immediately before any fire outbreaks can occur. The light sensor is a simulated radar, monitoring the distance of the rocket from any potential obstacle. The nearer the obstacle, the higher the light intensity since the radar signal reflected off the obstacle’s surface is stronger the closer the obstacle is. The pilots can then take evasive measures to avoid the obstacle. Once the obstacle is avoided, the pilots also get a notification as such so that they can concentrate on monitoring other on-board systems.

To prepare the rocket for its missions to space, it can toggle between modes. The rocket is first at Stationary mode when it is docked in Earth. The button SW3 is used to launch it into space and change the rocket into Launch mode. When the rocket has completed its mission, button SW3 is used to chart it on a course back to Earth in Return mode. Once it has successfully docked, button SW3 can be pressed to change NUSpace to Stationary mode.

The simulated NUSpace is shown below:

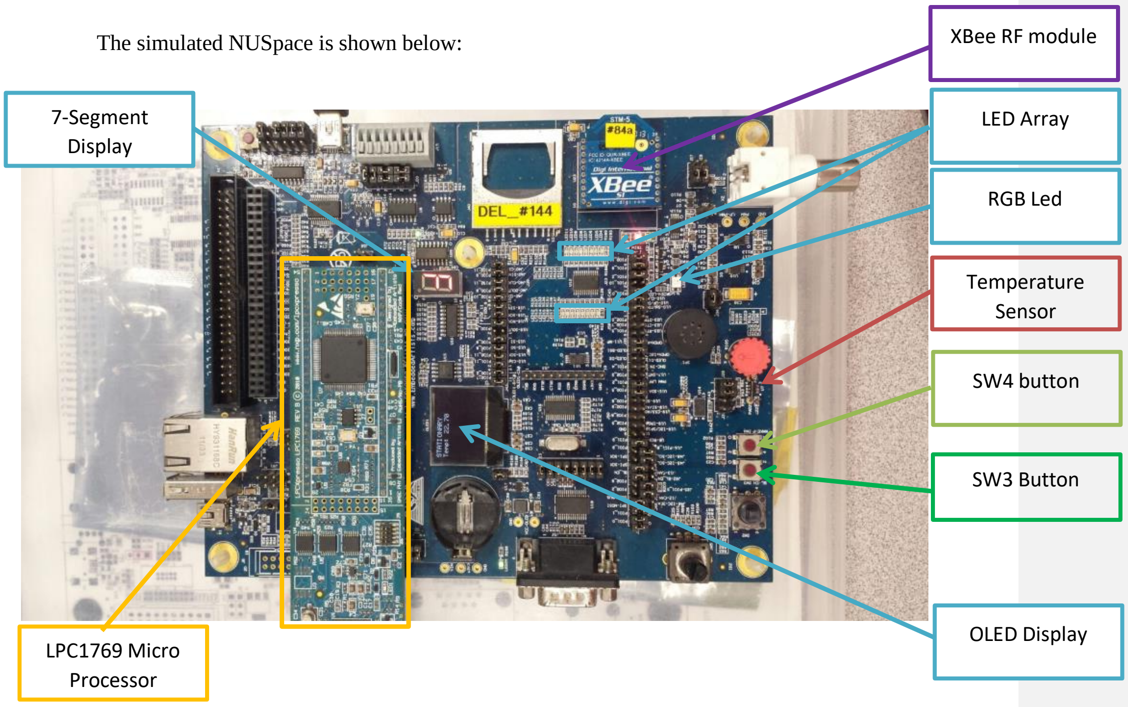


Figure 1: The peripherals of NUSpace

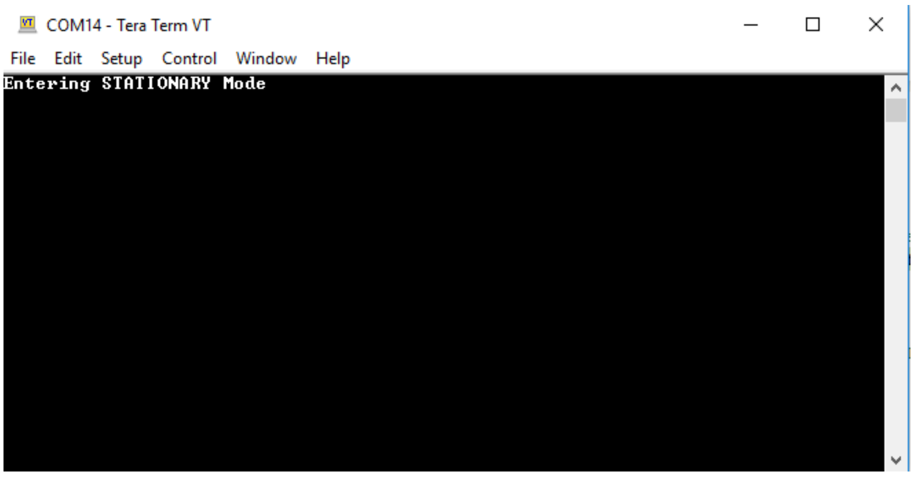
2. User Manual

NUSpace has 3 modes, namely Stationary, Launch and Return mode. SW3 is used to switch from Stationary to Launch mode, Launch mode to Return mode, and finally Return mode to Stationary Mode.

When NUSpace is first booted up, it goes into Stationary Mode. Pressing SW3 once causes it to go into Launch Mode after a 16 second countdown. When NUSpace is in Launch mode, pressing SW3 twice within one second causes it to go into Return Mode straight away. While in Return Mode, pressing SW3 once will cause it to return to Stationary Mode straight away.

2.1 Stationary Mode

Since logically the system should not be off when NUSpace is in flight, Stationary mode is the default mode when the system is turned on. The rocket also enters Stationary mode after SW3 is pressed once while it was in Return mode. After it just enters Stationary mode, it sends the message "Entering STATIONARY Mode" to NUSCloud as shown in Figure 2 below:



```
Entering STATIONARY Mode
```

Figure 2: NUSCloud receives the message from NUSpace

The system looks like this in Stationary mode:

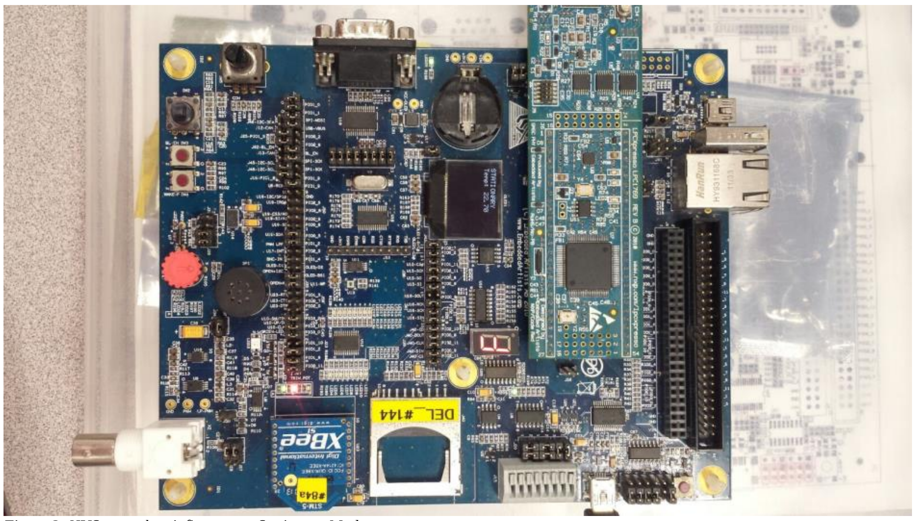


Figure 3: NUSpace when it first enters Stationary Mode

NUSpace is docked at its station and ready for launch! The current mode (Stationary) is shown on the OLED. Below the mode, the OLED displays the temperature of its fuel tank. The 7-segment display simply shows ‘0’.

If the temperature is more than the threshold of 28℃, there will be an additional line below this line which tells the pilots that the temperature is too high. The red RGB LED will then blink, turning on for 333ms, then turning off for 333ms, and so on. This is shown in Figure 4 below:

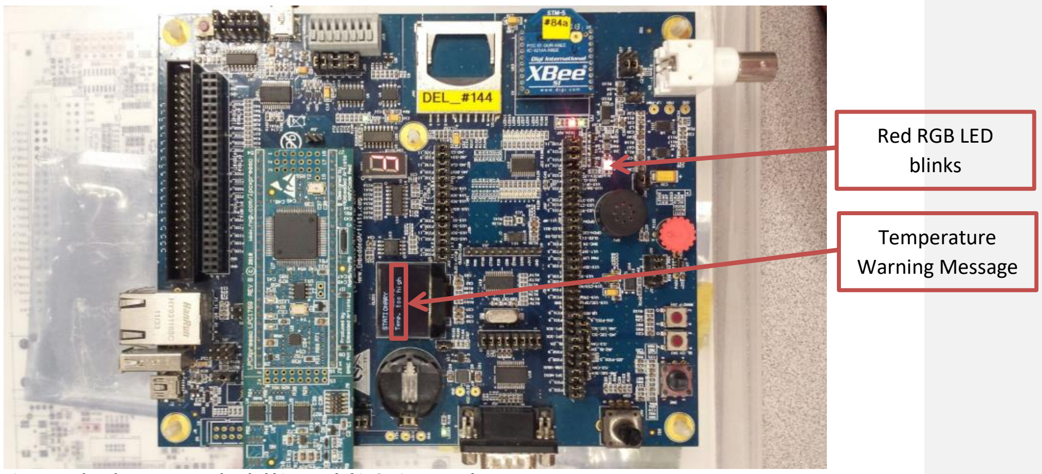


Figure 4: When the temperature threshold is exceeded in Stationary Mode

The temperature warning will persist even after the temperature drops below the threshold, and must be cleared manually by pressing SW4 button once. Pressing SW4 when the temperature is still above the threshold has no effect.

When the button SW3 is pressed once, a countdown of 16 seconds is initiated. The time left for NUSpace to go into Launch mode is shown on the 7-segment display (in hexadecimal). Once the countdown is over, the rocket goes into Launch mode.

However, during this 16 second countdown, if the temperature of the fuel tank goes above the threshold, then the countdown is aborted, and the rocket remains in Stationary mode to prevent the rocket from potentially exploding when the fuel tank bursts into flames. The board reverts to the initial Stationary state shown in Figure 3.

NUSpace returns to Stationary mode when SW3 is pressed once while it is in Return Mode.

2.2 Launch Mode

Launch mode is entered from Stationary mode when SW3 switch was pressed once and the countdown was not interrupted. After it just enters Stationary mode, it sends the message “Entering LAUNCH Mode” to NUSCloud as shown in Figure 6.

NUSpace has successfully launched from its station and is now on its space mission! The current mode is shown on the OLED. Below it shows the temperature of its fuel tank like in Stationary mode, but in addition to that the X and Y offset of the rocket is also shown in the units of g (g-force).

Like in Stationary mode, if the temperature is more than the threshold of 28°C, there will be an additional line below the Y offset reading which tells the pilots that the temperature is too high. The red RGB LED will then blink, turning on for 333ms, then turning off for 333ms, and so on.

In addition, if the X or Y offset is more than the offset threshold of 0.4g, there will be an additional line below the Y offset reading to tell the pilots to readjust their trajectory or else their rocket might go out of control and crash. The blue RGB LED will then blink, turning on for 333ms, then turning off for 333ms, and so on.

If both thresholds are exceeded, then both warnings are shown simultaneously. The red and blue RGB LEDs will then blink alternately, each turning on for 333ms. This case is shown below in Figure 5 below:

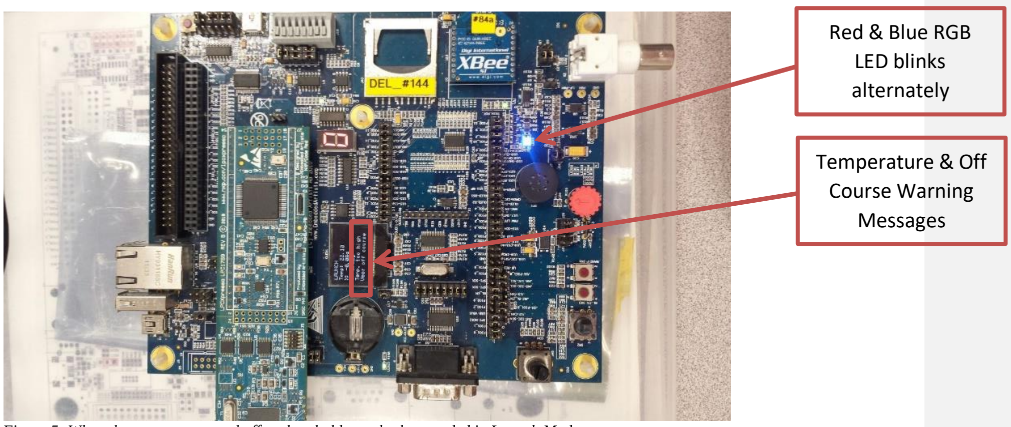
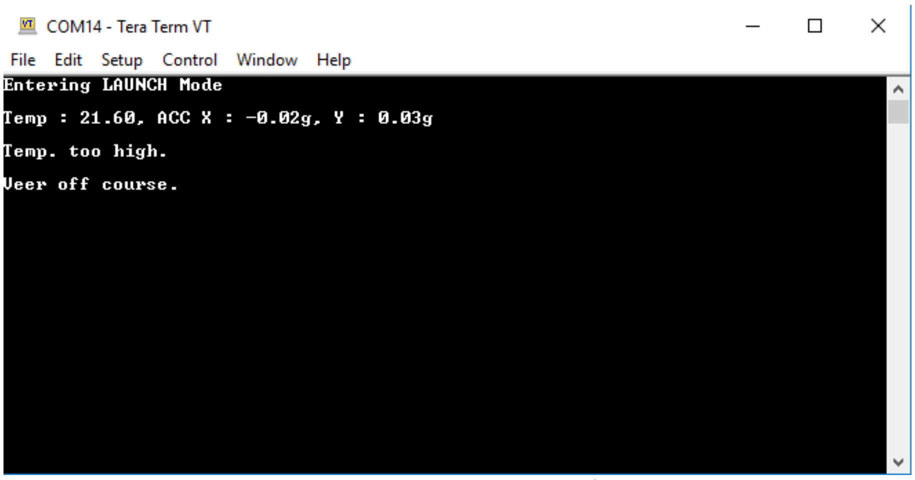

Red & Blue RGB LED blinks alternately
Temperature & Off Course Warning Messages
DEL_#144
XBee

Figure 5: When the temperature and offset thresholds are both exceeded in Launch Mode

Like in Stationary mode, both warnings will persist even after the respective value drops below the threshold. Both warnings can be cleared by pressing SW4 briefly once.

Commented [XY1]: Both warnings will be cleared when sw4 is pressed since they both check for sw4 state. Previously only 1 is cleared is cos of polling lag due to temp_read polling function which blocks for a long duration of time

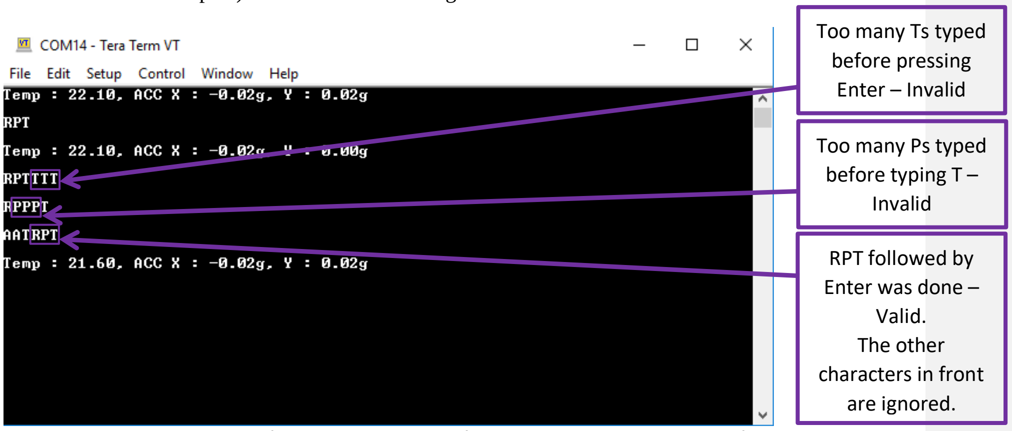
While in Launch mode, NUSpace periodically sends a status report about its fuel tank temperature and X & Y offset values to NUSCloud, once every 10 seconds. Immediately after the temperature warning pops up, the message "Temp. too high." is sent to NUSCloud. This only happens once until the warning is cleared. The message "Veer off course." is sent to NUSCloud immediately after the X or Y offset exceeds their threshold as well. All these are shown in Figure 6 below:



```
Entering LAUNCH Mode
Temp : 21.60, ACC X : -0.02g, Y : 0.03g
Temp. too high.
Veer off course.
```

Figure 6: NUSCloud receives the initial message, periodic messages & warning messages from NUSpace in Launch Mode

The NUSpace team can also demand an immediate status report by typing "RPT" followed by Enter on their NUSCloud computers back on Earth. Any typo mistakes in-between will cause no response. So long the characters entered are 'R', 'P', 'T' and finally Enter in that specific order, any other input is ignored. The status reports (including the sample commands typed in by the NUSpace team, as well as valid and invalid inputs) are shown below in Figure 7:


COM14 - Tera Term VT
File Edit Setup Control Window Help
Too many Ts typed before pressing Enter – Invalid
Too many Ps typed before typing T – Invalid
RPT followed by Enter was done – Valid.
The other characters in front are ignored.

```
Temp : 22.10, ACC X : -0.02g, Y : 0.02g
RPT
Temp : 22.10, ACC X : -0.02g, Y : 0.00g
RPTTTT
RPPPT
AATRPT
Temp : 21.60, ACC X : -0.02g, Y : 0.02g
```

Figure 7: NUSCloud receives responses from NUSpace by typing RPT followed by Enter in Launch Mode ONLY if the input is valid

Pilots can abort the mission by pressing SW3 twice within one second, which will cause NUSpace to go straight into Return mode.

2.3 Return Mode

Return mode is entered from Launch mode when SW3 switch was pressed twice within one second. After it just enters Return mode, it sends the message “Entering RETURN Mode” to NUSCloud as shown in Figure 8.

NUSpace has completed its mission and is on its way back to Earth! Unfortunately, the pressure is not over, the pilots must be careful not to collide into any potential obstacles! The current mode is shown on the OLED. The 7-segment display simply shows ‘0’. The LED array shows the distance of NUSpace from the nearest obstacle; the more LEDs lighted up, the nearer it is to the obstacle. The system looks like this in Return mode:

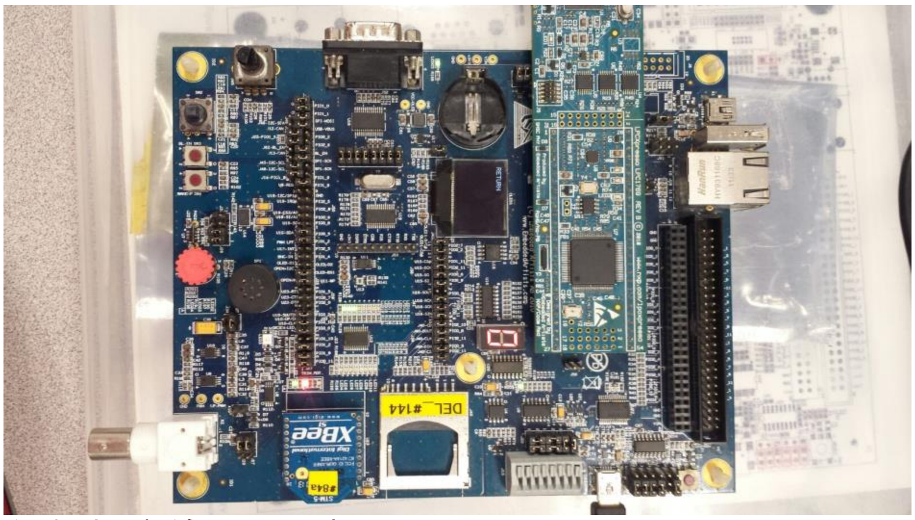


Figure 8: NUSpace when it first enters Return Mode

If the obstacle is very near (when the light sensor registers a reading of above 3000 lux), all the LEDs in the LED Array will also be lit up and there will be an additional line below the line that indicates Return mode which tells the pilots to take evasive measures. This is shown in Figure 9 below:

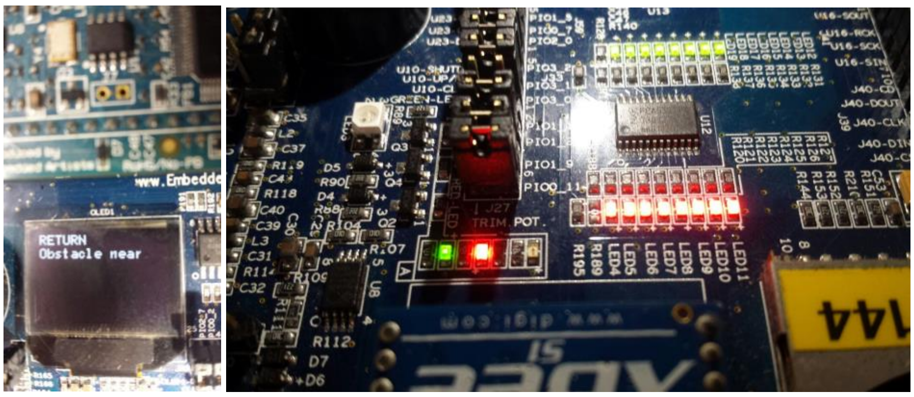
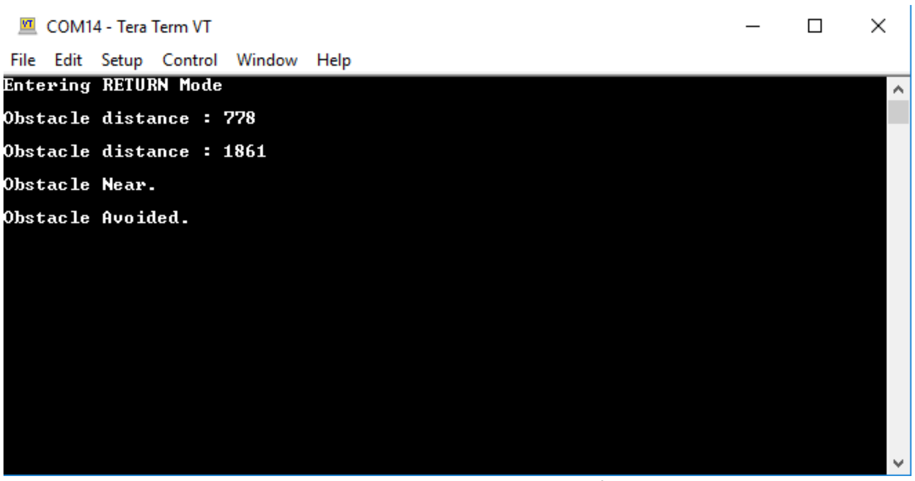


Figure 9: The OLED and LED Array when an obstacle is too near in Return Mode

Unlike in the other 2 modes, once the light sensor reading is below the threshold reading of 3000 lux, the message disappears and the LED Array will be updated according to the new obstacle distance, reverting the system back to what it was when it first entered Return mode like in Figure 8 (other than the potentially different LED Array due to varying obstacle distance).

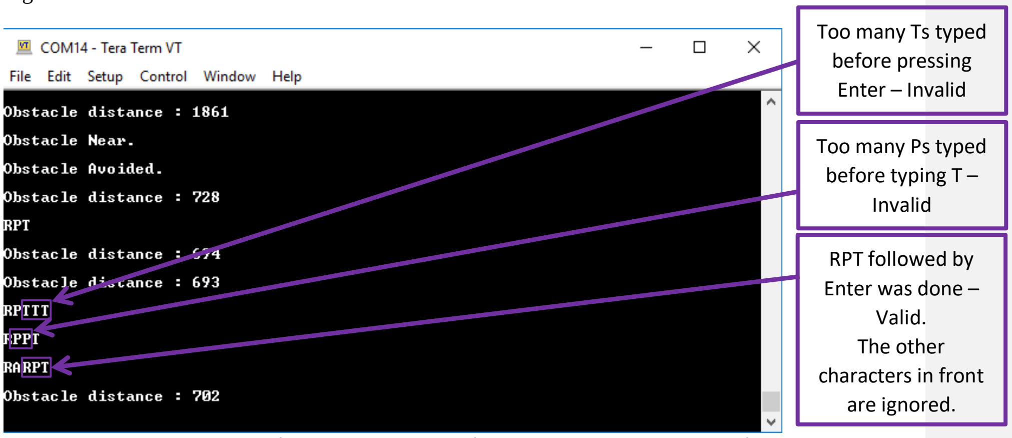
While in Return mode, NUSpace periodically sends a status report about its distance to the nearest obstacle to NUSCloud, once every 10 seconds. Immediately after the obstacle near warning pops up, the message “Obstacle Near.” is sent to NUSCloud. Immediately after the obstacle near warning clears, the message “Obstacle Avoided.” is sent to NUSCloud. This only happens once until the opposite event has occurred. All these is shown in Figure 10 below:



```
Entering RETURN Mode
Obstacle distance : 778
Obstacle distance : 1861
Obstacle Near.
Obstacle Avoided.
```

Figure 10: NUSCloud receives the initial message, periodic messages & warning messages from NUSpace in Return Mode

The NUSpace team can also demand an immediate status report by typing “RPT” followed by Enter on their NUSCloud computers back on Earth. Any typo mistakes in-between will cause no response. So long the characters entered are ‘R’, ‘P’, ‘T’ and finally Enter in that specific order, any other input characters in front or behind this string will be ignored. The status reports (including the sample commands typed in by the NUSpace team, as well as valid and invalid inputs) are shown below in Figure 11:


COM14 - Tera Term VT
File Edit Setup Control Window Help
Too many Ts typed before pressing Enter – Invalid
Too many Ps typed before typing T – Invalid
RPT followed by Enter was done – Valid. The other characters in front are ignored.

```
Obstacle distance : 1861
Obstacle Near.
Obstacle Avoided.
Obstacle distance : 728
RPT
Obstacle distance : 674
Obstacle distance : 693
RPTTT
RPPT
RARPT
Obstacle distance : 702
```

Figure 11: NUSCloud receives responses from NUSpace by typing RPT followed by Enter in Return Mode ONLY if the input is valid

Pilots can finally complete their mission after docking NUSpace at its station by pressing SW3, which causes NUSpace to go to Stationary mode.

4. Implementation

The program uses a combination of General Purpose Input Output (GPIO), Inter-Integrated Circuits (I^2C), Universal Asynchronous Receiver/Transmitter (UART) and Serial Peripheral Interface (SPI) protocols, to achieve the assortment of peripherals in the program.

4.1 SysTick_Handler

The default SysTick Handler provided by the CMSIS library is used to keep track of the milliseconds that has passed ever since the system has been running. It is used in many parts of our code due to the time constraints and periodic time intervals required by the system.

Commented [ZGTJL2]: Is this the right library?

Commented [XY3R2]: Its under CMSIS library which is for core on chip peripherals

For example, it has been used to keep track of when the second SW3 button press is in Launch mode after pressing SW3 once. This is because the second SW3 press must be done within one second of the first press for NUSpace to enter Return mode. We would first detect the first press of SW3, keep track of the time it was pressed returned by SysTick Handler (Point A), and enter a while loop to listen for a second SW3 press (Point B). Inside the while loop, the rest of the functions in Launch mode must still run (Point C). As SysTick Handler keeps increasing msTicks (Point B), the while loop will be broken out after another second has passed as seen in the condition (Point B). However, when another SW3 press is detected (Point D) before the while loop could be broken out of, all the warnings are reset and the mode Return is set (Point E).

Below shows the exact implementation of the code:
```
if (sw3Presses >= 1) {
        if (withinOneSecond == 0){
                oneSecond = msTicks;                                    // Point A
                withinOneSecond = 1;
        }
        while (((msTicks - oneSecond) <= 1000) && (withinOneSecond == 1)){  // Point B
                temp_warning();                                         // Point C
                acc_reader();                                           // Point C
                display_oled();                                         // Point C
                uart_display();                                         // Point C

                if (sw3Presses == 2) {                                  // Point D
                        sw3Presses = 0;
                        withinOneSecond = 0;
                        TEMP_WARNING = 0;                               // Point E
                        ACC_WARNING = 0;                                // Point E
                        modeChangeFlag = 0;
                        mode = RETURN;                                  // Point E
                }
        }
        sw3Presses = 0;
        withinOneSecond = 0;
}
```

msTicks is also used for UART Display in the Tera Term Terminal since a periodic message is required every 10 seconds. This is done by keeping track of the time the last UART message was sent to the terminal, storing that time into a variable called tenSecond. As SysTick Handler increases msTicks, the difference between msTicks and tenSecond will eventually exceed 10,000 milliseconds and that is when the code for sending UART message will execute.

4.2 7-Segment Display

The 7-segment display is only updated during the countdown in Stationary mode. Otherwise, it stays constant. Since the system always starts in Stationary mode and is only possible to go to other modes from Stationary mode, we chose to only set the 7-segment display during Stationary mode, since after exiting Stationary mode the 7-segment display would already show ‘0’, which is the same as what is required for the other 2 modes.

When the system is in countdown, it goes into a while loop. The functions in Stationary mode are still required to run while in countdown (especially temperature sensor) thus these functions are inside the while loop (Point A). The while loop only exits when the number of seconds elapsed is more than 15, or when a temperature warning pops up (Point B). In the while loop, a variable named countDownSeconds serves as an index to tell the 7-segment display which character to display (Point C). countDownSeconds is also used in the condition to exit the while loop as it increments 1 every one second using a timer (Point B). If the temperature warning is detected, then the 7-segment display is reset to showing ‘F’ and countDownSeconds is also reset (Point D). If the countdown is successful, countDownSeconds is reset and the mode is changed to Launch mode (Point E).

The exact code implementation is shown below:

```
const char countDownDisplay[] = "FEDCBA9876543210";                        // Point C

void countdown_run(void) {
        while ((countDownSeconds <= 15)) {                                 // Point B
                led7seg_setChar(countDownDisplay[countDownSeconds],FALSE);  // Point C
                temp_warning();                                            // Point A
                display_oled();                                            // Point A
                if (TEMP_WARNING == 1){                                    // Point B
                        countDownSeconds = 0;                              // Point D
                        countDownBegin = 0;
                        led7seg_setChar('F',FALSE);                        // Point D
                        break;                                             // Point B
                }
        }

        if (countDownSeconds >= 15) {                                      // Point E
                TEMP_WARNING = 0;                                          // Point E
                countDownSeconds = 0;                                      // Point E
                countDownBegin = 0;
                modeChangeFlag = 0;
                mode = LAUNCH;                                             // Point E
        }
}
```

4.3 LED Array

The LED Array is only used during Return mode. Otherwise, all LEDs are turned off. In our function light_reader(), a variable named lightValue simply takes on value returned by light_read() function in the EaBaseBoard library. As light_reader() is part of the code in the mode choose function which runs infinitely, the lightValue updates by polling. After which, in the led_array() function in Return mode (which is also in mode choose and thus also works by polling), the lightValue is read. Based on the value of lightValue, the appropriate number of LEDs in the LED Array lights up by setting the bits in the appropriate position. The scale is approximately linear.

Commented [ZGTJL4]: Is the library correct?

Commented [XY5R4]: yup

The exact code implementation is shown below:

```
lightValue = light_read();

void led_array(void){
        if (lightValue < 190){
                pca9532_setLeds (0b0000000000000000, 0xFFFF);
        }
        else if (lightValue >= 190 && lightValue < 380) {
                pca9532_setLeds (0b1000000000000000, 0xFFFF);
        }
        else if (lightValue >= 380 && lightValue < 570) {
                pca9532_setLeds (0b1100000000000000, 0xFFFF);
        }
        else if (lightValue >= 570 && lightValue < 750) {
                pca9532_setLeds (0b1110000000000000, 0xFFFF);
        }
        else if (lightValue >= 750 && lightValue < 940) {
                pca9532_setLeds (0b1111000000000000, 0xFFFF);
        }
        else if (lightValue >= 940 && lightValue < 1130) {
                pca9532_setLeds (0b1111100000000000, 0xFFFF);
        }
        else if (lightValue >= 1130 && lightValue < 1310) {
                pca9532_setLeds (0b1111110000000000, 0xFFFF);
        }
        else if (lightValue >= 1310 && lightValue < 1500) {
                pca9532_setLeds (0b1111111000000000, 0xFFFF);
        }
        else if (lightValue >= 1500 && lightValue < 1690) {
                pca9532_setLeds (0b1111111100000000, 0xFFFF);
        }
        else if (lightValue >= 1690 && lightValue < 1880) {
                pca9532_setLeds (0b1111111100000001, 0xFFFF);
        }
        else if (lightValue >= 1880 && lightValue < 2070) {
                pca9532_setLeds (0b1111111100000011, 0xFFFF);
        }
        else if (lightValue >= 2070 && lightValue < 2250) {
                pca9532_setLeds (0b1111111100000111, 0xFFFF);
        }
        else if (lightValue >= 2250 && lightValue < 2440) {
                pca9532_setLeds (0b1111111100001111, 0xFFFF);
        }
        else if (lightValue >= 2440 && lightValue < 2630) {
                pca9532_setLeds (0b1111111100011111, 0xFFFF);
        }
        else if (lightValue >= 2630 && lightValue < 2820) {
                pca9532_setLeds (0b1111111100111111, 0xFFFF);
        }
        else if (lightValue >= 2820 && lightValue < 3000) {
                pca9532_setLeds (0b1111111101111111, 0xFFFF);
        }
        else if (lightValue >= 3000) {
                pca9532_setLeds (0b1111111111111111, 0xFFFF);
        }
}
```

4.4 RGB LED

The RGB LEDs are only used during Stationary and Launch modes. In Return mode, they simply stay off. Furthermore, only the red and blue RGB LEDs are being used.

The RGB LEDs flash based on a timer which will toggle a flag called rgbDelay. When this flag is toggled every 333ms, the inner if-else condition (which involves the flag rgbDelay) statements in our function rgb_flash() determines which RGB LEDs will light up (Point A). Based on which warnings are active, the outer if-else condition will determine the pattern of the RGB LED lightings (Point B). We first check whether both warnings are active before checking whether each individual warning is active without the other active (Point B).

Even if only one of the warnings is active, we always clear the other RGB LED that will not be flashing at all (Point C). This is because during the case where both warnings were active at first and only one was cleared, there is a possibility that the LED associated with that warning remains lit since

LED was previously set, and the clearing of that LED will be unable to happen. We would not have cleared it when we continuously enter the outer if-else condition of that warning alone if the lines at all Points C were not existent.

The exact code implementation is shown below:

```
void rgb_flash (void) {
        if (TEMP_WARNING == 1 && ACC_WARNING == 1) {         // Point B
                if (rgbDelay) {                               // Point A
                        GPIO_SetValue(2, (1<<0));
                        GPIO_ClearValue(0, (1<<26));
                } else {
                        GPIO_SetValue(0, (1<<26));
                        GPIO_ClearValue(2, (1<<0));
                }
        }
        else if (TEMP_WARNING == 1){                          // Point B
                if (rgbDelay) {                               // Point A
                        GPIO_SetValue(2, (1<<0));
                        GPIO_ClearValue(0, (1<<26));          // Point C
                } else {
                        GPIO_ClearValue(2, (1<<0));
                        GPIO_ClearValue(0, (1<<26));          // Point C
                }
        }
        else if (ACC_WARNING == 1){                           // Point B
                if (rgbDelay) {                               // Point A
                        GPIO_SetValue(0, (1<<26));
                        GPIO_ClearValue(2, (1<<0));           // Point C
                } else {
                        GPIO_ClearValue(2, (1<<0));           // Point C
                        GPIO_ClearValue(0, (1<<26));
                }
        } else {
                GPIO_ClearValue(2, (1<<0));
                GPIO_ClearValue(0, (1<<26));
        }
}
```

4.5 UART Handler

Commented [ZGTJL6]: I'm not very happy with the way I wrote this one. See whether ya can find better way to rephrase.
Also, are the UART functions are in what library specifically?

Commented [XY7R6]: I don't know as well so I think we just remove the library part ba

Commented [XY8R6]: Your explanations are okay what, honestly I also don't really know how uart works on the low level side so can't really chip in much ps

NUSpace's UART has 2 aspects: Communication initiated by the Base Board UART peripheral to send its messages to the Tera Term Terminal, or communication initiated by the Tera Term Terminal to enquire about the status of the Base Board.

For the communication initiated by the Base Board UART peripheral, it is done by polling since our uart_display() function is part of the code in the mode choose function which runs infinitely. As stated in Section 4.1, UART_Send will only run once every 10 seconds using a variable called tenSecond which keeps track of the last time the UART message was sent, as well as the comparison between the tenSecond variable and the continuously changing msTicks variable (Point A). When the UART code is executed, a string called uartDisplayValues is updated and then this string is sent to the Tera Term

Terminal to be printed (Point B). When the appropriate warnings are triggered, the predefined strings uartTempWarning, uartAccWarning, uartObstWarning and uartObstClear are also sent to the Tera Term Terminal (Point C). However, since these warnings must only be sent once until the warning(s) is/are cleared and triggered again, a flag is set for each warning if the respective warning is sent (Point D).

Another function called uart_display_mode() will run once whenever there is a mode change. This run once functionality is achieved using a flag called modeChangeFlag, which gets reset every time right before the system changes mode and gets set after the code inside the if condition (which uses this flag) is executed. This function simply straight away sends the already predefined message to the Tera Term Terminal.

For communication initiated by the Tera Term Terminal, this must be done by interrupt since the system has no idea when the enquiry about its status will come. This interrupt is set up simply by just calling UART_SetupCbs() function when we initialize the UART Protocol. In our uart_callback() function which handles the inputs into Tera Term Terminal, the interrupt comes in for every character that is received by the system. (Point E) Each character is converted to a string simply by adding a string terminating character at the back (Point F). We then use the strcmp function in the C library to compare that character string to the characters we need to receive (Point G). To ensure that the order of the characters is precisely ‘R’, ‘P’, ‘T’ and then the Enter key, we set flags after each character input is correct (Point H). The flags ensure the previous character is indeed correct (by checking whether the corresponding flag is set), which includes not entering the same character again (by checking whether its own flag is not set). When the sequence is correct, the code checks what mode is the system in and writes the appropriate string to uartDisplayValues to be sent back to the Tera Term Terminal (Point I). At any time, if the character input is wrong or after the message is successfully sent, all the flags are reset to allow the next potential sequence of “RPT\r” to be entered again (Point J).

The exact code implementation is shown below:

```
UART_SetupCbs(LPC_UART3, 0, (void *) &uart_callback);

const char uartDisplayStationary[] = "Entering STATIONARY Mode \r\n";
const char uartDisplayLaunch[] = "Entering LAUNCH Mode \r\n";
const char uartDisplayReturn[] = "Entering RETURN Mode \r\n";
const char uartTempWarning[] = "Temp. too high.\r\n";
const char uartAccWarning[] = "Veer off course.\r\n";
const char uartObstWarning[] = "Obstacle Near.\r\n";
const char uartObstClear[] = "Obstacle Avoided.\r\n";

void uart_display_mode(void){
        if (mode == STATIONARY) {
                UART_Send(LPC_UART3, (uint8_t *)uartDisplayStationary , strlen(uartDisplayStationary), 
BLOCKING);

        } else if (mode == LAUNCH) {
                UART_Send(LPC_UART3, (uint8_t *)uartDisplayLaunch , strlen(uartDisplayLaunch), BLOCKING);

        } else if (mode == RETURN) {
                UART_Send(LPC_UART3, (uint8_t *)uartDisplayReturn , strlen(uartDisplayReturn), BLOCKING);
        }
}

void uart_display(void) {
        if (mode == LAUNCH){
                if ((msTicks - tenSecond) > 10000) {                                                    // Point A
                        tenSecond = msTicks;
                        sprintf(uartDisplayValues, "Temp : %2.2f, ACC X : %2.2fg, Y : %2.2fg \r\n", 
tempValue, x/g, y/g);                                                                                   // Point B
                        UART_Send(LPC_UART3, (uint8_t *)uartDisplayValues , strlen(uartDisplayValues), 
BLOCKING);                                                                                              // Point B
                }

                if (TEMP_WARNING == 1 && uartTempWarningFlag == 0){                                     // Point C
                        UART_Send(LPC_UART3, (uint8_t *)uartTempWarning , strlen(uartTempWarning), 
BLOCKING);                                                                                              // Point C
                        uartTempWarningFlag = 1;                                                        // Point D
                }

                if (ACC_WARNING == 1 && uartAccWarningFlag == 0){                                       // Point C
                        UART_Send(LPC_UART3, (uint8_t *)uartAccWarning , strlen(uartAccWarning), 
BLOCKING);                                                                                              // Point C
                        uartAccWarningFlag = 1;                                                         // Point D
                }
        } else if (mode == RETURN) {
                if ((msTicks - tenSecond) > 10000) {                                                    // Point A
                        tenSecond = msTicks;
                        sprintf(uartDisplayValues, "Obstacle distance : %d \r\n", lightValue);
                                                                                                        // Point B
                        UART_Send(LPC_UART3, (uint8_t *)uartDisplayValues , strlen(uartDisplayValues), 
BLOCKING);                                                                                              // Point B
                }

                if (OBST_WARNING == 1 && uartObstWarningFlag == 0){                                     // Point C
                        UART_Send(LPC_UART3, (uint8_t *)uartObstWarning , strlen(uartObstWarning), 
BLOCKING);                                                                                              // Point C
                        uartObstWarningFlag = 1;                                                        // Point D

                } else if (OBST_WARNING == 0 && uartObstClearFlag == 0) {                               // Point C
                        UART_Send(LPC_UART3, (uint8_t *)uartObstClear , strlen(uartObstClear), BLOCKING);
                                                                                                        // Point C
                        uartObstClearFlag = 1;                                                          // Point D
                }
        }
}

void uart_callback (void) {
        uint8_t data[2];

        UART_Receive(LPC_UART3, data, 1, BLOCKING);                                                     // Point E
        data[1] = '\0';                                                                                 // Point F

        if (strcmp((char *)data ,"R") == 0) {                                                           // Point G
                flagR = 1;                                                                              // Point H
        } else if ((strcmp((char *)data ,"P") == 0) && flagR == 1 && flagP == 0) {                      // Point G
                flagP = 1;                                                                              // Point H
        } else if ((strcmp((char *)data ,"T") == 0) && flagP == 1 && flagT == 0) {                      // Point G
                flagT = 1;                                                                              // Point H
        }
        else if ((strcmp((char *)data ,"\r") == 0) && flagT == 1) {                  // Point G
                if(mode == LAUNCH) {                                                       // Point I
                        sprintf(uartDisplayValues, "Temp : %2.2f, ACC X : %2.2fg, Y : %2.2fg \r\n",
tempValue, x/g, y/g);                                                                      // Point I
                }
                else if (mode == RETURN) {                                                 // Point I
                        sprintf(uartDisplayValues, "Obstacle distance : %d \r\n", lightValue); // Point I

                }
                UART_Send(LPC_UART3, (uint8_t *)uartDisplayValues , strlen(uartDisplayValues), BLOCKING);
                                                                                           // Point B

                flagR = flagP = flagT = 0;                                                 // Point J
        } else {
                flagR = flagP = flagT = 0;                                                 // Point J
        }
}
```

5. Special Features

5.1 Timer for RGB and Countdown

While it is possible to simply use SysTick Handler to keep track of the time for all functions that run on periodic intervals, it is better to have dedicated timer interrupts to perform periodic commands more accurately. By using a separate tick variable (oneSecond) to keep track of let's say a one second interval, we are using polling in to find out the current ticks (msTicks) and comparing it with our tick variable (oneSecond) to check whether one second has passed. In this case, there might be a short delay due to polling time and the time interval that has passed will be slightly longer than 1 second. However, using a timer interrupt to keep tracking of the one second interval instead will resolve this issue since an interrupt is raised as soon as 1 second has passed.

In the implementation of the timers, we first set up a pre-scalar function to scale the timer values to micro seconds. Next, we initialise the respective timers by powering them up and selecting their respective pins. Then the timers are configured using the respective values (333333 µs for RGB and 1,000,000µs for countdown). Lastly, the timer interrupts are started by setting the counter enable and they are enabled in NVIC.

The exact code implementation is shown below:

```
unsigned int getPrescalar(uint8_t timerPeripheralClockBit){
    unsigned int peripheralClock, prescalar;

    if(timerPeripheralClockBit < 10)
        peripheralClock = (LPC_SC->PCLKSEL0 >> timerPeripheralClockBit) & 0x03;
    else
        peripheralClock = (LPC_SC->PCLKSEL1 >> timerPeripheralClockBit) & 0x03;
        switch ( peripheralClock ){
                case 0x00:
                        peripheralClock = SystemCoreClock/4;
                        break;
                case 0x01:
                        peripheralClock = SystemCoreClock;
                        break;
                case 0x02:
                        peripheralClock = SystemCoreClock/2;
                        break;
                case 0x03:
                        peripheralClock = SystemCoreClock/8;
                        break;
    }
    prescalar = peripheralClock/1000000 - 1;

    return prescalar;
}
void init_timer0(){
        LPC_SC->PCONP |= (1 << 1);            // Power Up Timer 0
        LPC_SC->PCLKSEL0 |= 0x01 << 2;        // Select the Timer 0 pin
        LPC_TIM0->MCR  = (1<<0) | (1<<1);     // Clear Timer Counter on Match Register 0 match
and Generate Interrupt
        LPC_TIM0->PR   = getPrescalar(2);     // Prescalar for 1us
        LPC_TIM0->MR0  = 333333;              // Load timer value to generate 1ms delay
        LPC_TIM0->TCR  = (1 << 0);            // Start timer by setting the Counter Enable
        NVIC_EnableIRQ(TIMER0_IRQn);
}

void init_timer1(){
        LPC_SC->PCONP |= (1 << 2);            // Power Up Timer 1
        LPC_SC->PCLKSEL0 |= 0x01 << 4;        // Select the Timer 1 pin
        LPC_TIM1->MCR  = (1<<0) | (1<<1);     // Clear Timer Counter on Match Register 0 match
and Generate Interrupt
        LPC_TIM1->PR   = getPrescalar(4);             // Prescalar for 1us
        LPC_TIM1->MR0  = 1000000;             // Load timer value to generate 1s delay
        LPC_TIM1->TCR  = (1 << 0);            // Start timer by setting the Counter Enable
        NVIC_EnableIRQ(TIMER1_IRQn);
}
```

Timer 0 is used for the RGB LED flashes. After timer0 interrupt is raised (every 333333 μs), the interrupt bit is cleared and the code for RGB LED flashes run through the function rgb_flash() which is already shown in Section 4.4. rgb_flash() has finished running, the rgbDelay flag is toggled to trigger the alternate flash sequence in the inner if-else condition in rgb_flash() function.

Timer 1 is used for the countdown in Stationary mode. After timer1 interrupt is raised (every 1,000,000μs), the interrupt bit is cleared and the system checks whether the countdown has started (and the system is in Stationary mode). If so, then the variable called countDownSeconds will be incremented by 1 to be used by the countdown function mentioned in Section 4.2.

Timer0 has higher priority over Timer1 due to RMS scheduling since it has a shorter period and occurs more often.

5.2 Temperature Sensor Interrupt

While there is a function in the EaBaseBoard library called temp_read(), it is a very slow function since it uses a for loop which will make the system get trapped in the loop for some time before exiting it, which causes a delay in other functions such as the countdown and the accelerometer functions. Thus, it is more logical to use an interrupt to avoid delays to more important functions such as the accelerometer function which will warn the pilots of potential crash due to deviation from original trajectory.

The temperature interrupt is first initialized to recognize rising and falling edges as detected by the temperature sensor, and then NVIC is configured to enable the interrupt.

Commented [ZGTJL9]: What clock is this???

Commented [XY10R9]: It's the falling and rising edges of the temp sensor. Basically the temp sensor detects 1 and 0s really quick

In the EINT3_IRQHandler (which also handles SW3 interrupt which will not be discussed here), when the interrupt detects a rising or falling edge (Point A) from the temperature sensor, a variable named tempEdges which is used to keep track of the total number of edges, is incremented by 1 (Point B). The interrupt bit for EINT3 is then cleared. The system needs to receive 340 rising or falling edges to calculate the temperature. At the start of the 340 edge cycle, the time instance t_1 is recorded and a flag is being set to stop t_1 from getting set again (Point C). Thereafter, after the 340th clock edge has arrived, the time instance t_2 is also recorded (Point D). The tempEdges variable is reset to zero and the flag to record t_1 is unset (Point E). It then performs an arithmetic operation using t_1 and t_2 to determine the temperature.

The exact code implementation is shown below:

```
void EINT3_IRQHandler(void){
	…
	if (((LPC_GPIOINT->IO0IntStatF >> 2) & 0x1) || ((LPC_GPIOINT->IO0IntStatR >> 2) &
0x1)) {                                                                   // Point A
		tempEdges++;                                                      // Point B
		LPC_GPIOINT ->IO0IntClr = 1<<2;

		if (newPeriod == 0) {
			t1 = getTicks();                                          // Point C
			newPeriod = 1;                                            // Point C
		}
		if (tempEdges >= (NUM_HALF_PERIODS)) {                            // Point D
			t2 = getTicks();                                          // Point D
			if (t2 > t1) {                                            // Point E
				t2 = t2 - t1;                                     // Point E
			} else {                                                  // Point E
				t2 = (0xFFFFFFFF - t1 + 1) + t2;                  // Point E
			}
			tempEdges = 0;                                            // Point E
			newPeriod = 0;                                            // Point E
```

```
                        tempValue = ((2*1000*t2) / (NUM_HALF_PERIODS*TEMP_SCALAR_DIV10) - 2731)
/ 10.0 ;                                                                            // Point E
                }
        }
}
```

6. Significant Problems Encountered

6.1 SW4 switch press detection

As SW4 is on GPIO Pin 1, it is unable to use interrupt and thus must be done by polling. Previously, when we have not implemented the temperature interrupt, polling takes a long time due to the temp_read() function in temperature sensor function hogging the system since it loops many times before exiting it. This made the responsiveness of SW4 to be very sluggish and had to be usually held on for around 1 second for it to work properly.

But now that the temperature sensor uses interrupt, the polling frequency of SW4 has increased significantly. However, there are times where we still need to hold on to SW4 for a moment (rather than quickly pressing and releasing it) before it is registered and for the warnings to be cleared. This is despite our best efforts to optimize our code (which is now simply all if-else statements).

6.2 Lag in 7-Segment Update

The 7-segment display update during countdown was too inconsistent or too slow initially.

We realized that this is also due to the temperature sensor function which previously used temp_read(), causing the system to be stuck in the loop for some time before being able to update the 7-segment display properly despite the timer for countdown functioning correctly (since it uses interrupt, so it is not affected). After changing temperature sensor to interrupt-based, the problem has disappeared.

Commented [ZGTJL11]: Can specifically state what was our code like when the countdown was slow, and when the countdown was inconsistent? I forgot liao

Commented [XY12R11]: The inconsistent one is cos we use timer to increment but its ok can just leave that one out

6.3 Cannot receive message from BaseBoard despite correct string entered

Despite entering “RPT\r” into the Tera Term Terminal sometimes, the response message from the BaseBoard will not appear. We realized that this occurs after we accidentally made a typo mistake in typing the command into the terminal.

We realized that our code was flawed since our UART Receive function straight away took in 4 characters at the same time. Any typo would then shift the subsequent correct “RPT\r” string by however many character typos are there. This also means that the problem will only get fixed after 4 erroneous character inputs. For example, we typed the following characters in order: “RPPT\rRPT\r”. The first string recognized becomes “RPPT”, the second string recognized becomes “\rRPT”, with the last character ‘\r’ being pushed out.

We only figured out how to take in character by character after some time after realizing we could use flags, but even then it wasn’t perfect when we realized we could type duplicate characters after the original character like “RPPPPPPPPT\r” and the message would still get sent. However we just needed to set a stricter condition involving more pre-existing flags and the problem was solved.

6.4 OLED Screen clearing when clearing unused RGB LEDs

As the RGB LED shares port with other components such as the OLED display, using the function rgb_setLeds() to turn off the green RGB LED will result in the OLED display to be cleared as well. Therefore, modifications have been made. The issue of the green RGB LED lighting up was addressed by removing a jumper that was connected to the green RGB LED. This allows the green RGB LED to be turned off while keeping the OLED display on. For the Red and Blue LEDs, their respective GPIO configuration was being initialised and manually switch on and off by setting or clearing the value of the respective pins as shown in the code of Section 4.4.

6.5 7 segment countdown and OLED clashes in TIMER1_INT

Commented [ZGTJL13]: Dun understand this problem at all LOL

Commented [XY14R13]: Its ok just give some generic explanation I guess
LOL

Initially when we tried to set the 7 segment in the 1 second timer function, there were random glitches on the OLED. Since the timer is an interrupt, doing things that require relatively longer periods of time inside of it might affect the planned scheduling of certain tasks or functions. This happens since an interrupt pre-empts all other polling tasks that are currently running and this might cause framing issues for the peripherals that are currently controlled by polling.

7. Feedback

Commented [ZGTJL15]: I lazy write -_-
Help me fill in LOL

Commented [XY16R15]: I also lazy sia bro they should just remove this mod legit

7.1. There could be additional lab time allocated every week outside of our labs so that students can clarify their doubts with the TAs. Right now, it is very difficult for students to seek help outside of the weekly lab times if they face any issues or have any queries. The given lab times every week is barely sufficient for students to absorb the overwhelming amount of information since we are all pretty new when it comes to embedded systems programming.

7.2. It will be good to at least provide a sample skeleton structure for the students to work on for the assignment. Starting out on the assignment was really tough since the demo program had all the functionalities all over the place and it was really messy and confusing for a lot of people, especially when for the code sections that are not explained during the lab.

8. Conclusion

Through Assignment 2, we learnt how to program the BaseBoard using a microprocessor LPC1769, a very popular microprocessor. More specifically, we have learned how to apply system design approaches, to design embedded applications, understand the interfaces between microcontrollers and peripherals, and learn how to do C embedded programming controller-based applications. This enables us to take these skills learned into the working world where we could potentially program more hardware using similar types of microprocessors (belonging to the LPCXpresso family).

Our NUSpace prototype has been shown to be working very well (despite small lags in clearing the warnings due to the limitations of SW4) which would allow us to take our prototype to the next stage and perhaps program a mini-rocket that would really launch into space, using more peripherals or add on functionality to existing peripherals to conduct measurements for any potential space missions in the future.