EE2024 Assignment 2 Report
AY17/18 Semester 2

WEDNESDAY LAB GROUP 34

Aaron Soh Yu Han
Teo Meng Shin, Ryan

Table of Contents

1. Introduction and Objectives
2. Flowcharts
3. Detailed Implementation
4. Application Enhancements
5. Significant Problems Encountered and Solutions Proposed
6. Issues or Suggestions
7. Conclusion

Introduction

The aim of this project is to simulate a rocket monitoring system. It features 3 distinct modes to cater to different modes of operation for the rocket: Stationary, Launch and Return.
Below are the list of peripherals used and their roles

1. OLED Display — Display Monitor for the pilots to refer for status updates.
2. 7 Segment Display — Countdown Timer Display.
3. Temperature Sensor — Fuel Tank Monitoring system.
4. Red LED — Fuel Tank Warning.
5. Accelerometer — Detect Orientation of Rocket and Monitor Stability.
6. Blue LED — Orientation / Stability Warning.
7. Light Sensor — Radar System.
8. LED Array — Obstacle Distance Indicator.
9. SW_3 — MODE_TOGGLE Button for pilots to toggle between Modes of Operation.
10. SW_4 — CLEAR_WARNING Button for pilots to clear warnings.
11. Xbee — Wireless Transmitter to send/receive data/instructions from NUSCloud.

Below are the list of key objectives to be met for the system in their respective modes:

Stationary:

1. Warn pilots of unsafe temperatures in fuel tank. Triggers a blinking Red LED with a period of 666ms.
2. Prevent and/or abort countdown sequence if unsafe temperatures are detected in fuel tank.
3. Begin countdown within 1 second of MODE_TOGGLE being pressed, assuming MODE_TOGGLE is **only pressed once within a second and there is no temperature warning.**

Launch:

1. Warn pilots of unsafe temperatures in fuel tank. Send a message to NUSCloud within 1 second of detecting the unsafe temperature. Triggers a blinking Red LED with a period of 666ms.
2. Warn pilots of unsafe orientation or instability of rocket. Send a message to NUSCloud within 1 second of detecting the unsafe orientation or instability. Triggers a blinking Blue LED with a period of 666ms.
3. If both warnings are triggered, both Red and Blue LEDs will blink alternately at a period of 666ms.
4. Enter Return Mode within 1 second of MODE_TOGGLE being pressed twice, assuming MODE_TOGGLE is **pressed exactly twice within a second**. Mode change will occur regardless of warning(s) present.

Return:

1. Warn pilots of approaching obstacles. Send a message to NUSCloud within 1 second of detecting the obstacle.
2. LEDs in the array will light up proportional to the distance between the Rocket and an object. Closer distance = More LEDs.

3. Notify pilots of safe avoidance of approaching obstacles. Send a message to NUSCloud within 1 second of avoiding the obstacle.
4. Enter Stationary Mode within 1 second of MODE_TOGGLE being pressed once, assuming MODE_TOGGLE is **only pressed once within a second**. Mode change will occur regardless of warning(s) present.

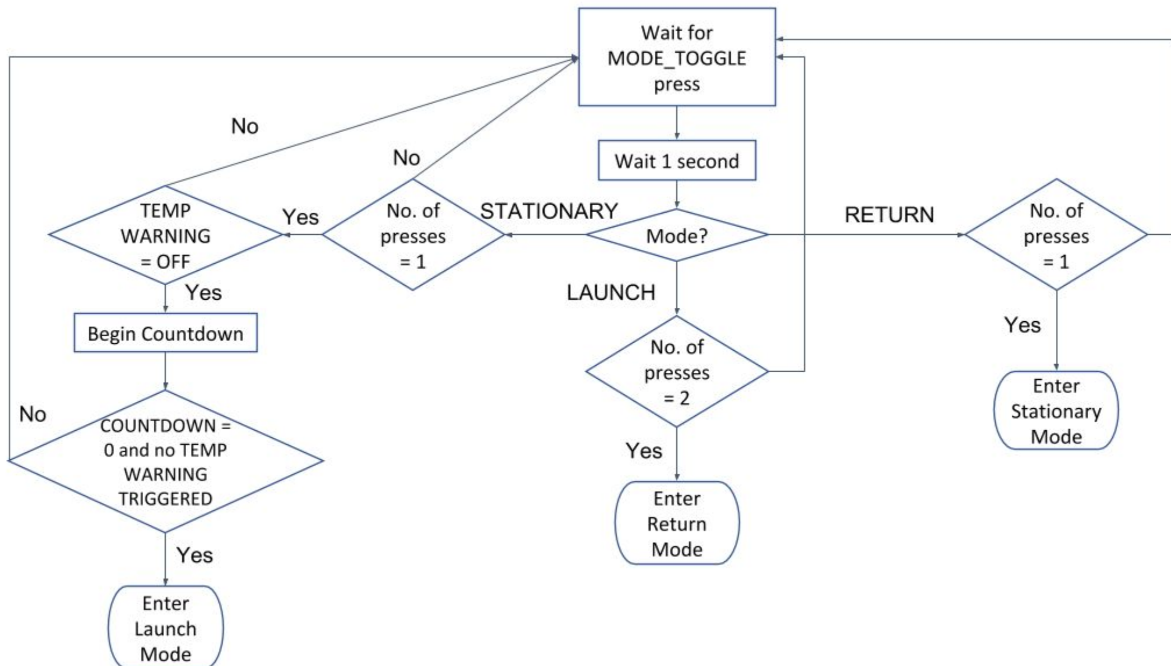
Launch and Return modes:

1. Send status updates to NUSCloud every 10 seconds.
2. Send status update to NUSCloud within 1 second of receiving a report request from HQ.

Flowcharts / Detailed Implementation

We have decided to use FreeRTOS to help schedule tasks accordingly so as to meet the real-time constraints of this system.

Implementation of MODE_TOGGLE



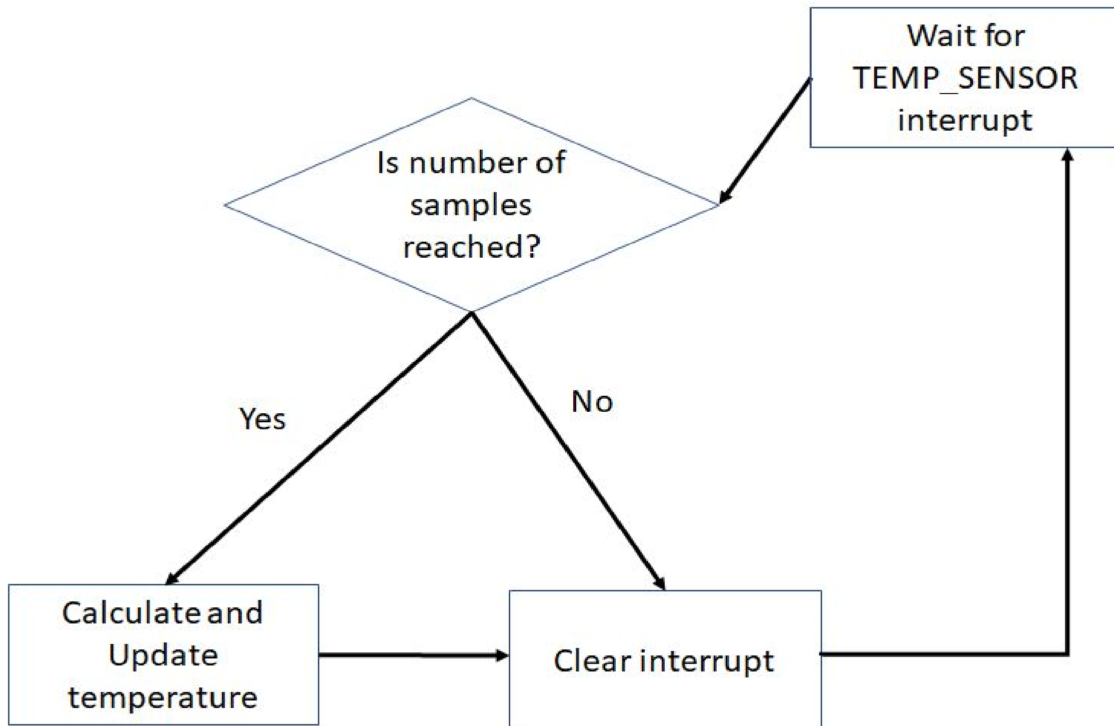
MODE_TOGGLE uses SW_3. SW_3 can be configured to interrupt via EINT3 or EINT0. We have chosen to route the SW_3 interrupt to EINT0. This is to reduce the Worst Case Execution Time (WCET) of EINT3_Handler which most other GPIO peripherals will be routed to. This was achieved as follows:

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

We also ensured that a mode change will only occur if the exact number of presses is done within a second. This is done by waiting for 1 second and checking if any additional presses were made. A code snippet for this used during LAUNCH mode is shown below:

```
last_press_count = MODE_TOGGLE_PRESSES;
xLastWakeTime = xTaskGetTickCount();
vTaskDelayUntil(&xLastWakeTime, xFrequency); //xFrequency is 1000
curr_press_count = MODE_TOGGLE_PRESSES;
count = curr_press_count - last_press_count;
if (count == 1) { //Set to detect 1 press because the first press would unblock this task.
        vTaskResume(enter_return_handle);
}
```

Reading Temperature

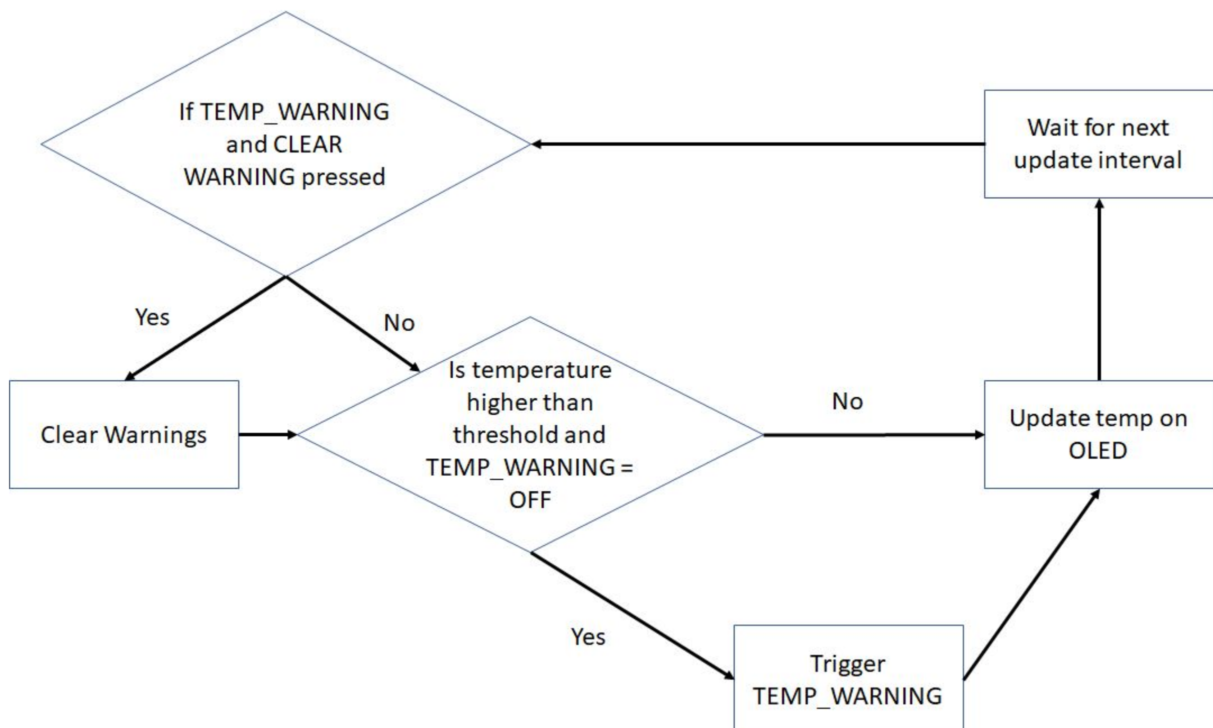


We have chosen to use interrupts in order to read the temperature. Due to the blocking nature of the built-in temp_read() function, it takes about 300 ticks (based on our measurement) for the function to complete. This is because the temperature sensor returns the current temperature by the “width” of each pulse, thus the function needs to obtain an average reading and process it accordingly to get a reliable reading. This would impact the real-time nature of our system. We have set up the temperature sensor interrupt to trigger whenever there is a rising or falling edge (state change). This is achieved with:

LPC_GPIOINT ->IO0IntEnF |= *1 << 2; LPC_GPIOINT ->IO0IntEnR* |= *1 << 2;*

Once the required number of state changes have been met, we take the time taken for the sensor to complete the required number of state changes and apply a formula [similar to the one found in temp_read()] to obtain the temperature. The temperature is then updated in the system.

Temperature Warning

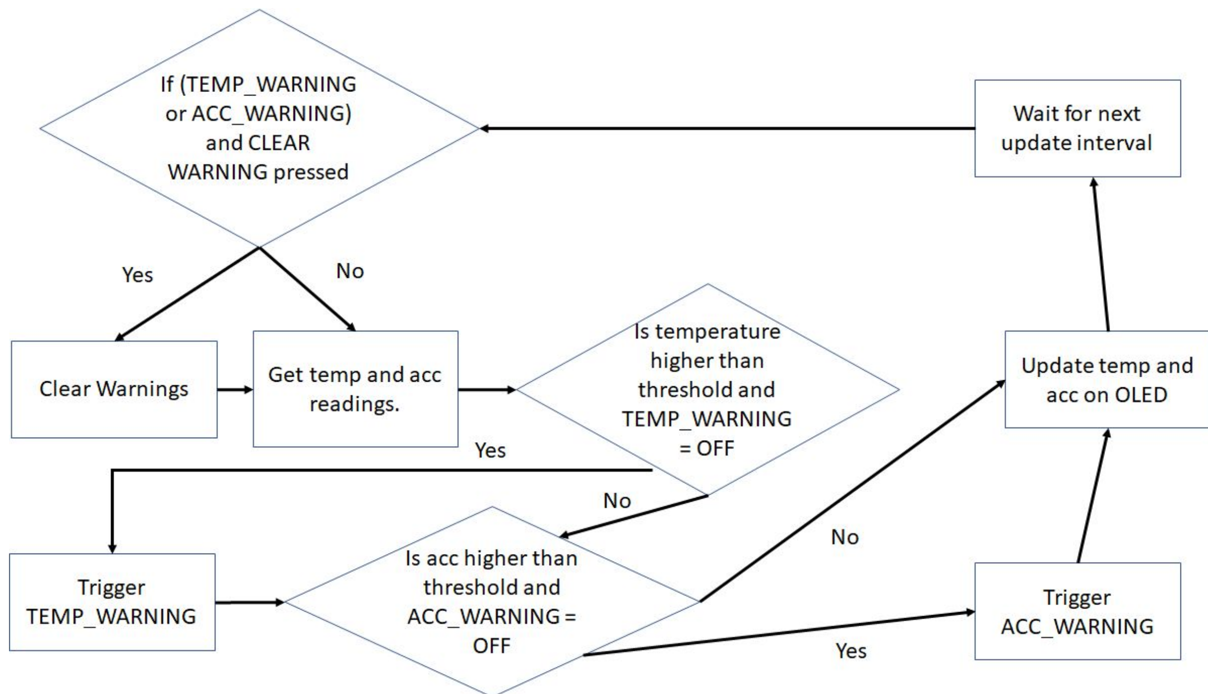


The checking of temperature is done in a periodic task. The task is frequent enough to provide a “near instantaneous response” which cannot be perceived by humans. This task checks whether the current temperature in the system (refer to READING TEMPERATURE) is greater than TEMP_HIGH_THRESHOLD. If it is, the system will trigger a temperature warning state, sending “Temp. too high” to NUSCloud **(This message is not sent if the system is in STATIONARY mode)** and blink the red LED.

Updating of OLED for Temperature and Accelerometer values

The updating of OLED display is typically done in a periodic task. The task is frequent enough to provide a “near instantaneous response” which cannot be perceived by humans.

Reading Accelerometer and Accelerometer Warning

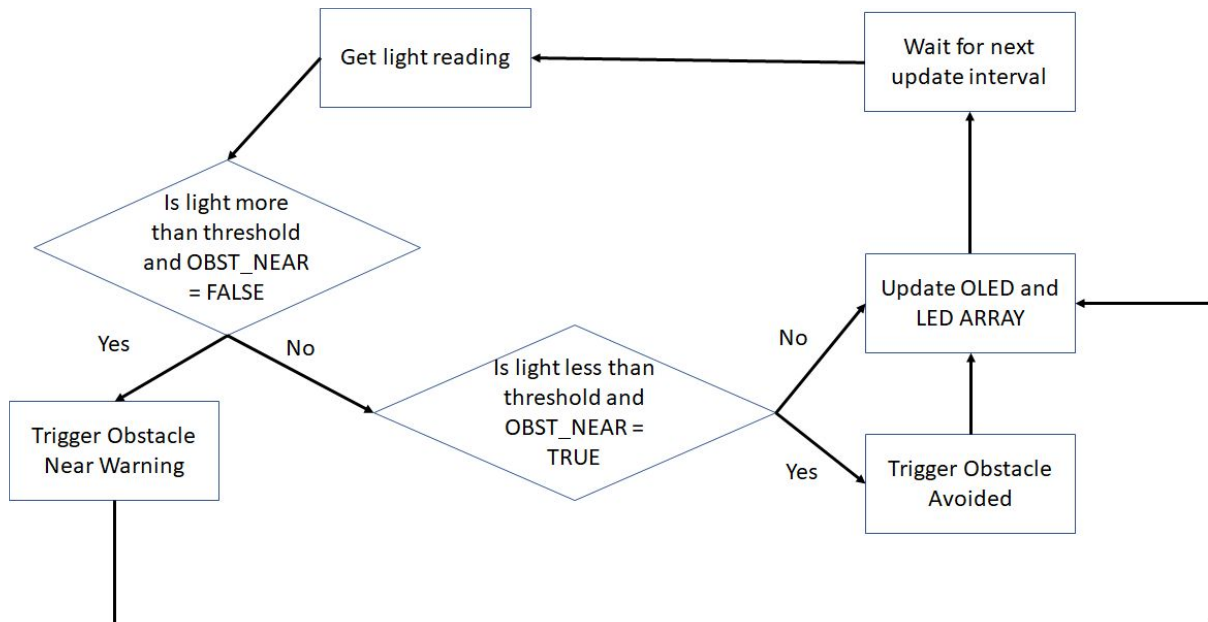


The accelerometer is configured with the Z-axis disabled. The reading and updating of accelerometer values are done in a periodic task. The task is frequent enough to provide a "near instantaneous response" which cannot be perceived by humans. As the range of our accelerometer is 4 Gs represented by an 8-bit value, we divide the reading by 64.0 to convert it to an accurate representation of G. The check for accelerometer thresholds are done immediately after the reading and updating of accelerometer values. If the X or Y axis has a reading larger than ACC_THRESHOLD, the system will trigger an accelerometer warning state, sending "Veer off course" to NUSCloud immediately and blinking the blue LED.

Implementation of CLEAR_WARNING

As SW_4 is connected to Port 1 of LPC1769, interrupts cannot be used to detect a button press. Hence, we are polling the button frequently at fixed intervals to detect the press. There are occasions whereby an extremely quick press and release of SW_4 might not be detected by the system due to the polling cycles. However, we feel that this has a negligible impact on the overall functionality of the system. The purpose of this button is to allow a pilot who is already aware of the respective warnings to deactivate them. This is not critical and time-sensitive, thus a pilot holding down the button for an extremely short time till the next polling cycle is acceptable.

Reading Light Sensor and Light Sensor Warning



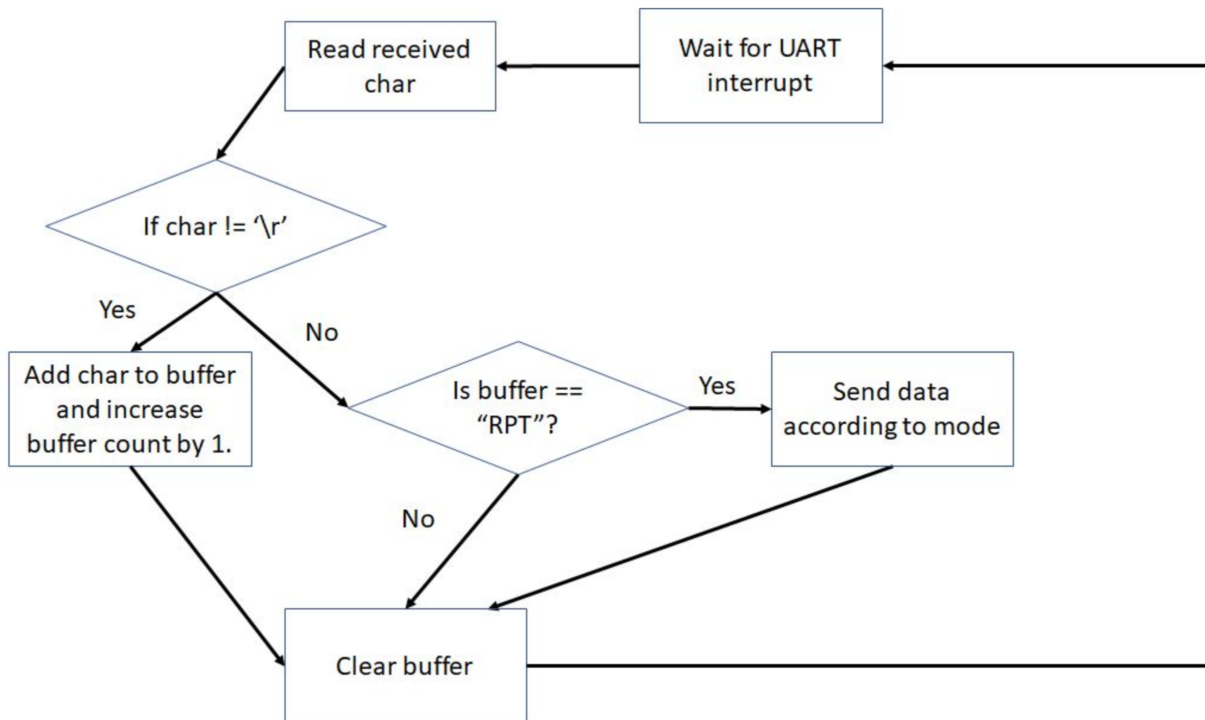
The light sensor has a periodic task that reads the value and checks if it exceeds the threshold. The task is frequent enough to provide a “near instantaneous response” which cannot be perceived by humans. We set the range of the light sensor to read up to 4000 and the default OBSTACLE_NEAR_THRESHOLD to be 3000 (in accordance with the guidelines). When the value read by light sensor is greater than 3000, OBST_WARNING is switched to ON and the message “Obstacle near” is sent to NUSCloud. When the light sensor reading drops back to below 3000, OBST_WARNING is switched to OFF and the message “Obstacle avoided” will be sent to NUSCloud immediately.

The LED array corresponds to the reading on the light sensor, such that for every increase of 200 lux, an additional LED will light up. This is done by bit shifting to the left by 1 whenever light sensor readings increase by 200.

UART transmission to NUSCloud

Whenever the system enters a mode, the message ‘Entering xxxx mode’ is sent to NUSCloud. If the system is in LAUNCH mode, the temperature and accelerometer readings will be sent to NUSCloud every 10 seconds. Similarly in RETURN mode, the light sensor reading will be sent to NUSCloud every 10 seconds. This is handled by a periodic task ‘send_NUSCloud’, which uses vTaskDelayUntil() to achieve a fixed period of 10 seconds.

UART interrupt



We set the interrupt to trigger at every character. This is achieved by:

```
UART_FIFOConfigStructInit(&uartFIFOCfg);
uartFIFOCfg.FIFO_Level = UART_FIFO_TRGLEV0; // set to trigger interrupt at 1 character
UART_FIFOConfig(LPC_UART3, &uartFIFOCfg);
```

Every time a character is received, it gets placed in a buffer. The buffer is set to cycle indefinitely to prevent an overflow. The relevant code snippets are shown below:

```
UART_Receive(LPC_UART3, &data, 1, BLOCKING);
if (data != '\r') {
        receivedCommand[len] = data;
        len++;
}
<<Additional code in between>>
if (len == 3) len = 0; //Refresh Command Scanning
```

When a "\r" character is received, the system will compare the characters in the buffer to "RPT". If it matches, a status report related to the current mode is sent immediately and buffer cleared. If there is no match, the buffer will still be cleared. This is to emulate a "clear" command whereby someone on the terminal might have entered wrong input and would like to start afresh.

Application Enhancement

We have chosen to enhance the system by making use of the Joystick to adjust the warning thresholds for temperature, accelerometer and light sensor while the system is in operation. We based our enhancement logic about how a rocket would operate in real life. A pilot would want to be able to change the thresholds according to the situation without having to redownload the entire rocket software again. This is useful when it is technically impossible to redownload the software, such as when the rocket is in flight or on another planet.

Changing the temperature warning threshold for the fuel tank monitoring will allow the rocket use different types of fuel which may have higher/lower combustion temperature. In addition, it is also to allow the rocket to operate in different environments with higher/lower temperatures.

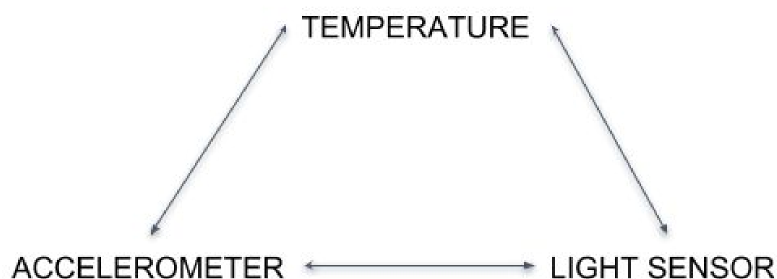
Changing the accelerometer warning threshold will allow the rocket to make sharper manoeuvres, such as avoiding obstacles, or account for different gravitational forces on different planets.

Changing the light warning threshold will allow the pilot to adjust to sensitivity of the radar to be more or less sensitive. For example, if the rocket is travelling fast, it would be better for the radar to more sensitive and detect obstacles further away. If the rocket is travelling slower, the pilot might want to make the radar less sensitive as the he has more time to react to obstacles.

Enhancement Implementation

The Joystick input is read by interrupts to process the input and update the relevant threshold. The updating of OLED display is done in a separate task to keep the ISR short. The last line of the OLED is reserved solely for the purpose of displaying messages related to the modification of thresholds (this enhancement) and is visible in all modes.

By moving the Joystick left or right, the user is able to select each of the 3 different thresholds to view on the OLED display in real time. By default, the Temperature threshold is selected and displayed upon system initialisation. It is implemented in a cyclic fashion to ensure easy navigability. This is illustrated below:



A snippet of code from Joystick right is shown below to show our implementation:

JOYSTICK_MODE = (JOYSTICK_MODE + 1) % 3;

By moving the Joystick up or down, the user will be able to increase or decrease the selected threshold (up to certain limits) and view the change in real time on the OLED display. Some samples of the OLED messages are given below:

Max Temp = 30

Max G = 0.65

Max Dist. = 3200

By pressing Joystick Center, all the thresholds will be reset to their default values. The OLED display will also update the currently selected threshold on display in real time. (The other thresholds are also reset, just not displayed on OLED due to space constraints.)

Problems Encountered & Solutions

Problem 1: We would sometimes encounter problems where OLED would display weird things or peripherals return wrong values. We narrowed it down to the fact that the SSP/I2C transmissions were being preempted by another task.

Solution 1: We protected these operations by disabling preemptions while still ensuring that the system would function with real-time constraints.

Problem 2: Memory Allocation Failure due to limited size of LPC1769 RAM

Solution 2: We allocated stack space for each task more frugally to prevent malloc failures. At the same time, we also had to ensure the the smaller amount of stack space would not result in a stack overflow. Hence, we carefully calculated the stack sizes of each task to find the optimal stack space allocation.

Issues or Suggestions

1. Issue — The baseboards are very old and unreliable. Our first baseboard had a faulty 7 segment display midway through the project. In our second kit, the LPC1769 had a faulty USB port. In our last kit, the wired UART USB port broke off (already reported to lab staff).
2. Issue/Suggestion — Introduce XCTU utility for XBee modules. One of the XBee modules we received had some sort of firmware issue which had to be rectified using XCTU. This was discovered with the help of Professor Rajesh.
3. Issue — We were unable to supply photos of our enhancement on the OLED. This is because the OLED display on our baseboard was so dim that it was barely visible. Even the GA assessing us had an extremely hard time reading what was displayed on the OLED.

Conclusion

Over the course of this project, we have learnt several key skills in electrical engineering. One of them would be the ability to interface with different peripherals. It is important for us to understand how each peripheral works by reading the respective datasheets. This is because each peripheral has their own way to represent data. For example, the accelerometer uses 8-bit values whereas the temperature sensor uses pulse widths. There is no “one size fits all” method to interface with every single peripheral, thus we must be able to adapt to whatever peripheral we are given.

We have also gained an appreciation for the various protocols used over the course of the project, learning the advantages and disadvantages of one over the other. In addition, we experienced first hand the dangers of preempting protocols while they are in the midst of transmitting data. Some protocols such as I2C do not have a defined behaviour if the transmission gets interrupted (to be clear, interruption in this case refers to a higher priority task preempting the current task using the I2C bus, **not** processor interrupts) for a relatively long period while sending an instruction to a slave.

We also recognised the constraints faced when programming for embedded systems. In this case, we were constrained by the limited RAM and had to find ways to work around it.
We could have either allocated more variables to static memory or be more frugal with our stack size allocated to each task. This is a good learning experience as in the future, we might be faced with other constraints such as power or limited I/O interfaces.