

**EE2024 ASSIGNMENT 2 LAB REPORT**

NUSpace

|  |  |
| --- | --- |
| Archana Pradeep | A0162694W |
| Khor Ru Shan | A0158339U |

Lab: Tuesday Group 2

**Table of Contents**

1. Introduction and Objectives 3
2. Flowcharts describing the system design and processes 4
3. Detailed implementation 8
4. Application logic enhancements 14
5. Significant problems encountered and solutions proposed 16
6. Conclusion 17

**1. Introduction and Objectives**

In this assignment, we implemented a manned space flight system known as NUSpace. The objective this system is to orbit in space and send data to NUSCloud to ensure the smooth functioning of the system. It sends timely data on its fuel tank, obstacle distance and orientation of the system, with the use of temperature sensor, light sensor and accelerometer respectively.

NUSpace operates in 3 modes:

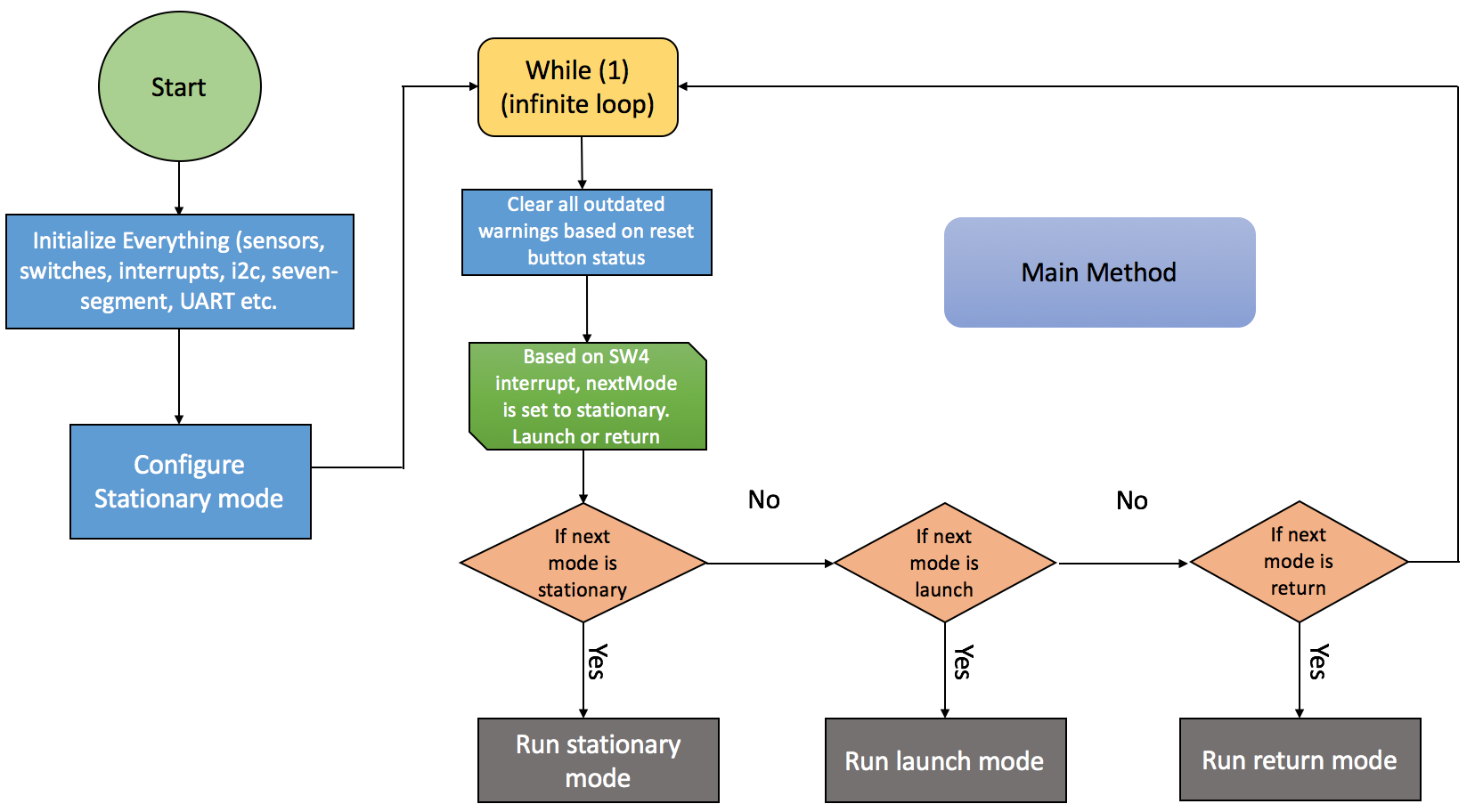
**Stationary Mode:** This is the default mode of the rocket when it is in the launch pad. The OLED display shows “STATIONARY”. Only the fuel tank is monitored in this mode using the temperature sensor and one click on SW4 allows the system to go into launch mode after a countdown sequence given that it is safe to transition to the next mode.

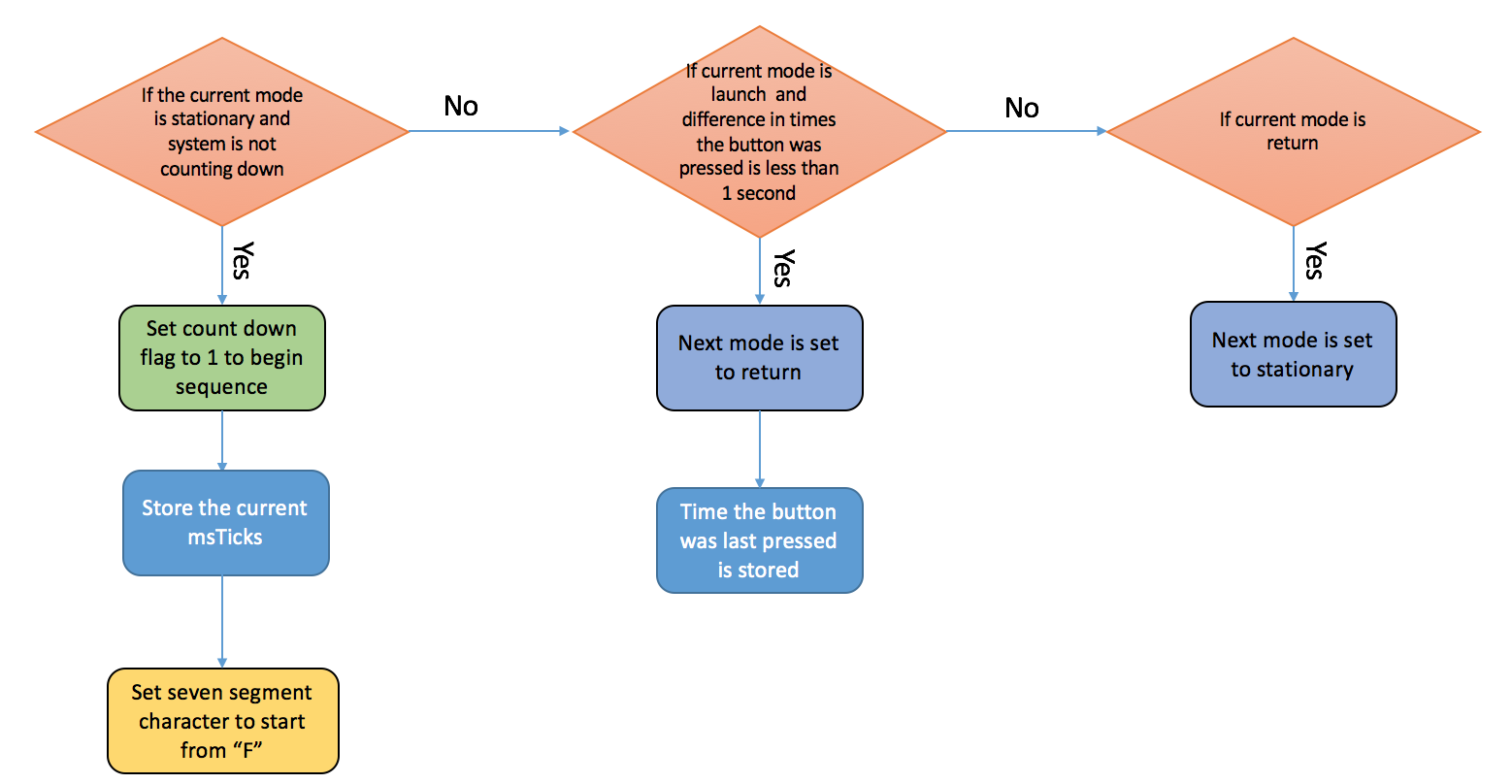
**Launch Mode:** This is the mode in which the rocket is launched towards space. The OLED display shows “LAUNCH”. Both the fuel tank and rocket orientation is monitored. 2 clicks on SW4 helps transition into return mode. The system sends data periodically on the temperature and accelerometer readings to NUSCloud.

**Return Mode:** This is the mode in which the space shuttle mounted on the rocket is detached and returns to land on earth. The OLED display shows “RETURN”. Only the obstacle detection is observed with the help of the light sensor. And the system sends periodic updates on obstacle detection readings to NUSCloud.

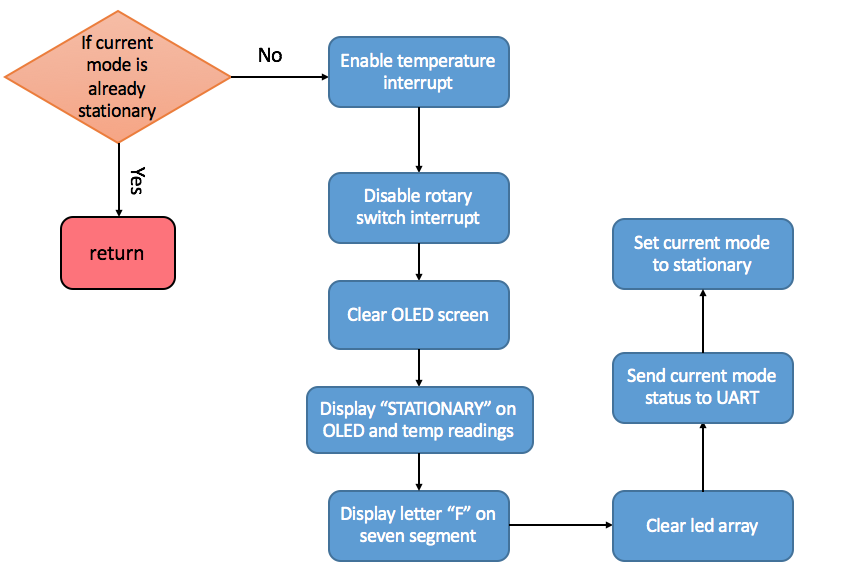
**2. Flowcharts describing the system design and processes**

**2.1 Main Function**

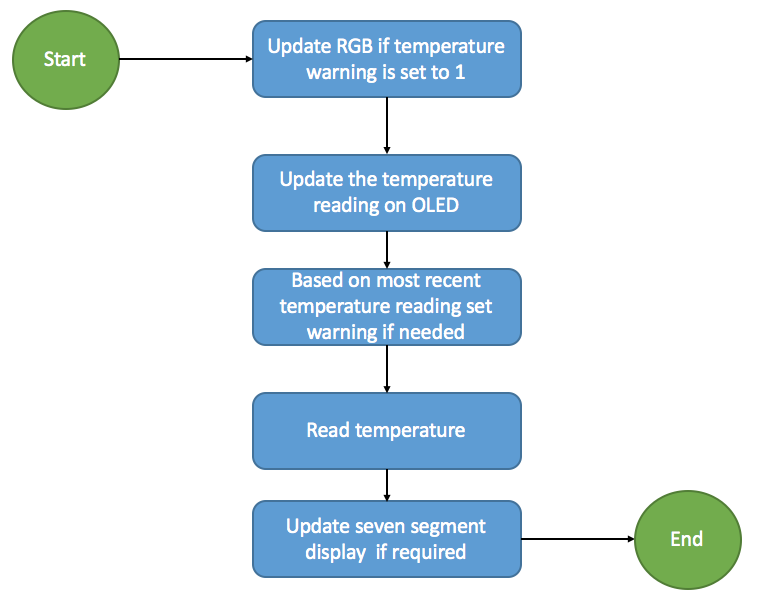
**2.2 EINT0 Handler**



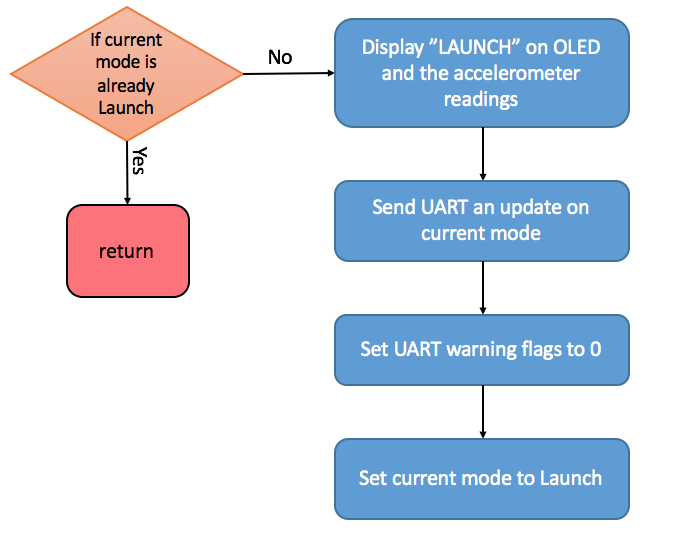
**2.3 Configure Stationary Mode**

****

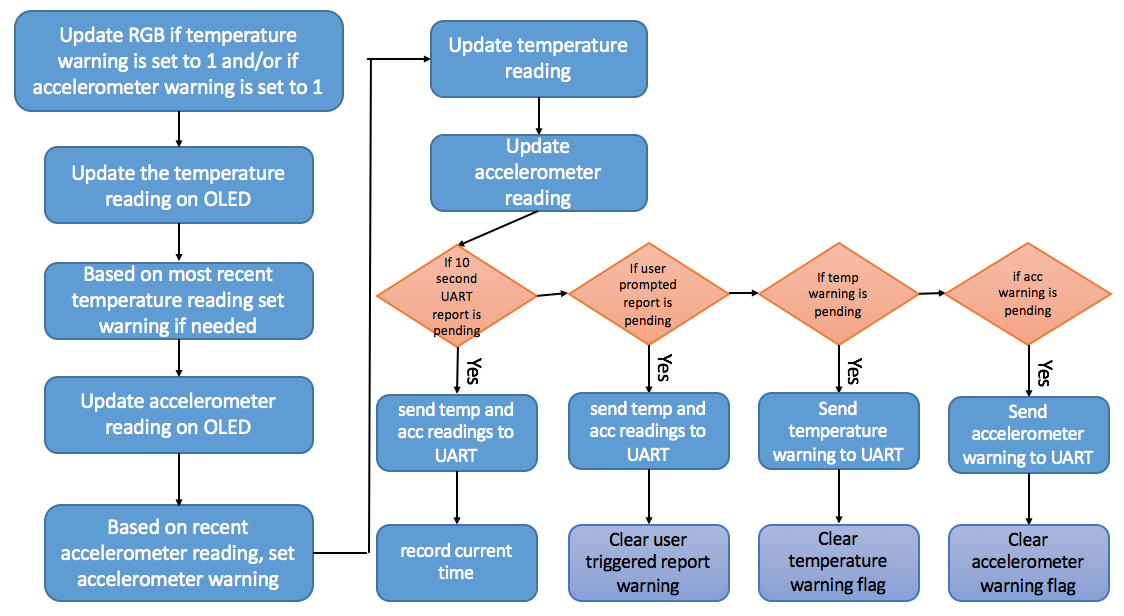
**2.4 Run Stationary Mode**

****

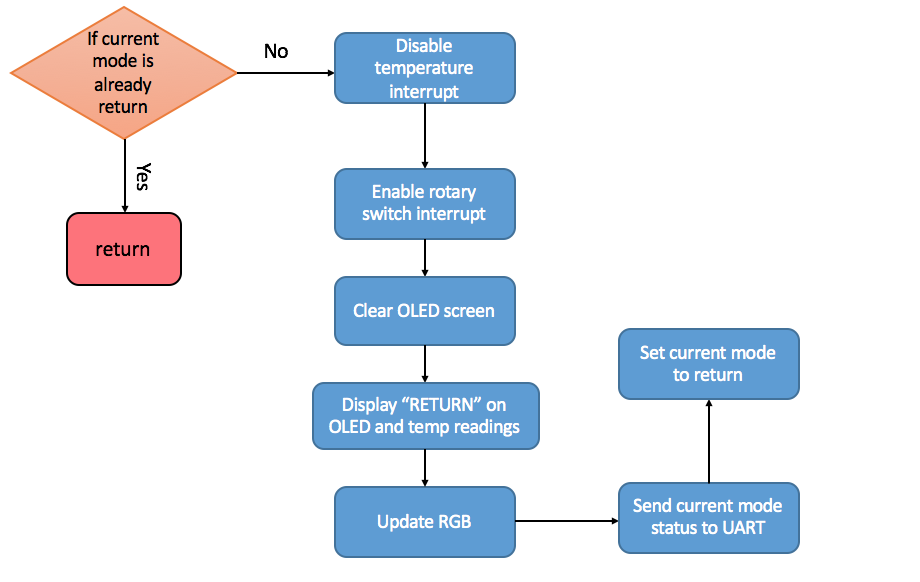
**2.5 Configure Launch Mode**

****

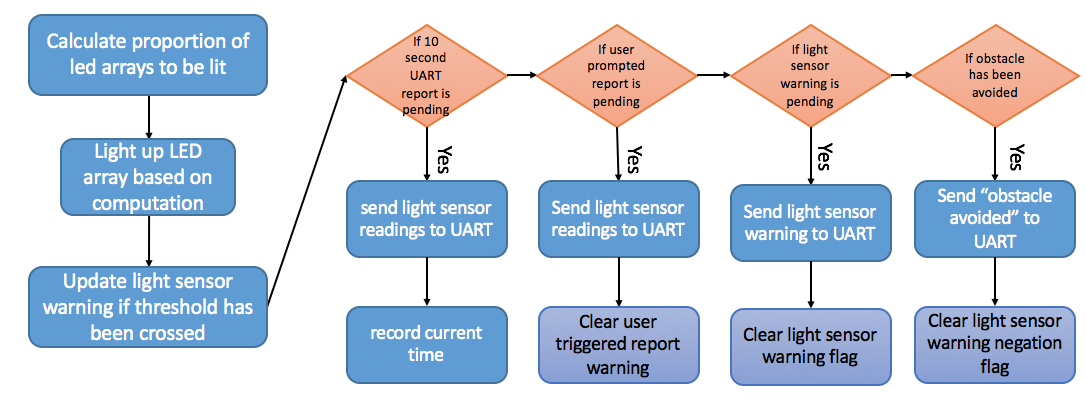
**2.6 Run Launch Mode**

****

**2.7 Configure Return Mode**

****

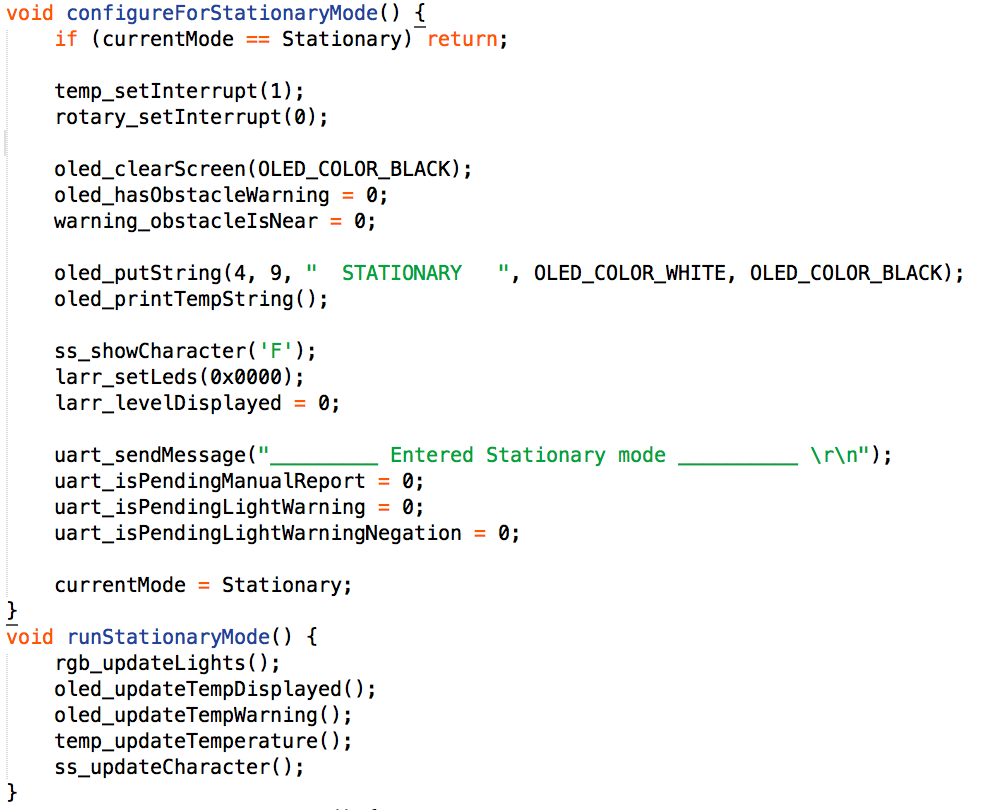
**2.8 Run Return Mode**

****

**3. Detailed Implementation**

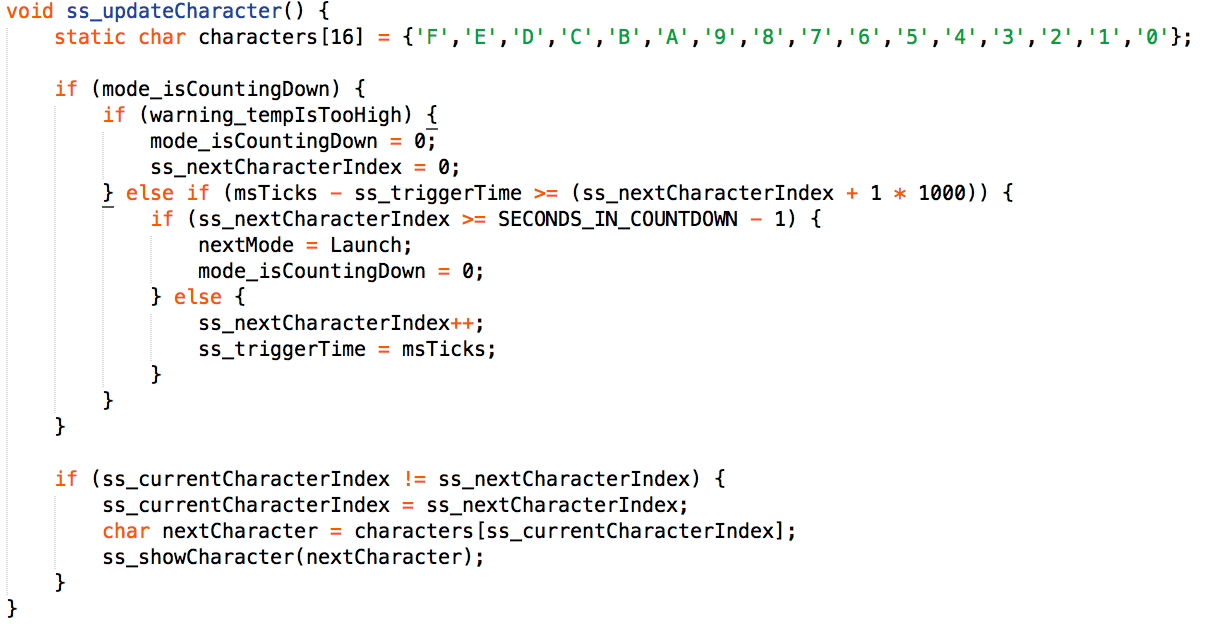
**3.1 Stationary Mode**

Stationary mode is the default mode of NUSpace when the rocket is in the Launchpad. In this mode only the temperature sensor readings are read using the EINT3 (GPIO) interrupt. Moreover, the OLED displays “STATIONARY” and the temperature reading.



As displayed in the code above, we chose to use two methods to run every mode. One is to setup the mode (configureForStationaryMode() ) and one is to run (runStationaryMode() ). In the setup function, we enable the temperature sensor interrupt and disable the rotary interrupt as it will only be available in launch mode. We clear the LED Arrays and the OLED. The OLED then displays “STATIONARY” and a message is sent to the terra term indicating the mode. All UART pending warnings are set to 0. The current mode of the system is set to stationary. We decided to define an enumeration called Mode to define the three modes, stationary, launch and return.

The run function checks to see if the RGB needs to blink. It is supposed to blink red if there is a temperature warning set by the variable warning\_tempIsTooHigh.



The next method call in the run function is to update the OLED display as new temperature readings come in. And if the temperature warning flag is set, the OLED also displays a temperature warning. If the stationary mode is set to go into launch mode, before and during the countdown sequence, if the temperature warning is set, the seven segment is updated to go back to the letter “F”.

The ss\_updateCharacter() function checks if SW4 was pressed within one second and if there is not temperature warning, the mode\_isCountingDown flag is set to 1 indicating that the countdown has begun. However, if during countdown warning\_tempIsTooHigh is set to 1, then the mode\_isCountingDown flag is set to 0 and the countdown is aborted. Else if the countdown is over (computed by checking the difference between msTicks and the time the seven segment was first triggered is greater than or equal to the index of the characters \*1000, as if they are equal that means the countdown is over). Then the nextMode variable is set to launch and the mode\_isCountingDown flag is set to 0. The last condition takes care of the actual countdown. In the case where there is no temperature warning to disrupt the countdown and the countdown isn’t over, then the seven segment character array’s index is incremented and the time is stored in the variable ss\_triggerTime. The last four lines take care of the actual display of the letter on the seven segment display screen.

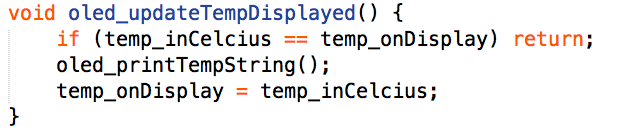
**3.2 Launch Mode**

Similar to stationary mode, the launch mode has a setup function (configureForLaunchMode()) and a run function (runLaunchMode()).

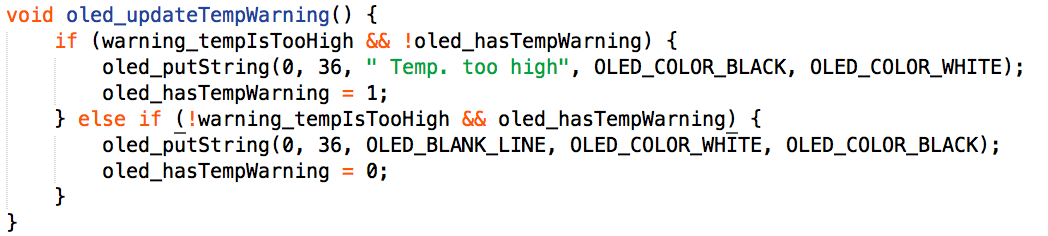


In configureForLaunchmode(), the function checks for whether the currentMode variable is already set to launch. If it has been set to launch, then the main program calls the runLaunchMode() function. The setup function for configureLaunchMode() calls oled\_putString() to display LAUNCH on oled. The oled\_printAccString() prints out the x and y values of the accelerometer on to the oled. The function also sends a message to NUSCloud to report that the system has entered LAUNCH mode. In launch mode, a report on the temperature and accelerometer readings are sent to NUSCloud every 10 seconds and the user may trigger this report by typing “RPT” onto terra term. The flag uart\_isPendingManualReport is flag that indicates whether the user has triggered a report. And the uart\_isPendingTempWarning and uart\_isPendingAccWarning are flags that indicate whether a warning needs to reported to NUSCloud. All these flags are set to 0 in the configure stage. The currentMode is set to launch.

In the runLaunchMode() , similar to the runStationaryMode(), the function rgb\_updateLights() is called. This function checks to see if the temperature warning flag is set and/or accelerometer warning flag is set. If both are set to 1, then the RGB blinks red and blue alternatingly with a period of 333s. Else if only temperature warning is set to 1 then only the red blinks every 333s. If only the accelerometer warning is set to 1 then the blue LED blinks for 333s.

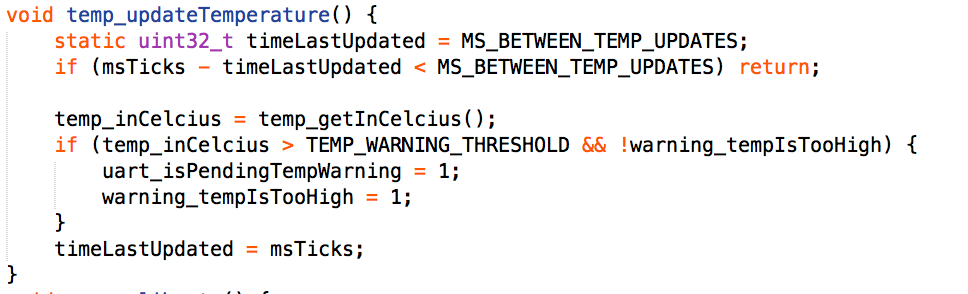


The oled\_updateTempDisplayed() , which changes the temperature displayed on the oled if the current temperature reading is different from the one that is currently on display.

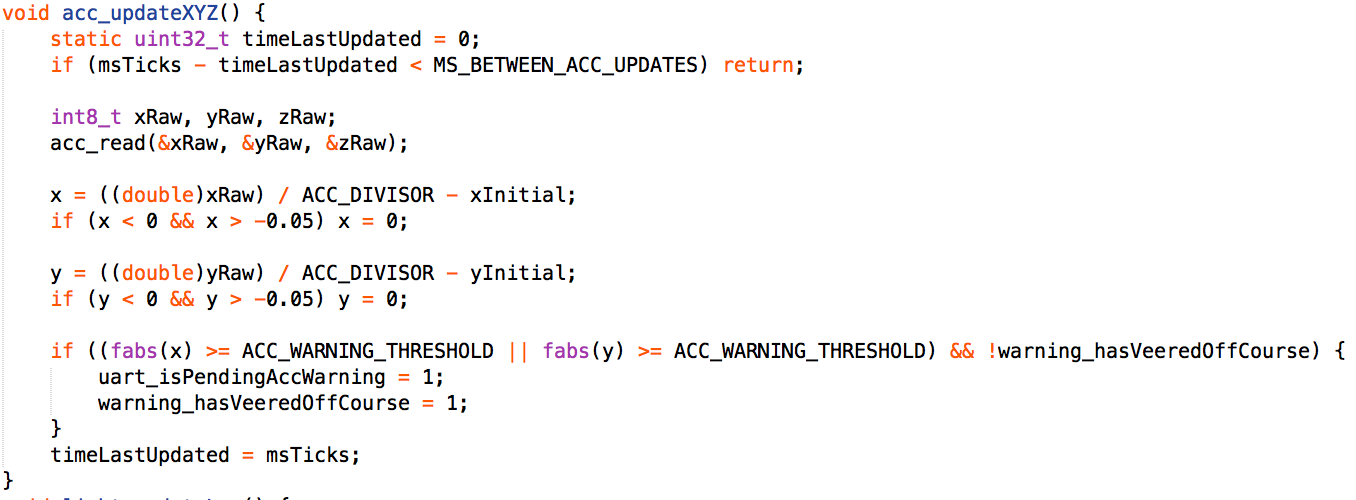


The oled\_updateTempWarning() function checks to see if there is a temperature warning that hasn’t been displayed on the OLED. Thus if the warning is set to 1 but the oled\_hasTempWarning is set to 0 then the oled diplays “Temp. too high”. Else in case the reset button is pressed anf the warning\_tempIsTooHigh is set to 1 but the oled displays a temp warning, then this warning message is replaced with a blank line. The same process takes place for oled\_updateAccDisplayed() and oled\_updateAccWarning().

The last two function calls made in the run method are the temp\_updateTemperature() and acc\_updateXYZ(). The codes with explanations are shown below:



The function above updates the temperature if the MS\_BETWEEN\_TEMP\_UPDATED is greater than the current time and the last time the temp was updated, then it proceeds to retrieve a temperature reading again. If this temperature reading is higher than threshold but there hasn’t been a temperature warning flag set previously, then the uart\_isPendingTempWarning flag and the warning\_tempIsTooHigh flag is set to 1 and the current time is stored in the variable timeLastUpdated for the next round of calls to this function. The reason we chose MS\_BETWEEN\_TEMP\_UPDATED to be 400 ms is because after a lot of trial and error, we realised a smaller time frame results in fast fluctuations of the temperature readings, making it instable to perform other functions on (such as warnings), and 400 ms was just stable enough to not fluctuate too much and give the system to act on the temperature readings.



The acc\_updateXYZ() function is used to update the accelerometer readings if the MS\_BETWEEN\_ACC\_UPDATES is greater than the difference between current time ans the last time the reading was updated which is supposed to be 100ms. It operates the same way as the temp\_updateTemperature() given on the previous page.

The last section of the runLaunchMode() function decides how messages are sent to the uart, the application logic is given by the flowchart 2.6.

**3.3 Return Mode**

the return mode also has a setup function (configureForReturnMode()) and a run function (runReturnMode()).

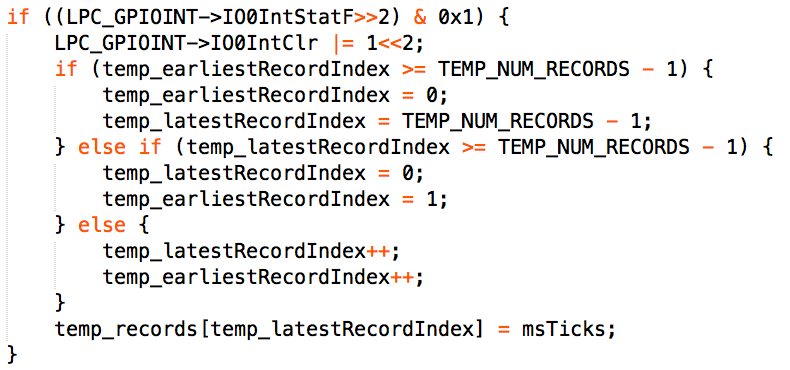


The configureForReturnMode() disables temperature interrupt as only light sensor is read in launch mode. Our extra feature which is the rotary switch is enabled. The oled screen is cleared and the temperature and accelerometer warning flags are effectively disabled by setting them to 0 in this mode. The oled\_updateTempWarning() and oled\_updateAccWarning() will not show any warnings on the oled as its warning flags have been set to 0. The oled then displays “RETURN” and a message reporting the current mode of the system is sent to NUSCloud. The current mode is set to return.

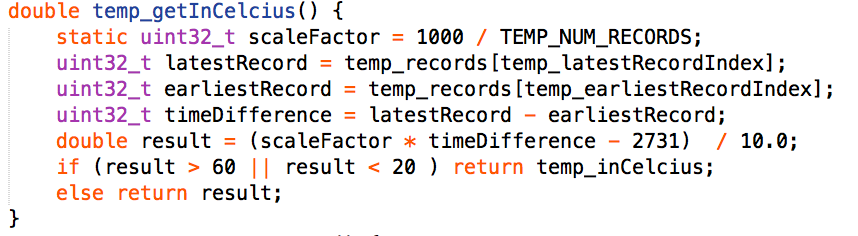
The runReturnMode() function calls light\_updateToBeLit() which computes the proportion of the LED arrays to be turned on for the current light sensor reading. The larr\_updateLeds() function turns on the number of led lights on the two columns red and green. If the light sensor’s readings cross the threshold (3000 lux) then the warning\_obstacleNear flag is set to 1 and the oled displays an “Obstacle near” warning which is later turned off in the block of code that deals with the uart. This part of the code is similar to that of the runLaunchMode() and is explained by the flowchart in section 2.8.

**4. Application Logic Enhancements**

**4.1 Reading temperature using interrupts**



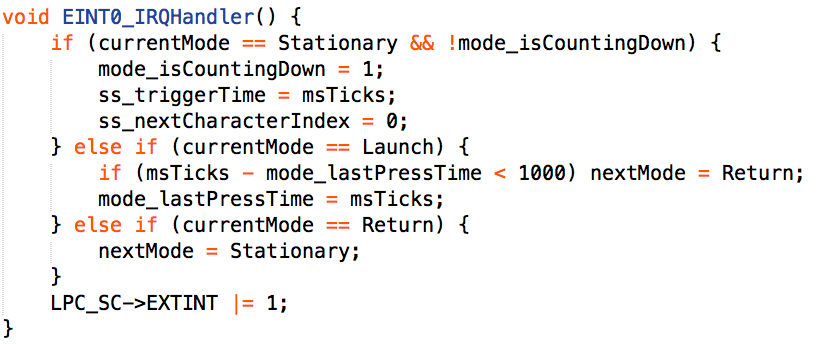
The temperature is read using an EINT3 interrupt. It is implemented using an array which acts as a buffer. The size of the array is 1000 as given by the constant TEMP\_NUM\_RECORDS. The first if condition checks if the Index of the earliest record has crossed the size of the array, then the temp\_earliestRecordIndex is set to 0. And the temp\_latestRecordIndex is set to 999(1000 - 1). So the buffer begins again. Else if the pointer to the latest index has crossed the size of the array. Then the latest record index points to the first element of the buffer and the earliest record index points to the second element in the buffer. In the normal case, both the earliest record index which is set to point to the first element and latest record index which is set to point to the second element of the buffer are both incremented by 1 and finally regardless of which of the if-else conditions are met, the current msTicks is stored into the latestRecordIndex of the buffer. This is later used inside the temp\_getInCelcius() function which computes the temperature from the difference between the time of the earliest record index and the latest record index and does the following computation as shown below:



The scale factor is currently 1 in this case but it is kept there to increase or decrease granularity of the temperature in the future if we need to. The reason we don’t use temp\_read() as we did initially for the polling method is because has a while loop which causes polling for about half a second, thus delaying all the other devices on the baseboard by the same time. In the interrupt driven method we take the average PWM across 1000 periods and use the formula shown above. Thus eliminating any lag caused by the temp\_read() function.

**4.2 Use of EINT0 for SW3 instead of EINT3 (GPIO)**

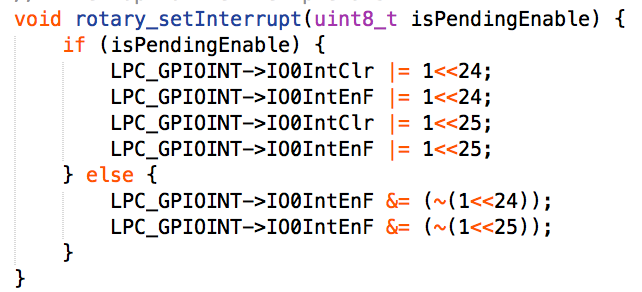
Since using an external interrupt is best for asynchronous interrupt while GPIO doesn’t work for asynchronous interrupts, we thought using EINT0 might lead to more responsiveness of switching modes when the button is pressed.



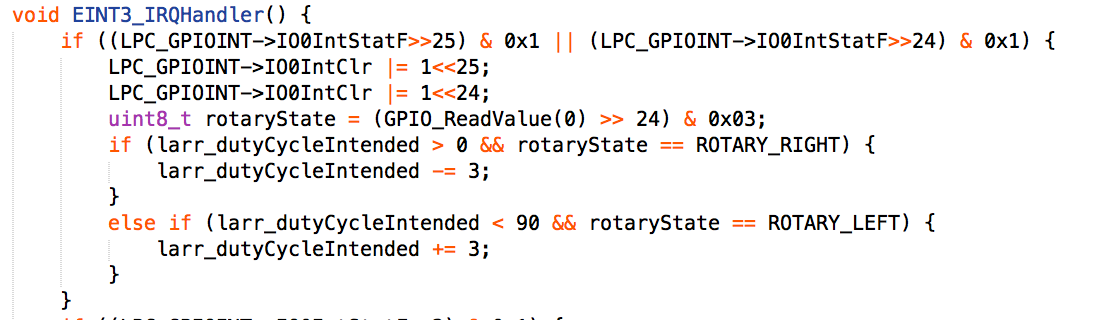
The EINT0 Handler deals with the switching of modes. If the current mode is stationary and the countdown sequence hasn’t been initiated as indicated by the mode\_isCountingDown flag, then the system prepares to go into launch by setting mode\_isCountingDownFlag to 1 and thus initiating the countdown sequence which is later taken care of by the ss\_updateCharacter() function. If the current mode is launch and the button was pressed twice within a second then the next mode is set to return. And finally if the current mode is return, then the next mode is stationary. The external interrupt is then cleared.

**4.3 Rotary Switch in return mode**

The rotary switch is used in return mode to control the brightness of the led arrays on NUSpace. The rotary switch is read using an EINT3 (GPIO) interrupt as shown in the code snippet below.

****

Inside the EINT3 Handler, two pins are checked to read the rotary switch, if the degree of rotation is greater than zero and the switch was rotated to the right then the duty cycle intended is decremented by 3 and if the rotary switch was rotated at less than ninety degrees and was rotated left then the duty cycle intended is incremented by 3. To make sure the ratio PWM duty cycle reacts according to the degree of the rotation.

****

**5. Significant problems encountered and solutions proposed**

The most pressing issue concerns the LED array. While it does respond to the obstacle distances as required, there exists a slight response delay. This is an issue because it adds on the overall delay of the pilot's response to nearby obstacles (overall delay = human response time + system response time). In order to circumvent this safety impediment, we propose making the light sensor reading interrupt-driven, which is a simple matter given that we currently employ robust interrupt-driven handling for the rotary switch, MODE\_TOGGLE button, and temperature sensor.

Indeed, our implementation of the temperature-reading mechanism allowed us to avoid temperature-related safety hazards. We realised that the original polling-driven temp\_read() function provided by Embedded Artists would pause the whole system for a significant amount of time just to poll for a value. This amounted to a safety issue because the temperature and heading of the rocket could change drastically and rapidly while leaving the atmosphere, without it informing the users in a timely manner. Furthermore, this temp\_read() function outputted temperatures in granularities of 0.8 degrees Celsius. In order to increase both the sensitivity and precision, we wrote our own temperature-reading function, and employed an interrupt-driven approach to obtaining our values. At every change in voltage outputted by the temperature sensor, our system would be interrupted. It would record the current timing of the voltage change into a circular buffer. In the main program (the while loop), calculations using the circular buffer were conducted. These are very fast since no interfacing is required outside of the core peripherals. Said calculations would subtract the earliest timing record from the latest timing record, in order to obtain the average temperature over the last 1000 interrupts. This approach gave us a precision of 0.1 degrees Celsius, which we felt was a dramatic improvement over the default temperature granularity. Furthermore, because the processor only needs to interface with core peripherals to conduct the temperature calculations, the entire system is much more responsive.

**6.Conclusion**

In conclusion, we have successfully implemented the NUSpace. We also enhanced the system with a goal of making a faster and more responsive system using interrupt driven peripherals and making our own read function for reading the temperature.