**EE2028 Assignment 2: LanderNUS**

|  |  |  |
| --- | --- | --- |
| **Done by** | | |
| Name: Wah Xinyan Kendrik | Matriculation Number: A0164914B | Lab: Wednesday W05 |
| Name: Varun Shaji | Matriculation Number: A0177065Y | Lab: Wednesday W05 |

|  |  |
| --- | --- |
| **Table of contents** | |
| Objectives and Introduction **..........................................................................................................................** | Page 1 |
| Flowchart and Detailed Implementation  **.......................................................................................................** | Page 3 |
| Enhancements **.............................................................................................................................................** | Page 9 |
| Significant Problems & Solutions **.................................................................................................................** | Page 11 |
| Issues or suggestions **..................................................................................................................................** | Page 13 |
| Conclusion **...................................................................................................................................................** | Page 13 |

**Objectives**

To use the LPC1769 microcontroller and the LPCXpresso Baseboard to simulate a moon lander. To do so, various peripherals as shown below will be used, with the use of each peripheral requiring the application of various concepts taught in EE2028 in order to be active in various modes as shown below.

In the table below, **Y** denotes an active peripheral for a given mode and **N** otherwise.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Peripheral** | **Designation** | **ORBITING** | **TOGGLE** | **LANDING** | **EXPLORING** |
| 96x64 White OLED | **GRAPHICS\_DISPLAY** | **Y** | **Y** | **Y** | **Y** |
| RGB LED | **RGB\_LED** | **N** | **Y** | **Y** | **Y** |
| 7-segment LED display | **SEGMENT\_DISPLAY** | **Y** | **Y** | **Y** | **Y** |
| UART terminal | **SatelliteNUS** | **Y** | **N** | **Y** | **Y** |
| SW3 | **MODE\_TOGGLE\_BUTTON** | **Y** | **N** | **N** | **N** |
| MMA7455L | **ACCELEROMETER** | **N** | **N** | **Y** | **N** |
| ISL29003 | **LIGHT\_SENSOR** | **N** | **N** | **Y** | **N** |
| PCA9532 | **LED\_ARRAY** | **N** | **N** | **N** | **Y** |
| MAX6576 | **TEMPERATURE\_SENSOR** | **N** | **N** | **N** | **Y** |
| SW4 | **CHARGING\_BUTTON** | **N** | **N** | **N** | **Y** |

**Introduction**

LanderNUS is a simulation of a moon lander that has three modes of operation: **ORBITING**, **LANDING** and finally, **EXPLORING**. Each mode can be triggered via certain conditions and embedded within each mode lies various other sub-modes like that of **TOGGLE** and that of Sleeping mode. Starting from **ORBITING**, activating **MODE\_TOGGLE\_BUTTON** will result in the transition from **ORBITING** to **TOGGLE** and then to **LANDING** mode. When light intensity of the surroundings of LanderNUS is sufficiently low, LanderNUS immediately enters **EXPLORING** mode, where it will enter Sleeping mode when its power reaches below a certain amount, where pressing **CHARING\_BUTTON** will awaken and continue functioning.

With four working modes, LanderNUS has various peripherals that will work depending on the mode LanderNUS is currently on. The table in the next page will contains details of how each device will work in each particular mode.

|  |  |  |
| --- | --- | --- |
| **Modes** | **Peripherals** | **Action** |
| **ORBITING** | **GRAPHICS\_DISPLAY** | At all times, display “Orbiting Mode. Press SW3 to Land”. |
| **SatelliteNUS** | Send “Start: Orbiting, waiting for Landing” only **ONCE**. |
| Send corresponding value if sensor reading requested. |
| **MODE\_TOGGLE\_BUTTON** | Press **TWICE** in one second to trigger activation of **TOGGLE** mode. |
| **SEGMENT\_DISPLAY** | At all times, display “O”. |
| **TOGGLE** | **GRAPHICS\_DISPLAY** | Display “ENTERING LANDING MODE”. |
| **RGB\_LED** | Only the **BLUE** LED must blink every **500 milliseconds**. |
| **SEGMENT\_DISPLAY** | Start a countdown from 5 to 0, both numbers inclusive. Decrement of 1 corresponds to 1 second, and then enter **LANDING** mode immediately. |
| **SatelliteNUS** | Send “LANDING Mode” only **ONCE**. |
| **LANDING** | **GRAPHICS\_DISPLAY** | Always display “LANDING” and acceleration in x, y and z-axes in g.  Immediately display “Poor Landing Attitude” **tilt angle exceeds 30˚.** |
| **RGB\_LED** | Both **RED** and **BLUE** LED must blink alternately every **500 milliseconds**. |
| **SEGMENT\_DISPLAY** | At all times, display “L”. |
| **SatelliteNUS** | 1. Send “LANDING Mode” to **SatelliteNUS** once and **ACCELEROMETER** and **LIGHT\_SENSOR** readings once every 5 seconds if conditions in 2 are not met.  2. Continuously send warning messages if either **tilt angle exceeds 30˚** **OR** **light intensity is less than 30 lux**. Stop sending if conditions are no longer satisfied. |
| Send corresponding value if sensor reading requested. |
| **ACCELEROMETER** | Read the acceleration in the x, y and z-axes, determine the tilt angle and disrupt when **tilt angle exceeds 30˚**. |
| **LIGHT\_SENSOR** | Continuously read light intensity value of surroundings with a peak intensity of **4000 lux**.  If **LIGHT\_SENSOR interrupt** notifies that the luminous intensity falls **less than 30 lux**, LanderNUS immediately enters **EXPLORING** mode. |
| **EXPLORING** | **GRAPHICS\_DISPLAY** | Always display “EXPLORING” and **TEMPERATURE\_SENSOR** reading to 1 decimal place. If power is below **12.5%**, display “Sleeping” instead. |
| **RGB\_LED** | At all times, on **BLUE** LED only. |
| **SEGMENT\_DISPLAY** | At all times, display “E”. |
| **SatelliteNUS** | Send “EXPLORING Mode” to **SatelliteNUS** once, and then **TEMPERATURE\_SENSOR** reading and remaining battery life once every 5 seconds. Stop sending if battery life is below **12.5%**. |
| Send corresponding value if sensor reading requested and if battery life is above **12.5%**. |
| **LED\_ARRAY** | Updated according to the current battery life. One LED corresponds to **6.25%** of battery life. |
| **TEMPERATURE\_SENSOR** | Read temperature if power is not below **12.5%**. Otherwise, stop reading temperature values. |
| **CHARGING\_BUTTON** | Restore power by **6.25%** of total initial power when pressed. |

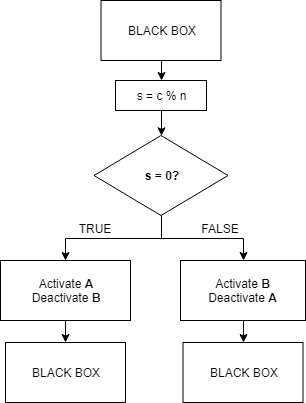
Before proceeding on, the following table details the names given for each unique specification. Not all unique specifications are detailed in the document; some are given for easier classification.

|  |  |  |
| --- | --- | --- |
| **Modes** | **Specification Name** | **Details** |
| **ORBITING** | **BLINK\_BLUE** | Alternate **BLUE** LED between ON and OFF every 500 milliseconds. |
| **MODE\_TOGGLE\_TRIGGER** | To determine whether **MODE\_TOGGLE\_BUTTON** has been pressed twice in 1 second. |
| **LANDING** | **ALTERNATE\_BLINK** | Alternate **BLUE** and **RED** LED between every 500 milliseconds. |
| **TILT\_THRESHOLD** | LanderNUS can tilt at a maximum of **30˚**. |
| **LIGHT\_THRESHOLD** | Luminous intensity must be **more than 30 lux** to remain in **LANDING\_MODE**. |
| **EXPLORING** | **POWER\_THRESHOLD/ ENERGY THRESHOLD** | Power must be above **12.5%** to not enter **Sleeping** mode. |

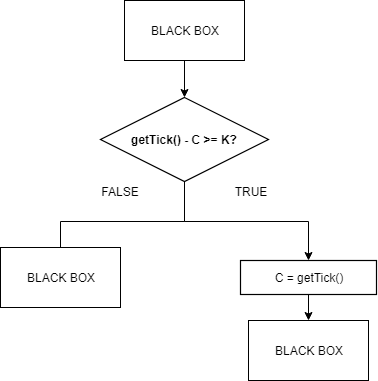
**Flowchart and Detailed Implementations**

Before going into the flowcharts and the specific detailed implementations existing within each mode, the following abstractions exist within at least more than one of these modes, and as such are placed before elaborating on **ORBITING\_MODE** for easier classification:

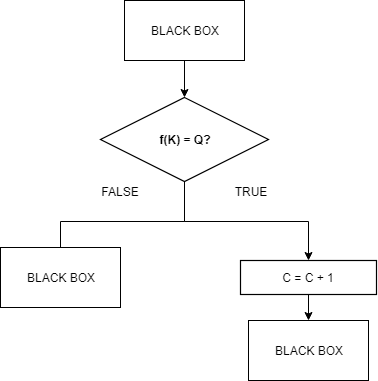
* **ARBITRARY\_BLINK**
  + For a LED with k sub-components being utilized and the requirement of alternate blinking between n states, the equation below holds true for our implementation with a counter of c that initially starts at 0:
  + Then, understand the following inequality is always true: . Thus, at most c-1 states can be accounted for at any given point in time. To avoid stagnation, c will always be incremented by 1 to allow traversal across various states.
  + **For the case of LanderNUS, k = 2, n = 2 at worst,** so s = 0 or 1 but not both and not equal or more than 2. As such,, let s = 0 (**BLUE\_LED** to be toggled **ON**, any state where s is not 0 to be toggled **OFF)** be **A** and s = 1 (**BLUE\_LED** to be toggled **OFF**, the state where s is not 0 to be toggled **ON)** be **B**.
  + The following flowchart describes one pass of **ARBITRARY\_BLINK,** where there will be similarities in the sections of **BLINK\_BLUE** and **ALTERNATE\_BLINK** will be witnessed in **ORBITING\_MODE** and **LANDING\_MODE** respectively:



* **Counter Implementation**
  + As you will observe in the following flowcharts (and not just in **ORBITING\_MODE**), there are two different kind of counters being implemented, both of which have been implemented for the sake of variety. These counters occur very frequently throughout the entire implementation.
  + The **FIRST** kind of counter involves the use of the function **getTick()** and the comparing it with a pre-existing counter and a constant value, usually a multiple of 1 millisecond, Clearly, this counter is time-dependent; to determine the subsequent logic upon reaching that point is wholly dependent on the time difference between at least two values recorded in time. The flowchart below describes the generic representation of the time-dependent counter.



* + The **SECOND** kind of counter is literally using a counter value, instead; **getTick()** is not necessarily used as a means of comparison to increment this counter. This is generally preferred when there are multiple devices running at the same time, or when the design specification demands a specific kind of array traversal to be implemented based on time differences. **SEGMENT\_DISPLAY\_COUNTDOWN** in the case of **ORBITING\_MODE** and **BATTERY\_RECOVERY** and **BATTERY\_CONSUMPTION** in the case of **EXPLORING\_MODE** are examples. Do note that these counters do not necessarily have to be relative to time; they can be relative to a certain value existing that happens to change independently of time.
  + For this case, let the counter once again be , and the constant to refer to be , where will determine the change to . is the variable that denotes the satisfactory result of . Note that the BLACK BOX and are interchangable as well.



**Flowchart and Summaries of Modes:**

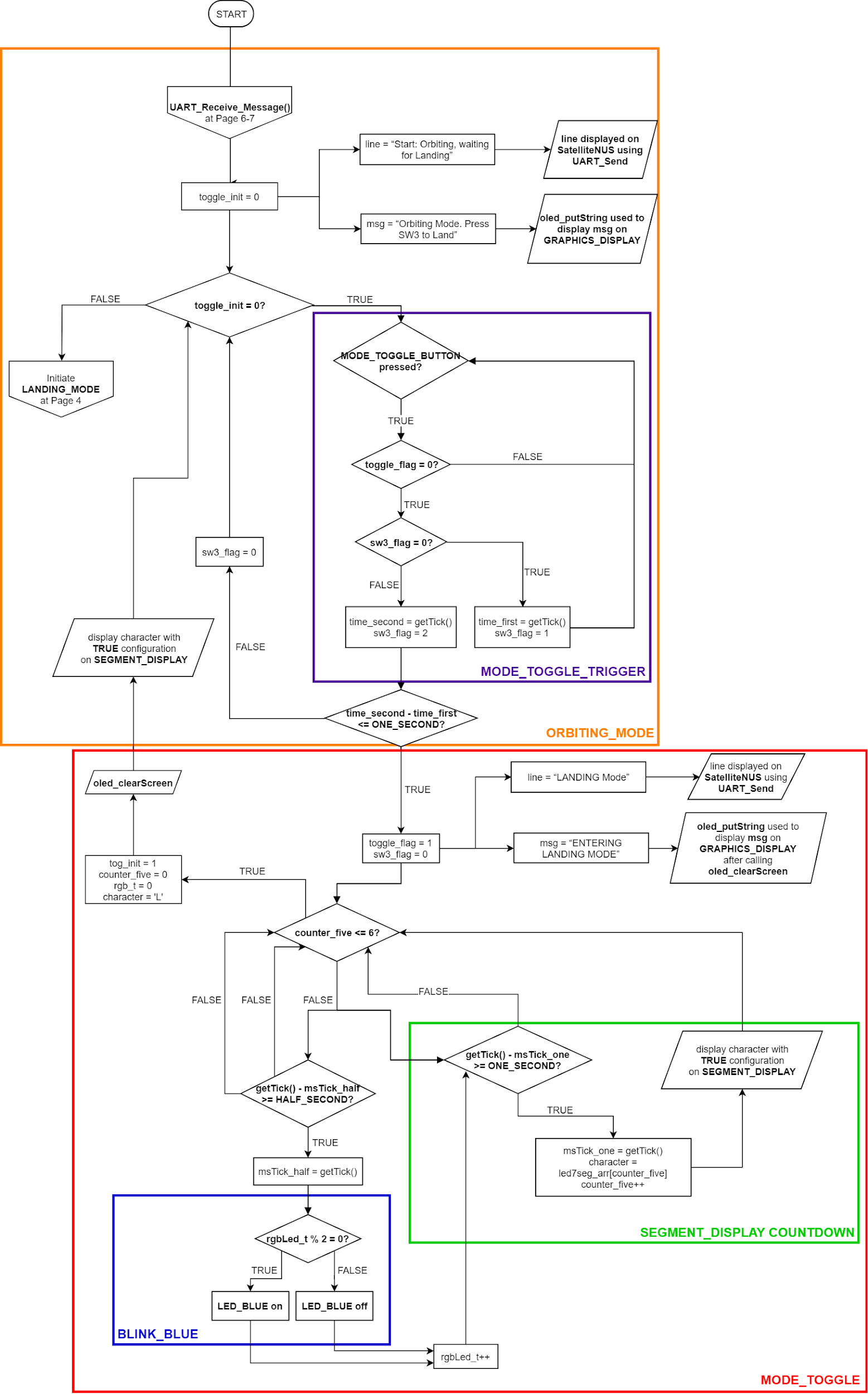
**ORBITING MODE:**

The start of the program will send the message “Start Orbiting, waiting for Landing” to **SatelliteNUS** and the OLED display will show “Orbiting mode. Press SW3 to Land”.

Then, as long as **toggle\_init** **= 0**, which is the check as to whether **MODE\_TOGGLE** is to be activated or not, there will be a continuous check on whether there are two presses on SW3 in 1 second. If within 1 second, there are two presses detected, there will be a counter incremental that will allow traversal through an array to display the 7-segment characters from 5 to 0 as shown in **SEGMENT\_DISPLAY\_COUNTDOWN**. Do note that here, there is a Type 2 counter that allows the traversal within the seven-segment display character array.

Through this while loop, a Type 1 counter will also be run before running **BLINK\_BLUE**, which is highly similar to the earlier discussed abstraction, **ARBITRARY\_BLINK**. This will go on until the required conditional is met as shown in the flowchart above. As shown, these two abstractions denoted earlier need not be mutually exclusive; they can have a one-way dependency relationship.

Then, once these are over, variables required in landing mode will be initialized and since in this implementation, **toggle\_init** never goes back to 0, landing mode will run immediately after the initialization of these variables are finished. The next page details the flowchart of **ORBITING\_MODE**.

****

**LANDING MODE:**

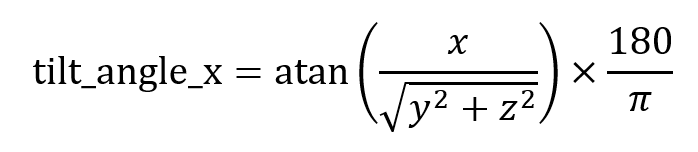
There is a continuous check as to whether thelight threshold is met. As long as it is not met, the system remains in **LANDING\_MODE**, where the red and blue LED alternates every 0.5s as shown in **ALTERNATE\_BLINK** and there are continuous accelerometer readings and updates on the OLED.

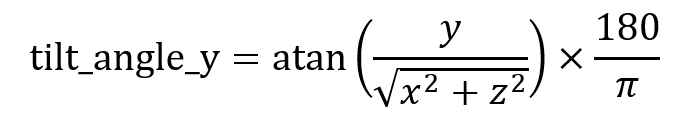
Notice that here, **ALTERNATE\_BLINK** corresponds to **ARBITRARY\_BLINK** and that there are no counters here at all.

If the angle of tilt is above 30**˚**, the system triggers a “Poor Landing Attitude” warning on the OLED and **SatelliteNUS** and continues to do so until the angle is below 30**˚.**

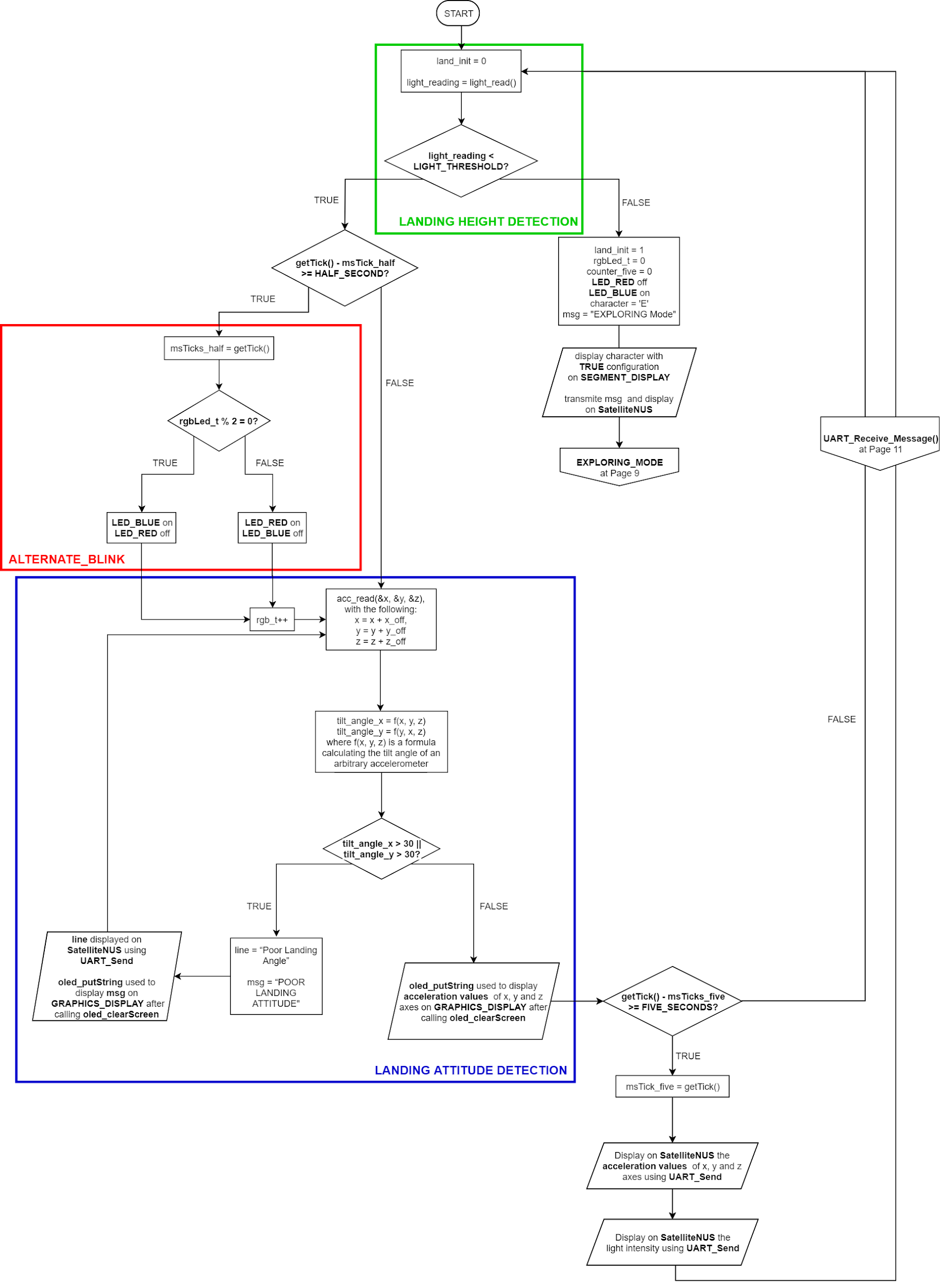
The light sensor interrupt alerts the system as soon as light thresholdis met, and the lander switches to **EXPLORING\_MODE**. **SatelliteNUS** can request for any sensor data from the Lander, as long as no warning messages are being transmitted.

The tilt angle is calculated as follows and its units is in degrees:





The next page details the flowchart for **LANDING\_MODE**.

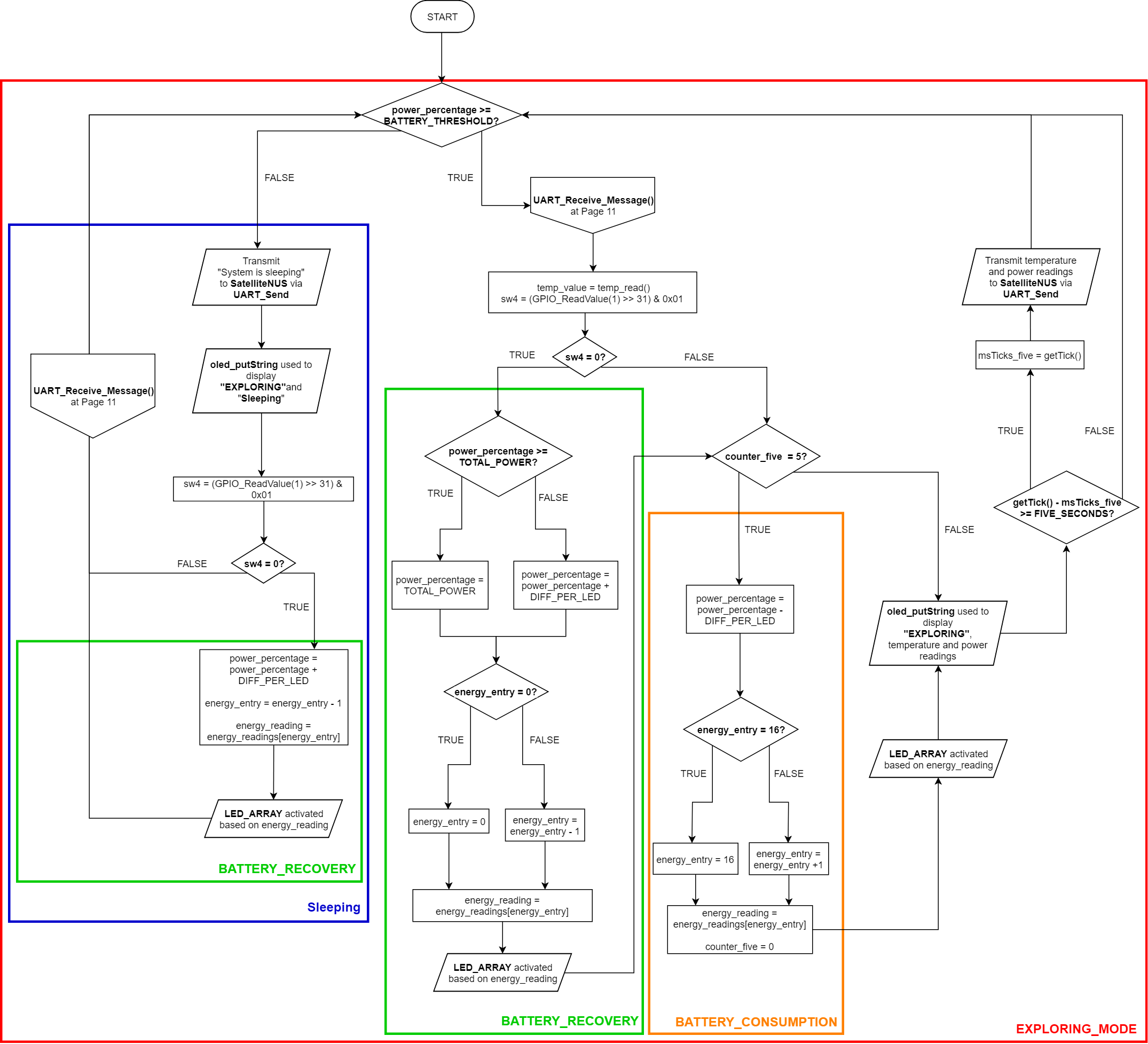
****

**EXPLORING MODE:**The temperature sensor is being continuously read.

The battery percentage reduces 12.5% every 10s. This is reflected in the 16-LED array by reducing the number of LEDs turned on by one every 5s corresponding to a 6.25% decrease in power. A array is used to store the LED masks for the different power percentages and 5 second counter that increments every second allows us to decrement the power percentage accordingly and move along the mask array. The **energy\_entry** variable contains the current location along the mask array. Incrementing it switches OFF more LEDs, decrementing it switches them ON.

When power percentage is below the threshold of 12.5%, LanderNUS goes to sleep and displays the message on the OLED and **SatelliteNUS**. The system remains asleep till SW4 is pressed. Pressing SW4 increases the battery percentage by 6.25%; this is reflected in the 16-LED array by decrementing along the Mask array. **SatelliteNUS** is only allowed to request sensor data while the system is awake.

Do note that both battery consumption and battery recovery modes have the earlier mentioned Type 2 counter, where the conditional to transmit messages to the UART essentially relies on the Type 1 counter.

****

**Enhancements**

1. **Light Sensor Interrupt**

The Light Sensor was provide a GPIO0 interrupt when the light intensity falls below **30 lux.** To setup, the following steps were taken:

1. The Interrupt Status Register in light sensor was cleared using **light\_clearIrqStatus()**.
2. The low threshold was set to 30 to interrupt below 30 lux using **light\_setLoThreshold(30)**.
3. Since the hardware interrupt is set as open-drain, the light sensor toggles from 1 to 0, hence a falling edge interrupt is set using **LPC\_GPIOINT ->IO2IntEnF |= 1 << 5**.
4. Since the GPIO0 interrupt would be handled by the **EINT3\_IRQn,** the pending interrupt is first cleared using **NVIC\_ClearPendingIRQ(EINT3\_IRQn)** and then enabled with **NVIC\_EnableIRQ(EINT3\_IRQn)**

EINT3 has a preempt priority of 0b10 and a sub-priority of 0b000. It has a comparatively lower preempt (3rd) with the other interrupts because during implementation the light sensor has lower importance compared to the System Clock and SW3.

1. **EINT0 Interrupt**

EINT0 interrupt was configured as SW3 interrupt instead of using a GPIO interrupt. This was primarily done to give it a different interrupt priority to the light sensor, which could be only be configured as a GPIO0 interrupt. Below details the steps for setting up the EINT0 interrupt:

1. Use **PINSEL4** and initialize the 20th bit in the register with ‘1’ using **LPC\_PINCON**. As control pin 20 of **PINSEL4** corresponds to the GPIO Port 2 and pin 10, which is the GPIO port-pin combination of SW4, this setup allows control pin 20 of **PINSEL4** and subsequently SW3 to function as an EINT0 interrupt via the operation of **LPC\_PINCON->PINSEL4 |= (0x01 << 20)**. The subsequent parts will use **LPC\_SC**, which essentially manipulates system control within the LPC1769 in order to set the control behaviour of **EINT0** accordingly.
2. Next, use **LPC\_SC** to initialize **EXTMODE** as **1**. According to Table 11. of the User Manual of the LPC17xx microcontroller, this initialization culminates the **EINT0** to be an **edge-sensitive interrupt**. This is done via **LPC\_SC->EXTMODE |= 1**.
3. Then, use **LPC\_SC** to initialize **EXTPOLAR** as **0**. According to Table 12. of the User Manual of the LPC17xx microcontrollers, this initialization culminates the **EINT0** to be a **falling-edge sensitive interrupt**. This falling-edge sensitive interrupt is used because of how the recording of SW3 being pressed and the subsequent activation of EINT0 is caused by a **toggle from 1 to 0**. This is done by calling **LPC\_SC->EXTPOLAR &= 0**.
4. Before enabling **EINT0**, **LPC\_SC** is used to clear the interrupt bit using **EXTINT** because not doing so will culminate in SW3 not being able to be used as an interrupt. The toggling of **EXTINT** as **1** will allow **EINT0 to EINT3** to be cleared. This is because **EXTINT** is an interrupt flagging register that contains flags in **EINT0 to EINT3**. This is done via **LPC\_SC->EXTINT = (1 << 0)**. It is also because of this that when **EINT0\_IRQHandler** is being run, the interrupt register is always cleared.
5. Finally, activate EINT0 using **NVIC\_EnableIRQ(EINT0\_IRQn).** This is in light of how the preceding procedures suggest that EINT0 is an programmable external interrupt and as such, can be activated accordingly.

EINT0 has a preempt priority of 0b01 and a sub-priority of 0b000. It has a comparatively higher preempt

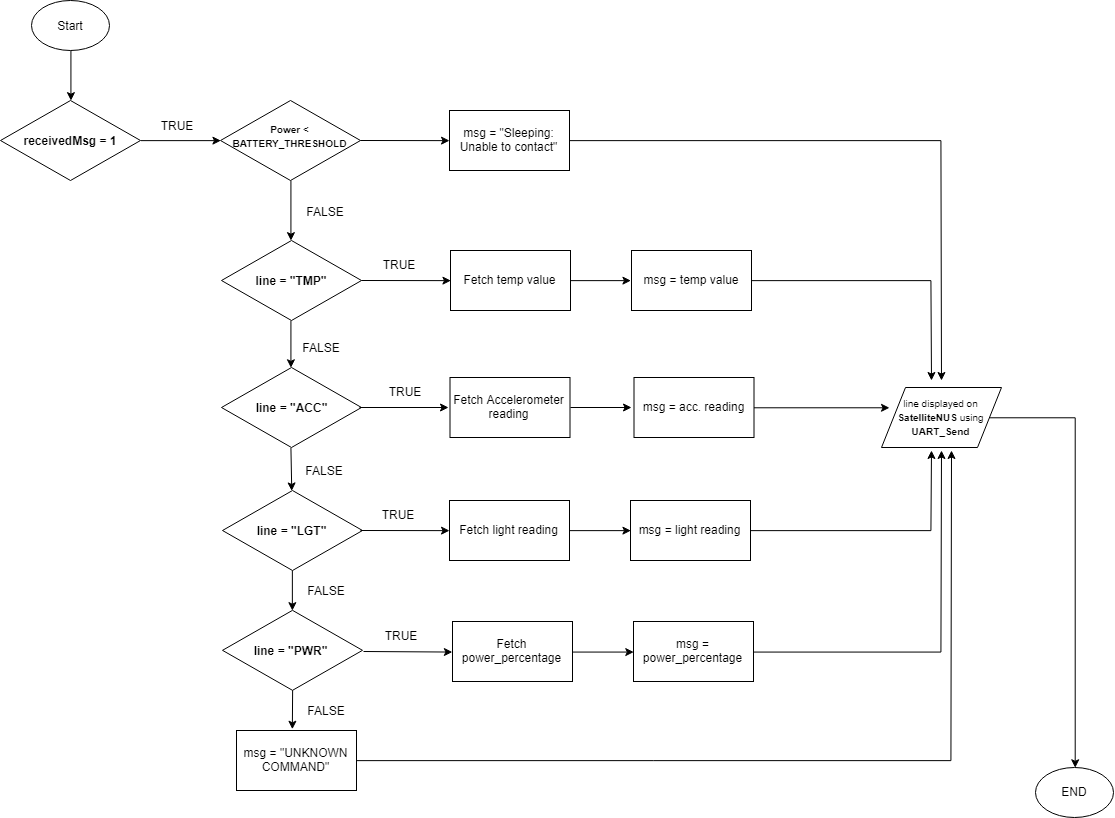
and equal sub-priority with the other interrupts because during implementation and operation of the other

modes, the toggling of SW3 is essentially what activates **MODE\_TOGGLE**, and then the other modes.

1. **UART3 Interrupt**

The UART3 Interrupt was configured to alert whenever a message is sent from TeraTerm to LPC. This message is a command to send back values of a specific sensor. To configure it, the following steps according to the User Manual were taken:

1. Using a **UART\_FIFO\_CFG\_Type,** the RX trigger level 0 for one character (0b00) is set for RDA (Receive Data Available) Interrupt.
2. Both RX FIFO Reset and TX FIFO Reset is cleared and the DMA Mode is disabled.
3. The RDA Interrupt is configured using **UART\_FIFOConfigure()**
4. Finally the **UART\_IntConfig()** is used to enable RBR Interrupt.
5. To clear the pending Interrupts from UART3 in the NVIC, **NVIC\_ClearPendingIRQ(UART3\_IRQn)** is called.
6. Finally, the UART3 Interrupt is enabled in NVIC using **NVIC\_EnableIRQ(UART3\_IRQn)**.
7. To identify which interrupt was triggered, the Interrupt Identification Register **LPC\_UART3->IIR** was read. A value of 0b0100 would be reflected in **LPC\_UART3->IIR** if the RDA was triggered.

UART3 has a preempt priority of 0b11 and a sub-priority of 0b000. It has the lowest preempt (4th) compared to the other interrupts because during implementation received command has lower importance compared to the System Clock, SW3 and the light sensor. Below shows the commands for UART\_ReceiveMessage().

1. **LED 7-Segment**

The original 7-segment library used would display the character inverted as compared to the OLED. This was not user friendly and difficult to read. The "led7seg.c" file was modified to invert the character and display it on the 7-segment. After analysing the bit pattern corresponding to the 7-segment LEDs, the order was found to be FBDpCAGED not the expected ABCDEFGDp.

**Significant Problems & Solutions**

1. **Systems level consideration**

During the project, one key problem was about how to append the separate modes into a single coherent machine capable of meeting the design specifications. The most noticeable issue about trying to put every component together was how each sub-part should be positioned and placed, as the components that worked and their corresponding behaviours within each mode implied that positional difficulties were bound to happen because of how the same lines of code, when placed in different lines, could have varying unintended behaviour.

To resolve this dilemma, a divide-and-conquer approach was adopted, where an initial “prototype” function was defined and the foundations of setting up the modes were essentially accomplished within. This gave a better overview on the project as to the type of devices required within each mode and how the codes should be positioned.

1. **How to initialize EINT0**

Upon meeting every basic requirement, one such enhancement we immediately thought of was how to use EINT0 instead of relying on purely EINT3. However, the lecture notes had not covered EINT0 and while the LPC17xx User Manuals had explicitly stated the registers required to initialize and subsequently use EINT0 and its corresponding behaviours, studying how EINT3 was set up yielded very little insights to the initialization of EINT0 as both interrupts were set-up very differently.

Through looking up the internet and the lpc17xx.h, it was eventually realized that LPC\_PINCON and LPC\_SC could be used to set up EINT0, and what LPC\_PINCON and LPC\_SC are actually used for. They are essentially typedef structs that are integral to setting up various other functions critical to the operation of the LPC1769.

1. **UART3 Interrupt and Messages received**

The UART3 interrupt was used to alert whenever a message was received from TeraTerm. The UART3 Interrupt configuration requires a few extra considerations compared to GPIO interrupts. Understanding the need for those steps took up some time. The lecture notes did help, but more research in terms of going through the libraries and user manual was required. In addition, the messages received contained extra new line characters (\n) even though the string was cleared after every interrupt. The messages appeared to contain extra \n only when messages of different length was sent. To resolve this, a 3 letter code was given to every command sent from TeraTerm.

**Suggestions**

**Kendrik:**

The codes given accelerated the progress of this project and as such allowed us to cover a fair bit of enhancements despite being overwhelmed by projects, assignments and assessments from other modules. I think this is commendable and should remain for the future batches. The same holds for the case of the introduction of flowcharts.

However, what I feel may require improvement is the way lectures are being conducted in the first half of the module. While the lab sessions did allow us to reconcile what we learned in lecture with what we did in labs and tutorials, in my opinion, there is a massive disjoint in the existing method of teaching EE2028. Essentially, the current methodology is to attend lectures to learn about microcontroller programming and then actually understand what is going on by having hands-on experience with them in lab sessions and tutorials that try to check that students understand the concepts. However, the lack of a tangible means to showcase the concepts taught in lectures and tutorials on the immediate term means that students are usually not able to appreciate the concepts taught in the early half of the module until Assignment 2 is released, or when labs have commenced. This usually means a waste of time and effort in attempting to bring across the concepts in the middle of tutorial. This problem is less prominent in the second half, as we have been sufficiently inducted into the use of the LPC1769 and the baseboard, with the content being significantly more interesting and yet theoretical such that there is a lower reliance on the baseboard.

As such, I suggest the introduction of a software that allows students to practice assembly skills before labs. This would allow students to practice these skills and use them effectively and meaningfully such that the second half would not feel overwhelming to them.

**Varun:**

Overall, the module is well-planned. It contains in-depth concepts that requires understanding, the lab sessions and tutorials further complement the content taught in lectures. Most of the time in lab sessions were spent in briefings instead of working with LPC. While i understand the need for briefings; it makes it easier for us to adapt to the new environment and understand the working better, there is a need for longer in-lab practise. Since the first half of the module is less work intensive as compared to the second half, some content for the labs can be covered earlier and the two gap weeks for lab can be avoided. This would give us more time to practise in class and consult with the GAs and the tutors.

**Conclusion**

Through this assignment, we have learned about the various protocols a microcontroller can have as well as the underlying concepts that when pieced up together, build up the microcontroller. Through the project, we have learned how to refer to datasheets and user manuals and seek information wherever necessary to obtain the data we need as well as to understand how various devices can take readings, transmit data and even be used. By the use of interrupts, we have learned how to compact our code and make it comparatively more efficient. While there were various problems arising from this project, we were able to overcome most of it to present the LanderNUS with the aid of Dr Gu Jing, Dr Rajesh, Prof Tham and Christopher.