

**EE/CSE 474 Lab Report**  
**Project 4/5**



**Radleigh Ang**  
**Abdul Katani**  
**Daniel Snitkovskiy**

# TABLE OF CONTENTS

<b>ABSTRACT</b>	<b>5</b>
<b>1. INTRODUCTION</b>	<b>8</b>
<b>2. DESIGN SPECIFICATIONS</b>	<b>9</b>
2.1 System Requirements	9
2.1.1 General Summary of Satellite Management and Control System	9
2.1.2 Subsystems Overview	10
2.1.3 Scheduler for these Tasks	12
2.2 Specific Details and Requirements	12
2.2.1 Tasks and Task Control Blocks	12
2.2.2 Requirements for Power Subsystem	14
2.2.3 Requirements for Thruster Subsystem	15
2.2.4 Requirements for Satellite Coms	16
2.2.5 Requirements for Command Parser	18
2.2.6 Requirements for Console Display	19
2.2.7 Requirements for Warning Alarm	19
2.2.8 Requirements for Solar Panel Control	20
2.2.9 Requirements for Console Keypad	20
2.2.10 Requirements for Vehicle Comms	21
2.2.11 Requirements for Transport Distance	22
2.2.12 Requirements for Image Capture	23
2.2.13 Requirements for Pirate Detection	23
2.2.14 Requirements for Pirate Management	23
<b>3. SOFTWARE IMPLEMENTATION</b>	<b>24</b>
3.1 High-Level Block Diagram	24
3.2 General UML Diagrams	25
3.3 UML Activity Diagrams	28
3.3.1. Warning Alarm	28
3.3.2. Power Subsystem	29
3.3.3. Thruster Subsystem	30
3.3.4 Pirate Detection/Management	31
3.4 UML Sequence Diagram	32

3.5 Function Prototypes for Tasks	33
3.6 Data Structs Defined for Application	34
3.7 Pseudocode	37
3.7.1 WarningAlarm Pseudocode	38
3.7.2 SatelliteComs Pseudocode	39
3.7.3 ConsoleDisplay Pseudocode	40
3.7.4 Power Subsystem Pseudocode	41
3.7.5 Thruster Subsystem Psuedocode	44
3.7.6 solarPanelControl Pseudocode	46
3.7.7 commandParser Pseudocode	47
3.7.8 keyboardConsole Pseudocode	50
3.7.8 vehicleComs Pseudocode	51
3.7.9 Dynamic Task Queue	52
Design Decisions:	54
3.7.10 Startup Task Pseudocode	55
3.7.11 Scheduler Pseudocode	56
3.7.12 Image Capture Pseudocode	58
3.7.12 Transport Distance Pseudocode	59
3.7.12 Pirate Detection Pseudocode	61
3.7.12 Pirate Management Pseudocode	61
<b>4. TEST PLAN</b>	<b>62</b>
<b>5. TEST SPECIFICATION</b>	<b>63</b>
<b>6. TEST CASES</b>	<b>67</b>
<b>7. PRESENTATION AND RESULTS</b>	<b>73</b>
7.1 Output of the System	73
7.1.1 Terminal Interface (Start tasks and stop tasks)	73
7.1.2 Battery Level	74
7.1.3 Battery Temperature/Warning Alarm	76
7.1.4 Vehicle Comms	78
7.1.5 Transport Distance	78
7.1.6 Pirate Detection/Management	80
7.1.7 Image Capture	81

7.2 Results for Task Execution Time	83
7.3 Lab Spec Questions	85
<b>8. ANALYSIS OF ANY RESOLVED ERRORS</b>	<b>86</b>
8.1 Measuring Errors for files	86
8.2 GPIO Pin Error in measurement	88
<b>9. ANALYSIS OF UNRESOLVED ERRORS</b>	<b>89</b>
9.1 Task Time Standard Deviation	89
9.2 Battery Level Updating Error	89
9.3 Image Capture Limitations	90
<b>10. SUMMARY</b>	<b>90</b>
<b>11. CONCLUSION</b>	<b>91</b>
<b>12. APPENDICES</b>	<b>92</b>
<b>13. CONTRIBUTION</b>	<b>92</b>

## ABSTRACT

The objective of these labs were to incrementally add features and capabilities to the Satellite Management and Control System begun in Project 2. These features included:

1. Additional tasks to:
  - a. Monitor battery temperature and respond to critical conditions.
  - b. Support image capture.
  - c. Support information exchange between the comms link and the mining vehicle.
  - d. Support transport vehicle docking.
  - e. Facilitate a web browser type interface linking the earth based terminal and the satellite system, modeling a protocol and messaging ('command') subsystem.
  - f. Detect pirates and 'manage' them accordingly.
2. Modifying the dynamic task queue to support task prioritization.
3. Incorporating software interrupts into the overall flow of control of the system.

To meet these objectives, we started by creating diagrams to get a better understanding of the expected output and flow of control of the updated system. After this, we proceeded to implement the requirements in the following order:

1. Modifying the definition of a "TCB" and the dynamic task queue to support task prioritization.
2. Implementing functionality/tasks that are hardware independent.
3. Integrating additional hardware peripherals and unit testing each non-communication based TCB (e.g. batteryTemperature).
4. Amending existing communication TCBs related to output/communication (Satellite Comms, Console Display, Vehicle Comms), and integrate these with the new "Web Browser Messaging Protocol." (i.e. the commandTask)

After integrating these components, we utilized the GPIO on the Beaglebone to update the measured execution times of each of the TCBs and updating the delays for the “major” and “minor” cycles, accordingly. The tasks that execute on a “minor” cycle (i.e. every 100 milliseconds) are: Warning Alarm, Satellite Comms, Vehicle Comms and Console Display, while all eleven TCBs execute on a “major cycle” (i.e. every 5 seconds).

The resulting system has the following behavior: First, the status and annunciation information can be printed on two terminals: one for the satellite, and one for the Earth. To control what’s being printed, the user controls a second, “input” terminal that can switch the display mode for the satellite terminal (status or annunciation) Additionally, the input terminal can communicate with the vehicle via the satellite, printing the vehicle’s response on the Earth terminal. The input terminal can control the speed at which the motor drive operates the solar panels. Next, the input terminal can display pertinent information to the satellite such as battery level, battery temperature, transport distance, et cetera. Commands to retrieve current measurements, issue thruster commands, start/stop data collecting tasks, and display the status/annunciations can be sent through the earth terminal’s web browser-like interface.

If the battery or fuel is low, the system flashes leds at specific intervals on the panel that serve as a warning. If an overheating event occurs, the console will alert the user by changing the terminal screen, and flash the leds at a 10Hz rate when unacknowledged for more than 15 seconds. The system can also communicate with a transport vehicle on the mining surface, sampling the frequency from the transport vehicle’s transmitter and calculating its to the satellite. Moreover, the user can call the vehicle to capture an image, and displays the resulting frequencies from the image capture task. Lastly, the system has built in pirate detection and pirate management subsystems to launch and deter any alien pirates from stealing the valuable raw material.

Internally, the system's thrusters are controlled by user input which is parsed for thruster magnitude, duration, and direction, and used to drive a PWM. The battery is connected with a voltage divider circuit, limiting the max output to 1.8V. This voltage is read through an ADC, converted to a digital measurement from 0 to 36V, and displayed on the Earth and satellite terminals. Lastly, the solar panels are also driven with a PWM; its motors varied via user input or an automatic setting.

Through this lab, we have cemented our understanding of the C language, refined our use of formal design tools, and utilized several pieces of external sensors and devices to realize a more complete model of a real-time Embedded System.

# 1. INTRODUCTION

In this lab, our main goal was to refine the design and implementation of our Satellite Management and Control System. Using the rapid prototyping design methodology, we iterated on Phase II's design, adding functionality that improved the safety, reliability, and capability of the resulting system.

Features such as software interrupts and command parsing helped make the system more robust and responsive to user input via hardware initialization confirmation and a bidirectional, web-browser user interface. Other features, such as added ADC channels (e.g. imageCapture), helped realize a more complete model of the target system (e.g. allowing for the form of mining image sampling data to be accurately simulated.) All of these features were conceived, implemented, and tested through the course of this project, with an analysis of the results and errors at the end.

Below are the details of environment/tools that was utilized throughout the various stages of this project. For tools that were used in specific parts, the context of use will be explained in the appropriate section.

## **Environment:**

- AM3358BZCZ100 BeagleBone Black
- Ubuntu 16.04.2 LTS Work Station
- GDB 7.4.1 Debian debugger
- GCC 4.6.3 Debian compiler



**Tools:**

- Scope Probe
- 9 volt battery
- 2 resistors (40.02 k $\Omega$  and 100  $\Omega$  )
- Potentiometer with max resistance of 1 k $\Omega$
- Breadboard Wires
- Hardware Counter
- Buzzer
- External LEDs

## **2. DESIGN SPECIFICATIONS**

This section shows the design specifications for Project 4/5.

### **2.1 System Requirements**

#### **2.1.1 General Summary of *Satellite Management and Control System***

- Main purpose of *Satellite Management and Control System*:
  - Control surface-based mining equipment from an orbiting command center and communicate with various Earth stations.
- Specific functions include:
  - Displaying status, warning, and alarm information on the Satellite console (modeled with a terminal display in the Linux environment).
  - Sending pertinent mission information and receiving commands from Earth station.
  - Including a startup to initialize the system.
  - Creating a dynamic task queue to manage solar panel deployment.
  - Measuring the battery level through an analog to digital converter (ADC).
  - Extend communication with a mining vehicle via the satellite terminal.

- Detect battery temperature using two transducers.
- Capture images by sampling through the ADC at a set sampling frequency, and converting them into frequencies to be delivered.
- Receive and interpret commands from the Satellite Coms tasks, directing the local subsystems to perform the requested tasks.
- Measure analog signals from an alien pirate transducer and deploy thermonuclear devices as a deterrence.
- *For an external view of system functionality, see the “Use Case Diagram” (Figure 1) in section 3.2.*

### 2.1.2 Subsystems Overview

The purpose of the system is to display alarm information, satellite status, and support basic command/control signaling. The following is a general overview of the subsystems (*See the “Functional Decomposition” (Figure 2) in section 3.2 for a high level summary*):

- Power Management Control
  - Tracks these variables:
    - Power Consumption of Satellite
    - Battery Level
  - If Battery Level is low, deploy solar panels, else, retract solar panels.
- Thruster Management and Control
  - Control duration and magnitude of thruster burn to move in desired direction.
  - Tracks direction:
    - Left
    - Right
    - Up
    - Down

- Tracks Thruster Control:
  - Thrust (Full OFF or Full ON)
- Thrust Duration
- Communications Requirements
  - Supports communication to earth through...
    - Status
    - Annunciation
    - Warning information
  - Currently, incoming commands are limited to thruster control.
  - Supports bidirectional communication with the land-based mining vehicle to receive mission commands from the earth station.
- Status and Annunciation Subsystem Requirements
  - Displays status, annunciation, and alarm information via the console and the lights on the front panel.
  - Tracks these variables for Status:
    - Solar Panel State (retracted or deployed)
    - Battery Level
    - Power Consumption and Generation
    - Fuel Level
  - Tracks these variables for Warning and Alarm:
    - Fuel Low
    - Battery Low
  - Warning Alarm controls the function of the LEDs based on the battery/fuel levels. The LEDs will flash at a rate corresponding to the amount of battery (LED2) and fuel (LED1) left in the system:
    - Flashes every two seconds if between 10% and 50%
    - Flashes every second if less than 10%
    - If both battery and fuel are > 50%, turn on LED3.

### **2.1.3 Scheduler for these Tasks**

The schedule task manages the execution order and period of the tasks in the system.

The scheduler executes tasks in a round robin fashion, utilizing a major cycle and a series of minor cycles. The period of the major cycle is 5 seconds, while the duration of the minor cycle is 100 milliseconds. Following each major cycle through the task queue, the scheduler will cause the suspension of all task activity, except for the operation of the warning and error annunciation, for five seconds. In between major cycles, there shall be a number of minor cycles to support functionality that must execute on a shorter period. (i.e. the Warning Alarm).

## **2.2 Specific Details and Requirements**

### **2.2.1 Tasks and Task Control Blocks**

The task control block represents a task that is to be implemented as part of the system software.

The system will need to cycle through these task control blocks through a dynamic task queue.

The eleven TCB elements in the queue correspond to tasks managing the powerSubsystem, solarPanelControl, thrusterSubsystem, satelliteComms, vehicleComms, consoleDisplay, consoleKeypad, warningAlarm, imageCapture, pirateDetection, and pirateManagement. During startup, only warningAlarm and satelliteComs are executed normally. The other tasks are added or removed depending on user input.

The Task Control Block (TCB) is implemented as a C struct; utilizing a separate struct for each task. Each TCB will have five members:

1. The first member is a pointer to a function taking a void\* argument and returning a void.
2. The second member is a pointer to void used to reference the data for the task.
3. The third and fourth members are pointers to the next and previous TCBs in a doubly linked list data structure.
4. The fifth member, is the task's priority.

The following gives a C declaration for such a TCB.

```
struct MyStruct
{
    void (*myTask)(void*);
    void* taskDataPtr;
    struct MyStruct* next;
    struct MyStruct* prev;
    int priority;
};
typedef struct MyStruct TCB;
```

### 2.2.2 Requirements for Power Subsystem

The powerSubsystem manages the satellite's power subsystem, monitoring the battery level, the solar panels' deployment, and the power consumed/generated by the system. The following operations are to be performed on each of the data variables referenced in powerSubsystemData (the struct held by the taskDataPtr).

#### powerConsumption:

Increment the variable *powerConsumption* by 2 every even numbered time the function is called and decrement by 1 every odd numbered time the function is called until the value of the variable exceeds 10. The number 0 is considered to be even. Thereafter, reverse the process until the value of the variable falls below 5. Then, once again reverse the process.

#### powerGeneration:

If the solar panel is deployed:

    If the battery level is greater than 95%

        Issue the command to retract the solar panel

    Else

        Increment the variable by 2 every even numbered time the function is called and by 1 every odd numbered time the function is called until the value of the battery level exceeds 50%. Thereafter, only increment the variable by 2 every even numbered time the function is called until the value of the battery level exceeds 95%.

If the solar panel is not deployed:

    If the battery level is less than or equal to 10%

        Issue the command to deploy the solar panel

    Else

        Do nothing.

#### batteryLevel

- The battery level is to be read as an analog signal on A/D Channel 0, AIN0. The A/D is a 12 bit (0-4095 binary value) device with a resolution of 0.44 mV per bit. The A/D will return the reading as a 4 digit decimal number. 1.8v -> 1800.
- Post measurement, the measured value must be scaled back into the 0 to +36V range for display. The system shall store the 16 most recent samples into a circular buffer.

### **2.2.3 Requirements for Thruster Subsystem**

The thrusterSubsystem task handles satellite propulsion and direction based upon commands from the Earth.

The task interprets each of the fields within the 16-bit thruster command and generates a control signal of the specified magnitude and duration to the designated thruster.

- Bits 15 through 8 controls the thruster's duration
- Bits 7 through 4 controls the magnitude
- Bits 3 to 0 controls thruster direction.

The thruster control signals, for a specified duration, have magnitudes of 0%, 50%, and 100% of full scale. If fuel is expended at a continuous 5% rate, the satellite will have a mission life of 6 months. The thrusterSubsystem will update the state of the fuel level based upon the use of the thrusters. The thruster system will also be driven with a PWM signal. The PWM's duty cycle and period is determined by the magnitude and duration of the thruster command.

#### **2.2.4 Requirements for Satellite Coms**

Satellite Coms handles the communication with earth. It receives and sends any data to be displayed on a remote display, presenting the following information:

- The name of the satellite
- Current date
- The operator's name
- Data, status, and warning information
  - Fuel Low
  - Battery Low
  - Solar Panel State
  - Battery Level
  - Fuel Level
  - Power Consumption
  - Battery Temperature
  - Transport Distance
  - Image Data

The data, status, and warning information will be displayed based on user input and transmission from the Command Parser Task.



Satellite Coms also takes in a raw user input for the thruster command. This input is sent to the Command Parser task to be parsed and sent to the thruster subsystem.

The thruster command shall be interpreted as follows:

*Thruster ON Bits 3-0*

*Left 0*

*Right 1*

*Up 2*

*Down 3*

*Magnitude Bits 7-3*

*OFF 0000*

*Max 1111*

*Duration - sec Bits 15-8*

*0 0000*

*255 1111 1111*

### 2.2.5 Requirements for Command Parser

The Command Parser task shall receive a command from the Satellite Coms task and format an outgoing message to be transmitted to a remote computer.

When a message is received:

1. The task first verifies if the received message is valid
2. If invalid, send an error response back to SatelliteComs
3. If valid, interpret the command using the legal command list and respond accordingly as specified:

Command	Description
S	The S command indicates START mode. The command shall start the embedded tasks by directing the hardware to initiate all the measurement tasks. In doing so, the command shall enable all the interrupts.
P	The P command indicates STOP mode. This command shall stop the embedded tasks by terminating any running measurement tasks. Such an action shall disable any data collecting interrupts.
D	The D command enables or disables the display.
T<payload>	The T command transmits the thrust command to the satellite.
M<payload>	The M command: Requests the return of the most recent value(s) of the specified data. The M response: Response returns of the most recent value(s) of the specified data.
A<payload>	The A response acknowledges the recipient of the identified command.
E	The E error response is given for incorrect commands or non-existent commands.

### 2.2.6 Requirements for Console Display

The *ConsoleDisplay* task will support two modes: *Satellite Status* and *Annunciation*.

In the *Satellite Status* mode, the following will be displayed for the satellite

- Solar Panel State
- Battery Level
- Fuel Level
- Power Consumption
- Battery Temperature
- Transport Distance

In the *Annunciation* mode, the following will be displayed

- Fuel Low
- Battery Low

### 2.2.7 Requirements for Warning Alarm

The *warningAlarm* task interrogates the state of the battery and fuel level to determine if they have reached a critical level.

- If both are within range, the LED3 on the annunciation panel shall be illuminated and on solid.
- If the state of the battery level reaches 50%, LED2 on the annunciation panel shall flash at a 2 second rate.
- If the state of the fuel level reaches 50%, LED1 on the annunciation panel shall flash at a 2 second rate.
- If the state of the battery level reaches 10%, LED2 on the annunciation panel shall flash at a 1 second rate.
- If the state of the fuel level reaches 10%, LED1 on the annunciation panel shall flash at a 1 second rate.

- If the over temperature event occurs, an audible alarm shall be set and shall remain active until acknowledged. If the alarm is unacknowledged for more than 15 seconds, LEDs 1 and 2 shall flash with the pattern: flash for 5 seconds at a 10Hz rate – remain solid on for 5 seconds repeat cycle

### **2.2.8 Requirements for Solar Panel Control**

The solarPanelControl task manages the deployment and retraction of the satellite's solar panels, and is to be scheduled on demand.

- The system shall provide a PWM signal to drive an electric motor. The speed shall be set to a default value or manually controlled from the command console. The speed shall range from full OFF to full ON.
- The system shall be capable of providing a PWM signal to drive the electric motor that deploys or retracts the solar panels. The period of the PWM signal shall be 500 ms. The speed shall be controlled either manually through pushbuttons on an earth based command and control console or set, based upon a preset value.
- If the panels are being controlled manually, the speed shall be incremented or decremented by 5% for each press of the corresponding console pushbutton.

### **2.2.9 Requirements for Console Keypad**

The console keypad is used to manually control the solar panel drive motors in increments of  $\pm 5.0\%$ . The keypad shall be interrogated for new key presses on a two-second cycle or as needed. The task is only scheduled during deployment or retraction of the solar panels.

### **2.2.10 Requirements for Vehicle Comms**

The vehicleComms task will manage a bidirectional serial communication channel between the satellite and the land based mining vehicle.

The exchange shall comprise specified commands from earth as well as returned status, warning, and alarm information. Information returned from the vehicle will be relayed by the satellite to the earth where it can be displayed using a Linux terminal window.

In the final phases, the following commands and responses will be implemented:

- Commands to Mining Vehicle:
  - F Forward
  - B Back
  - L Left
  - R Right
  - D Drill down – Start
  - H Drill up – Stop
  - S Start image capture
  - I Send image data
- Response:
  - W Image complete
  - P Image data
  - A <sp Command sent>

Additionally, the following commands are to be implemented for the requests between the Mining and Transport Vehicles:

- Requests from the Mining and Transport Vehicles:
  - T Request for transport lift-off
  - D Request to dock
- Response:
  - K OK to lift-off
  - C Confirm dock

### **2.2.11 Requirements for Transport Distance**

This task tracks the distance between the transport vehicle and satellite. The transport vehicle transfers the mined minerals back to earth, so it's launched from the land based vehicle, and rendezvous/docks with the satellite.

The transport is equipped with a transmitter that is automatically activated when it is within 1 km of the satellite. The task will track this distance by detecting the frequency of the emitted signal from the transport's transmitter.

The distances are stored in a circular eight reading buffer when the current reading differs by more than 10% of the previous stored reading. The transmitter - receiver pair are under development, so the frequency limits will correspond to distances between 100 meters and 2 kilometers.

### **2.2.12 Requirements for Image Capture**

*Image Capture* captures the output of an analogue scanner used to collect thermal image data from the surrounding environment. The measurements are taken from a thermal transducer, which are then ran through an fast fourier transform algorithm, outputting component frequency values. These values are sent, on command, to the remote data collection system.

The 16 most recent frequency values are tagged and retained in a buffer for transmission and presentation.

### **2.2.13 Requirements for Pirate Detection**

*Pirate Detection* detects any obstacles that appear in front of the satellite. When an alien pirate vehicle appears within 100 meters of the satellite, a *detected* flag is set, scheduling the *Pirate Management* task. The detection task shall provide proximity data to the *Pirate Management* subsystem.

### **2.2.14 Requirements for Pirate Management**

The system allows the user to release a blast of phasor fire and barrage photo torpedos to manage any alien vehicles appearing in front of the satellite.

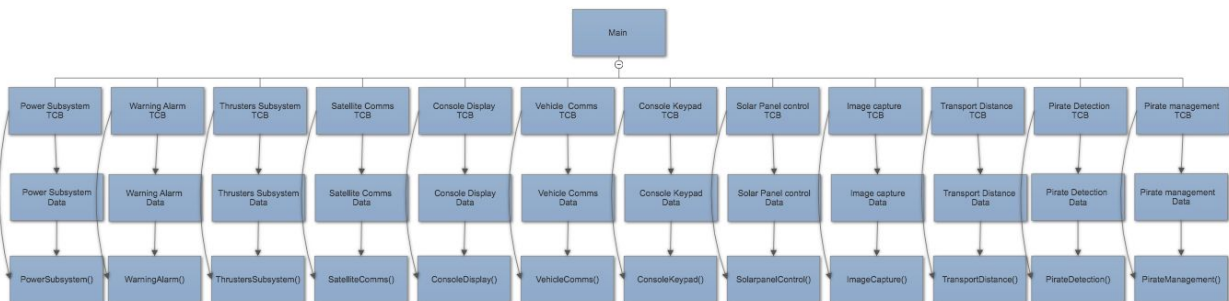
The management system shall operate as follows:

- If the *pirateProximity* is within 30 meters, the phasor subsystem is enabled and the communications officer can fire phasors at will.
- If the *pirateProximity* is within 5 meters, the photon subsystem is enabled and the communications officer can fire photons at will – or any other obstacle that might be interfering with the satellite’s progress.
- The *Pirate Management* subsystem shall fire one phasor burst or one photon torpedo at a time in response to each command.

### 3. SOFTWARE IMPLEMENTATION

The following section describes in detail how the specifications was implemented in this project. First, a top level view of the code will be explored through a high-level block diagram, and the individual TCB methods will be modeled via UML Activity Diagrams. Then, the relationship and sequence of all the TCBs will be modeled via UML Sequence Diagram. Finally, the function prototypes, data structs, and pseudocode implementation of the project will be examined.

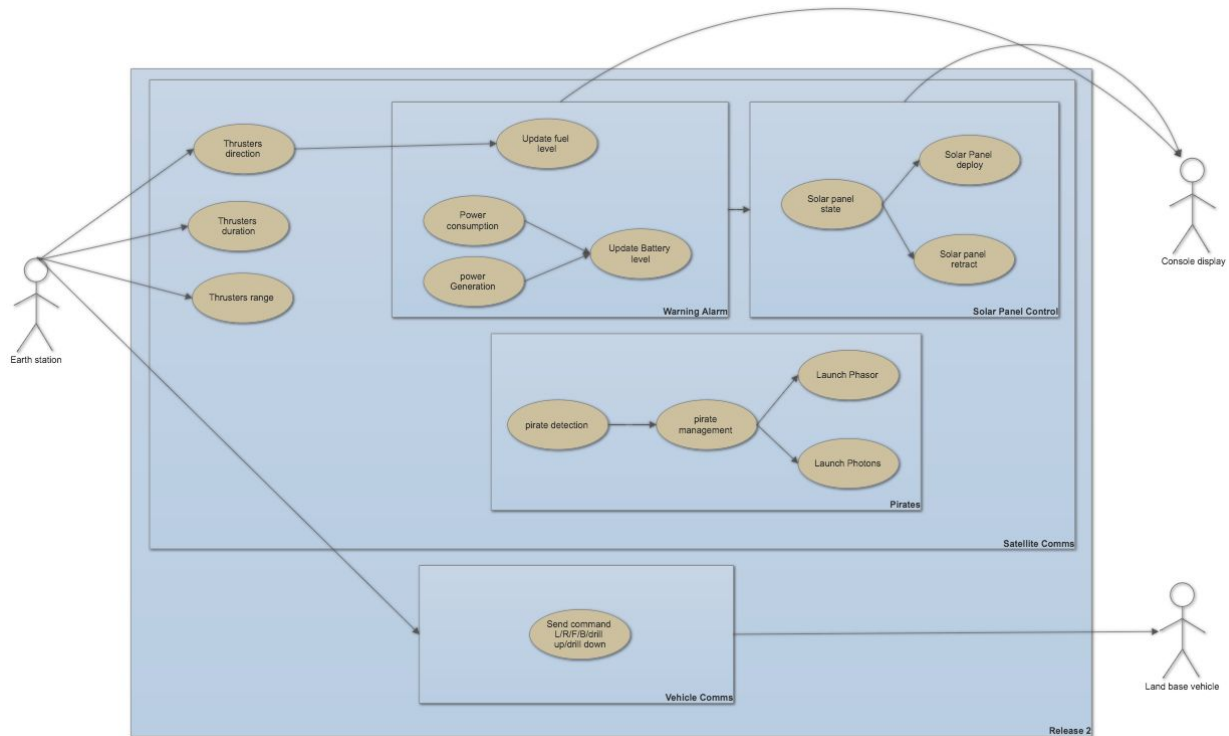
#### 3.1 High-Level Block Diagram



From this diagram, we see that the main function coordinates the eleven different TCBs. Within each TCB is a reference to a struct with the shared variables and a function that uses them. For example, we see that main creates a Power Subsystem TCB which points to shared data. Both the TCB and the shared data are utilized in the powerSubsystem function. This same concept applies to the ten other tasks.

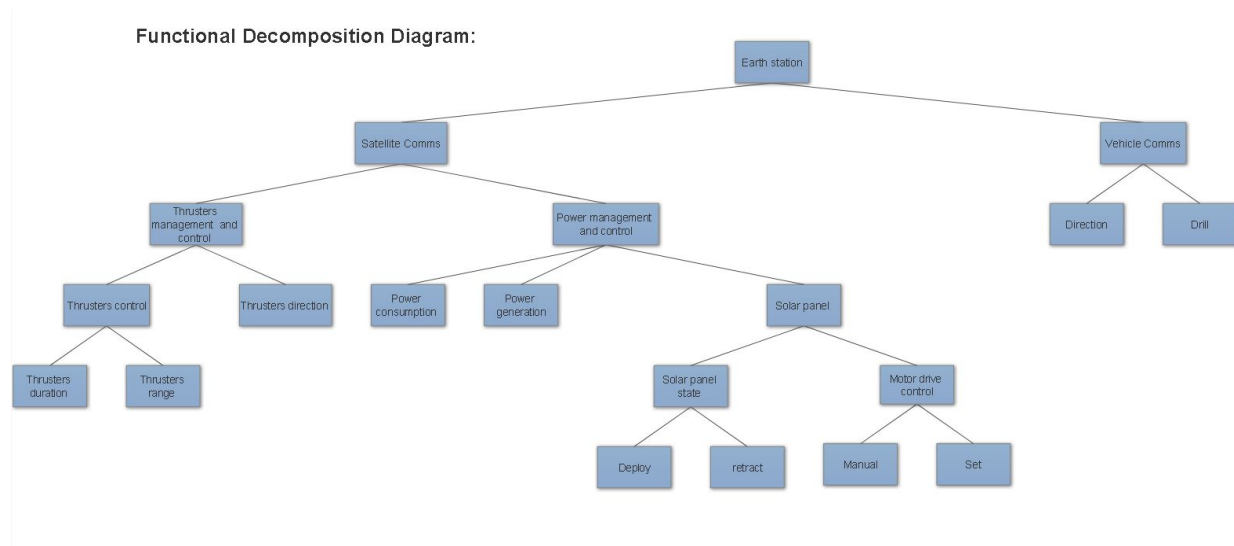


## 3.2 General UML Diagrams



**(Figure 1: Use Cases Diagram)**

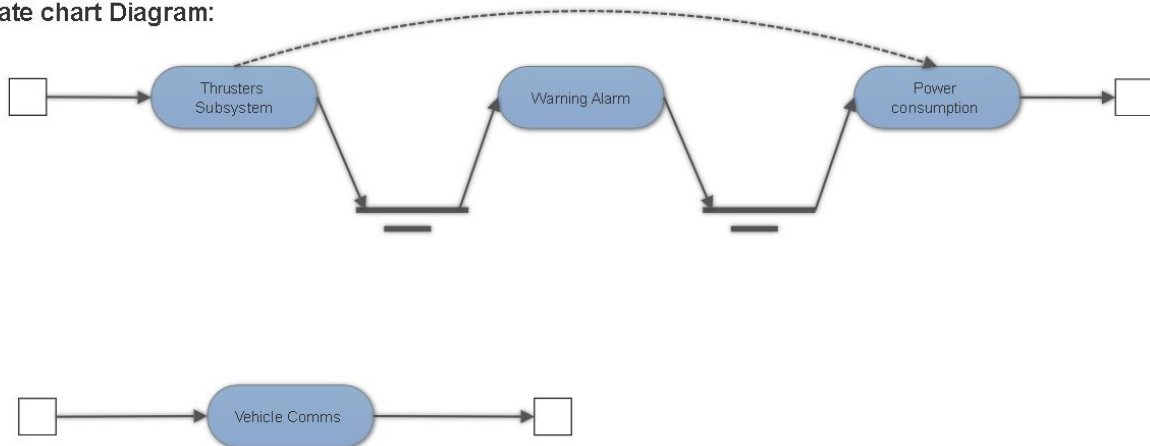
This use case diagram visualizes the interactions between the Earth station, the vehicle, and the satellite. Here, we see that the system objectives are divided into four sections: (i) warning alarm, (ii) solar panel control, both of which are contained in (iii) satellite coms, and (iv) vehicle comms. Each section contains their own objectives, which are linked to the other actors.



***(Figure 2: Functional Decomposition)***

This diagram demonstrates how the functions are decomposed from the Earth station. We see that the satellite management and control system breaks down into the main parts of the satellite comms and vehicle comms. The specific subsystems associated with these pieces are later decomposed into the individual tasks for each function. The functions separate until what is left is a low level command.

**State chart Diagram:**



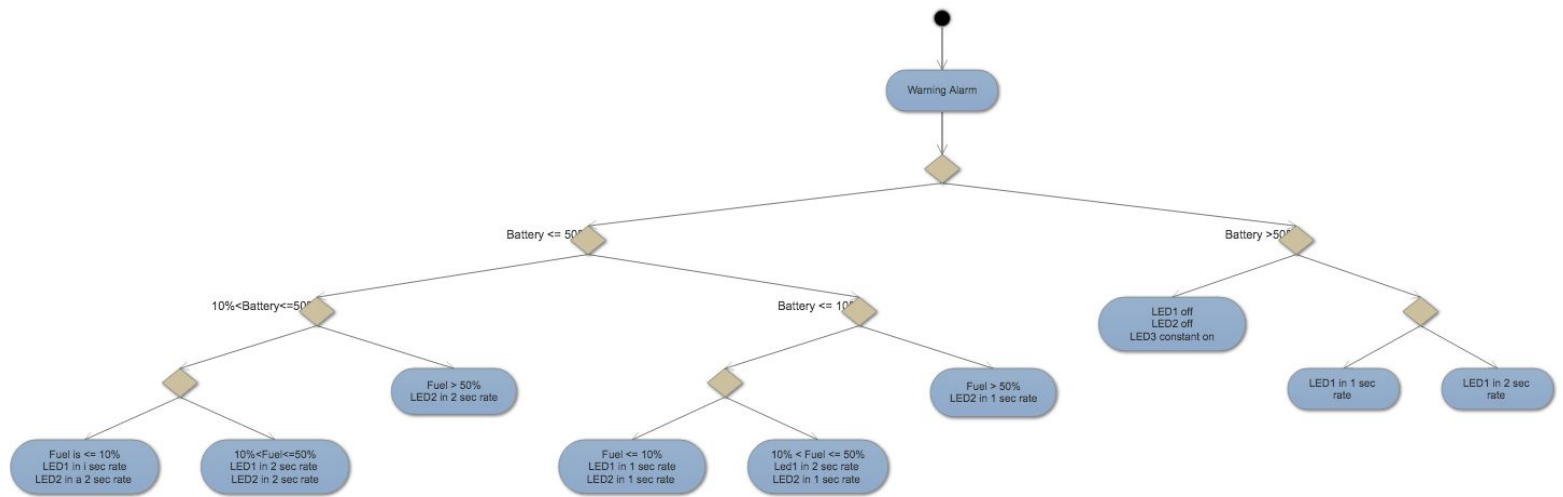
***(Figure 3: State Chart Diagram)***

This state chart diagram shows the dynamic nature of the random number generator that produces the 16 bit number. This number then gets decoded and sent to the thrusters subsystem, which operates the thrusters and updates the fuel level in warning alarm, updating the power consumption, and finally the displaying the output to the user. The user can also send a command to vehicle comms which makes the the vehicle, echoing a response to the terminal.

### 3.3 UML Activity Diagrams

The following section will show the UML Activity Diagrams depicting each of the eight TCBs in action. (note that “Satellite Comms” is reduced to an activity within the “Thruster Subsystem”)

#### 3.3.1. Warning Alarm

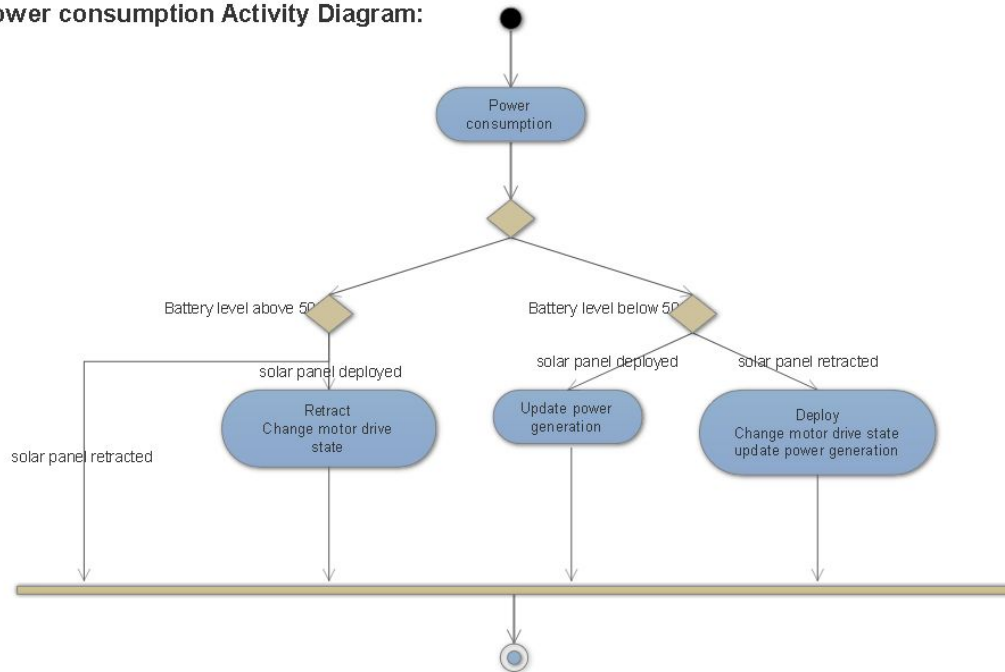


This figure demonstrates the decision points that the warning alarm function encounters.

Namely, by examining the battery and fuel level, it turns on the appropriate led based on the state of these variables.

### 3.3.2. Power Subsystem

Power consumption Activity Diagram:



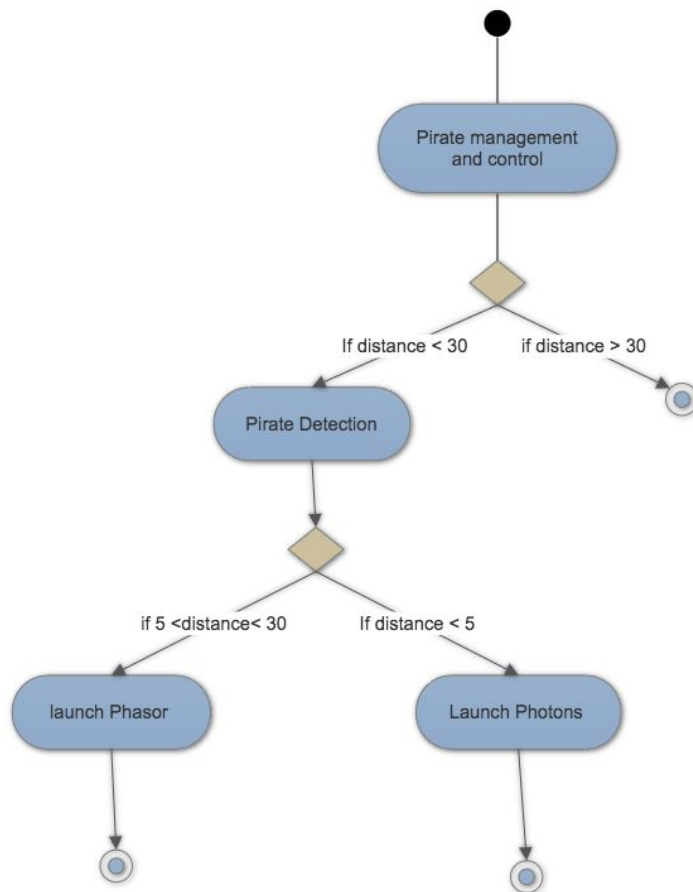
This diagram shows how power consumption and power generation are updated. Currently, the update is simulated using the conditions specified in section 2.2.2.

### 3.3.3. Thruster Subsystem



This activity diagram depicts how the thruster command is parsed. It shows satellite comms sending a 16 bit number, which has a direct impact on the fuel level, which, if it reaches less than 10%, will cause the system to terminate; otherwise, execution proceeds as usual.

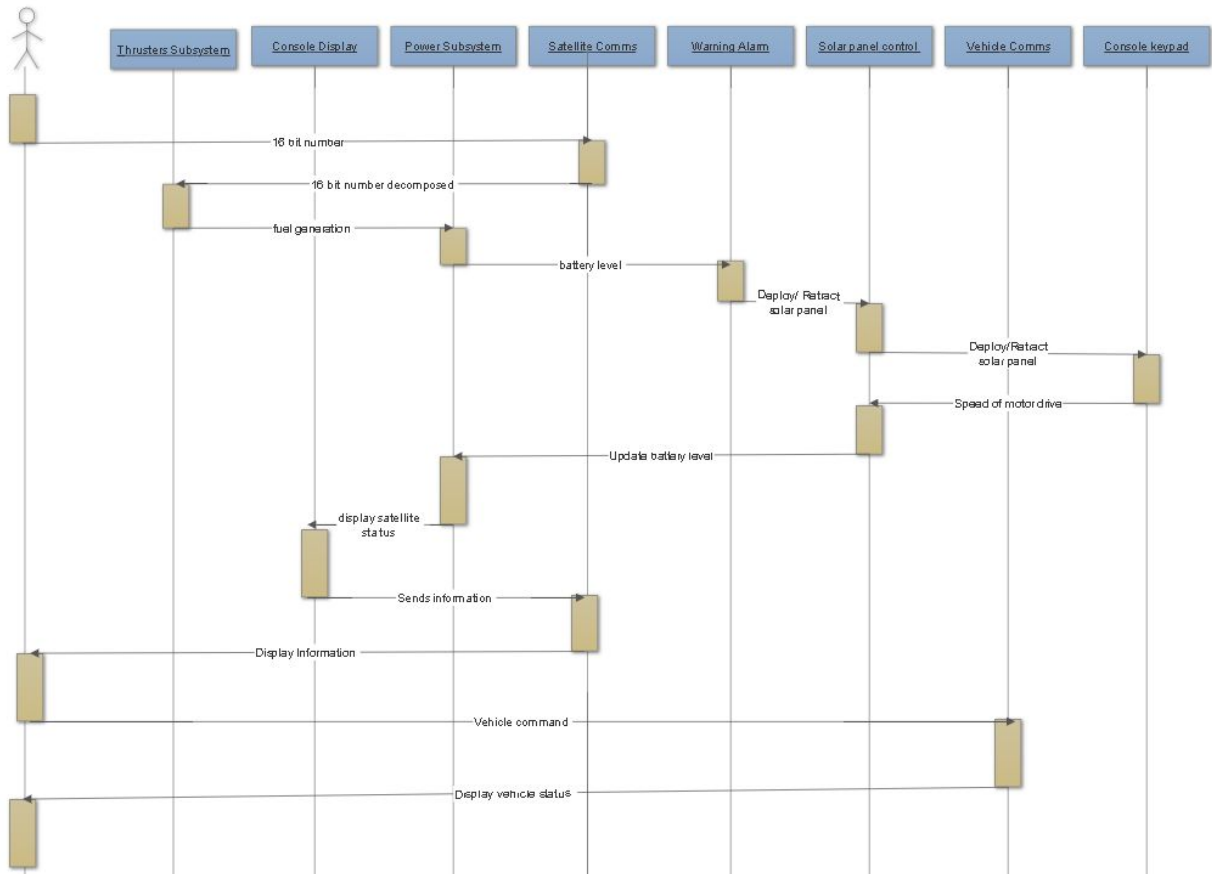
### 3.3.4 Pirate Detection/Management



This activity diagram depicts the process by which pirates are detected and dealt with. If a pirate is within 30 meters of the vehicle, it will either launch its phasors (if the distance is between 5 and 30 meters), or its photons (if the distance is less than 5 meters).

### 3.4 UML Sequence Diagram

Sequence Diagram:



From this diagram, we can see that the flow of control between the TCBs passes between each other via status checks, conditionally flowing from one TCB to another based on variables such as the “16-bit” command, “fuel level”, etc.



### 3.5 Function Prototypes for Tasks

The following function prototypes are given for the tasks defined for the application:

- *void powerSubsystem(void \*powerStruct);*
  - Controls the power of the satellite.
- *void satelliteComs(void \*satStruct);*
  - Establishes the comms link between the satellite status and the earth station.
- *void consoleDisplay(void \*consoleStruct);*
  - Displays satellite status and annunciation on the console panel of the satellite.
- *void thrusterSubsystem(void \*thrustStruct);*
  - Controls the magnitude, duration, direction of the thruster, and the fuel usage.
- *void warningAlarm(void \*warnStruct);*
  - Displays the LEDs on the console which warn when the fuel and/or battery is low.
- *void solarPanelControl(void \*solarStruct);*
  - Controls the solar panel state, flags on whether solar panel has been retracted or deployed, and flags on whether to increase or decrease the motor drive to activate solar panel
- *void keyboardConsole(void \*keyboardStruct);*
  - Controls the flags on increasing or decreasing the motor drive
- *void vehicleComms(void \*vehicleStruct);*
  - Controls command from the user and response from the vehicle
- *void commandParser(void \*cmdStruct);*
  - Manages web browser interface and messaging protocol.
- *void imageCapture(void \*imageStruct);*
  - Captures frequencies and stores them into a 16 length buffer
- *void transportDistance(void \*transportStruct);*
  - Captures frequencies and stores them into an 8 length buffer
- *void pirateDetection(void \*pdStruct);*
  - Detects any pirates near the satellite and calls pirateManagement task
- *void pirateManagement(void \*pmStruct);*
  - Activates photon and phasor functionality based on pirate proximity

The pointers passed into these functions are structs which contain shared variables that the satellite system.

### 3.6 Data Structs Defined for Application

The following struct definitions define all the shared variables within this application.

```
typedef struct powerSubsystemData {  
    bool *solarPanelStatePtr;  
    bool *solarPanelDeployPtr;  
    bool *solarPanelRetractPtr;  
    unsigned int **batteryLvlPtr;  
    unsigned short *pConsumePtr;  
    unsigned short *pGeneratePtr;  
} powerData;
```

(The power system struct tracks pointers to the solar panel state, battery level, power consumed and power generated by the solar panels when deployed).

```
typedef struct thrusterSubsystemData {  
    unsigned int *thrusterCommandPtr;  
    unsigned short *fuelLvlPtr;  
} thrustData;
```

(The thruster system struct tracks pointers to the thruster command from the earth station and fuel level).

```
typedef struct satelliteComsData {  
    bool *fuelLowPtr;  
    bool *batteryLowPtr;  
    bool *solarPanelStatePtr;  
    unsigned int **batteryLvlPtr;  
    unsigned short *fuelLvlPtr;  
    unsigned short *pConsumePtr;  
    unsigned short *pGeneratePtr;  
    unsigned int *thrusterCommandPtr;  
    char *commandPtr;  
    char *responsePtr;  
} satData;
```

(The satellite communication system struct tracks pointers to the satellite status and thruster command. Also, it tracks vehicle response and commands. This allows information to be shared with the earth station).

```
typedef struct consoleDisplayData {
    bool *fuelLowPtr;
    bool *batteryLowPtr;
    bool *solarPanelStatePtr;
    unsigned int **batteryLvlPtr;
    unsigned short *fuelLvlPtr;
    unsigned short *pConsumePtr;
    unsigned short *pGeneratePtr;
} consoleData;
```

(The console display struct tracks pointers to the satellite status and annunciation (fuel low and battery low)).

```
typedef struct warningAlarmData {
    bool *fuelLowPtr;
    bool *batteryLowPtr;
    unsigned int **batteryLvlPtr;
    unsigned short *fuelLvlPtr;
} warnData;
```

(The warning alarm struct tracks pointers to annunciation (fuel low and battery low), fuel level and battery level).

```
typedef struct solarPanelControlData {
    bool *solarPanelStatePtr;
    bool *solarPanelDeployPtr;
    bool *solarPanelRetractPtr;
    bool *motorIncPtr;
    bool *motorDecPtr;
} solarData;
```

(The solar panel control struct tracks pointers to the solar panel state, whether the solar panel has been successfully deployed, retracted, and whether to speed up or slow down motor speed for deployment).

```
typedef struct keyboardConsoleData {
    bool *motorIncPtr;
    bool *motorDecPtr;
} keyboardData;
```

( keyboardConsole points to the speed of the motor. These bools are set by user input).

```
typedef struct vehicleCommsData {
    char *commandPtr;
    char *responsePtr;
} vehicleData;
```

(vehicleComs points to command and response, which are stored by user input and the vehicle respectively).

```
typedef struct commandParserData {
    unsigned int *thrusterCommandPtr;
    char *received;
    char *transmit;
    bool *commandOnPtr;
    char *ack;
    bool *displayPtr;
} cmdData;
```

(commandParser holds pointers to both raw and processed user input, a string to signal acknowledgement/errors, and booleans to indicate that certain tasks should run/do something).

```
typedef struct transportDistanceData {
    unsigned int *distancePtr;
} transportData;
```

(transportDistance holds a pointer to the 8 length buffer that stores the distances of the transport vehicle)

```
typedef struct imageCaptureData {
    int *processImagePtr;
} imageData;
```

(imageCapture holds a pointer to the 16 length buffer that stores the frequencies for the image)

```
typedef struct pirateDetectData {
    bool *pirateDetectedPtr;
    unsigned int *pirateDistancePtr;
} PirateData;
```

(pirateDetection holds pointers to whether a pirate is detected (under 100 units) and the proximity of the pirate).

```
typedef struct pirateManageData {
    bool *pirateDetectedPtr;
    unsigned int *pirateDistancePtr;
} mPirateData;
```

(pirateManagement also holds pointers to whether a pirate is detected (under 100 units) and the proximity of the pirate, processing its distance).

### 3.7 Pseudocode

The following sections provide pseudocode implementations of the 11 TCB functions. For a complete implementation in C, see the attached “a5.zip” file. Although the pseudocode for the PWM/ADC interfaces have been included in the appropriate functions, the actual implementation can be found in “adc\_utils.\*” and “pwm\_utils.\*”

### 3.7.1 WarningAlarm Pseudocode

The warning alarm code can be characterized by two sections: (i) set state of battery low/ fuel low, and (ii) update leds.

*warningAlarm:*

```
    open leds
    check if opened successfully
```

(i) Set state of battery and fuel low

```
    int battery region = LOW, MED or HIGH
    int fuel region = LOW, MED or HIGH
    bool batteryLow = (battery region == LOW);
    bool fuelLow = (fuel region == LOW);
```

(ii) Update LEDs

```
    if battery and fuel are HIGH
        turn on led 3
        turn off other leds
    else // (either battery or fuel is not HIGH)
        turn off led 3
        if battery is at MED
            if (GLOBALCOUNTER - prev) % 2 seconds
                flip state of led 2
        else if battery is LOW
            if (GLOBALCOUNTER - prev) % 1 seconds
                flip state of led 2
        else //(battery is HIGH)
            turn led 2 off
```

*Note: The same steps from above are repeated for fuel level, but led 1's state is flipped instead. Also, "prev" gets the value of GLOBALCOUNTER during the start of a minor cycle and holds that value until the next minor cycle.*

### Design Decisions:

1. Splitting warning alarm into these two sections helped isolated functionality of pointers and the leds, allowing for greater modularization and readability.
2. Since led state was based off of the GLOBALCOUNTER, i.e., the system's time base, the code keeps track of static variables to save the previous state of the led, allowing for more consistent flashing of the leds.

### **3.7.2 SatelliteComs Pseudocode**

SatelliteComs is split into sections (i) randomly generate thruster command (ii) print satellite status, annunciation, and the land-based vehicle response.

*satelliteComs:*

Store all shared variables locally;

(i) Randomly generate thruster command

thrusterCommand = call random number generator;  
mask third and fourth bits of thrusterCommand to 0

(ii) Print satellite status, annunciation, and the landbased vehicle response.

open the earth display terminal  
create text friendly descriptors for solar panel state, fuelLow, and battLow  
dprintf(satellite status);  
dprintf(annunciation);  
dprintf(vehicle response);

### Design Decisions:

- *Section (i):*
  1. The bits are randomly generated as per the requirements since the bidirectional link between the satellite and earth isn't implemented yet.
  2. The mask needs to happen since we only have four directions, left, right, up and down that we've assigned 0000, 0001, 0010, 0011 respectively.
- *Section (ii):*
  1. The text friendly descriptors like "retracted" rather than "0" were chosen for readability.
  2. The printing of satellite status, annunciation and vehicle response were to demonstrate the bidirectional communication of the satellite and Earth station.

### **3.7.3 ConsoleDisplay Pseudocode**

Console Display reads stdin from the earth terminal to switch modes from satellite status to annunciation.

*consoleDisplay:*

```
Store all shared variables locally;
read and store input character;
if (c == SATELLITESTATUS)
    terminalComs(satellite status info)
else if (c == ANNUNCIATION)
    terminalComs(annunciation info)
else
    if (input character is a different task command)
        ungetc(character, stdin)
```



### Design Decisions:

1. The reading of character functionality is specific to each function, in this case, consoleDisplay, which increases modularity.
2. The string information passed to terminalComs is designed to simulate sending info to the satellite terminal, which more closely models expected system behavior.

### **3.7.4 Power Subsystem Pseudocode**

Power Subsystem manages the battery level by reading from the ADC and updating the Power Consumption, Power Generation, and Solar Panel variables.

*powerSubsystem:*

```
init local vars for:
    batteryLvl, powerConsumption, powerGeneration, solarPanelState, solarPanelDeploy,
    and solarPanelRetract

if (!adc_initialized) {
    initializeADC();
}

init local var nextMeasurement = getADCMeasurement(adc_channel, help_number);
set batteryBuffer[nextMeasurementIndex] = nextMeasurement;
updatePowerConsumption(powerConsumption);
updatePowerGenerationAndSolarPanel(solarPanelState, solarPanelDeploy, solarPanelRetract,
powerGeneration, batteryLvl);
```

*initializeADC():*

```
open file for adc;
configure adc;
close file for adc;
```

*getADCMeasurement(adc\_channel, help\_number):*

```
open file for the adc channel; (using adc_channel and help_number)
init local var value = read_from_file(adc channel);
close file for adc channel;
return value;
```

*updatePowerConsumption(powerConsumption):*

```
if (was reversed condition) { // powerConsumption was > 10
    if (not currently in reversed condition) { // powerConsumption is < 5
        flip(condition);
    }
} else { // powerConsumption was <= 10
    if (not currently in unreversed condition) { // powerConsumption is > 10
        flip(condition);
    }
}
if (even function call) {
    powerConsumption = (reversed_condition) -> powerConsumption - 2
                        : (else) powerConsumption + 2;
} else { // odd function call
    powerConsumption = (reversed_condition) -> powerConsumption + 1
                        : (else) powerConsumption - 1;
}
```

*updatePowerGenerationAndSolarPanel(solarPanelState, solarPanelDeploy, solarPanelRetract, powerGeneration, batteryLvl):*

```
if(solar panel state ON) {
    flip solarPanelDeploy;
    flip solarPanelRetract;
    if (batteryBuffer[currentMeasurementIndex] > 95) {
        set solar panel state to OFF;
        Zero out powerGeneration;
    } else {
        updatePowerGeneration(powerGeneration, powerConsumption);
    }
} else { // solar panel state OFF
    if (batteryBuffer[currentMeasurementIndex] <= 10) {
        deploy solar panel;
    }
}
return true if solar panel state ON, and false otherwise;
```

*updatePowerGeneration(powerGeneration, powerConsumption):*

```
if (batteryBuffer[currentMeasurementIndex] <= 95) {  
    if (batteryBuffer[currentMeasurementIndex] <= 50) {  
        powerGeneration = (even_call) -> powerGeneration + 2  
                                : (else) powerGeneration + 1;  
    } else { // batteryBuffer[currentMeasurementIndex] > 50  
        powerGeneration = (even_call) -> powerGeneration + 2  
                                : (else) powerGeneration;  
    }  
}
```

#### Design Decisions:

1. Separate functions for initializing/retrieving ADC measurements allows for future ADC extensions without altering the current interface.
2. Parameterizing “adc\_channel” and “help\_number” increases portability of ADC code to accommodate different initializations/channels.
3. Updating power generation and solar panel variables in the same function increases readability and facilitates efficient data access since these variables are localized.
4. Utilizing an array for the circular buffer allows for constant time access and modification of all 16 measurements.
5. Separate branching for reverse condition and call number in powerConsumption minimizes the amount of nested branching, increasing readability.
6. Zeroing out powerGeneration upon solar panel state OFF in updatePowerGenerationAndSolarPanel helps prevent negative power generation, which is not defined within the specification.

### 3.7.5 Thruster Subsystem Psuedocode

The Thruster Subsystem manages the fuel level and updates the state of the PWM according to the latest thruster command.

*thrusterSubsystem:*

```
init local vars for thrusterCommand and fuelLvl;
init preciseFuelCost;
parsedCommand = parseCommands(thrusterCommand);
preciseFuelCost = preciseFuelCost + getFuelCost(parsedCommand);
if (fuelLvl = 0) {
    terminate_the_program();
} else {
    fuelLvl = fuelLvl - (coerce_to_int) preciseFuelCost;
}
preciseFuelCost = preciseFuelCost - (coerce_to_int) preciseFuelCost;
if (!pwm_initialized) {
    initializePWM();
    set pwm_initialized = true;
}
init local var duty = parsedCommand.magnitude / parsedCommand.duration;
init local var period = parsedCommand.magnitude;
setPWMProperty (pin, "period", period, help_number);
setPWMProperty (pin, "duty", duty, help_number);
```

*parseCommands(thrusterCommand):*

```
duration = mask_out_bits_8_to_1(thrusterCommand);
magnitude = duration = mask_out_bits_16_to_9_and_4_to_1(thrusterCommand);
thruster_duration = mask_out_bits_16_to_5(thrusterCommand);
return { duration, magnitude, thruster_duration };
```

*getFuelCost(parsedCommand):*

```
cost = 0.0001284522 * parsedCommand.magnitude * parsedCommand.duration;
return cost;
```

*initializePWM():*

```
open file for pwm;
configure pwm;
close file for pwm;
```

*setPWMProperty(pin, property, property\_value, help\_number)::*

open file for property of pwm (using help\_number);  
overwrite current value of property to property\_value;  
close file for property of pwm;

Design Decisions:

1. Having a separate “preciseFuelCost” reduces the effects of truncation when returning fuel costs as integers since the precise decimal representation is retained between calls.
2. Having the constant “0.0001284522” (i.e. the ratio  $100 / (6 \text{ months} * 0.05 \text{ magnitude})$ ) as a decimal allows for the scaling of fuel costs in constant time (as opposed to recalculating this constant each time).
3. Using a struct to represent a “parsedCommand” clearly communicates the separate components of a command from satComms, increasing readability.
4. Separate functions for initializing/setting the PWM allows for future PWM extensions without altering the current interface.
5. Parameterizing “pin” and “help\_number” increases the portability of PWM code to accommodate different initializations/pins.

### 3.7.6 solarPanelControl Pseudocode

Solar Panel Control manages the operation of the solar panel motors, updating the duty cycle of the PWM according to the state of the solar panels.

*solarPanelControl:*

```
init duty and period;

if (solar panel is deployed or retracted and its state is ON) {
    set PWM = 0;
} else {
    if (needs to speed up the motor) {
        change duty accordingly;
        cap the duty to not go above period;
    } else if (need to decrease the speed of the motor){
        change duty accordingly;
        cap duty to not go below zero;
    }
}
```

#### Design Decisions

1. Splitting the code into two cases (whether solar panel is deployed or not) helps with readability, and allows for logical separation of PWM updates.

### 3.7.7 commandParser Pseudocode

The commandParser is tasked with interpreting commands from the earth terminal and enabling the appropriate functionality for the appropriate tasks to be executed in response to user input.

*commandParser:*

```
init local variables to store shared variables in cmdStruct;
    // received == user input, ack == acknowledgement string, transmit == optional
    // payload string, display == add console display to Task Queue? ,
    // commandOn == add commandParser to the TaskQueue?
init command = first_character_of(received);
init payload = every_character_after_first_character_of(received);
if (command == THRUSTER_COMMAND) {
    if (payload is within the valid range for a thruster command) { // command in (0, 65536)
        first_character_of(ack) = 'A';
        ProcessThrusterCommand (toInteger(payload)); // prep command for thrusterSubsystem
    } else {
        first_character_of(ack) = 'E';
    }
    transmit = ' '; // no payload
    third_character_of(ack) = THRUSTER_COMMAND;
} else if (command == MEASURE_COMMAND) {
    if (payload is a valid measurement request) { // measurement request in set of valid requests
        first_character_of(ack) = 'A';
        transmit = first_character_of(payload); // e.g. 'B' for 'batteryLevel'
    } else {
        first_character_of(ack) = 'E';
        transmit = 'E'; // bad measure request
    }
    third_character_of(ack) = MEASURE_COMMAND;
} else if (command == START_COMMAND) {
    if (AddMeasurementTasks()) { // Successfully added measurement tasks
        first_character_of(ack) = 'A';
    } else { // Error in removing measurement tasks
        first_character_of(ack) = 'E';
    }
    transmit = ' '; // no payload
    third_character_of(ack) = START_COMMAND;
} else if (command == STOP_COMMAND) {
    if (RemoveMeasurementTasks()) { // Successfully removed measurement tasks
```

```

        first_character_of(ack) = 'A';
    } else { // Error in removing measurement tasks
        first_character_of(ack) = 'E';
    }
    transmit = 'Q'; // no measurements!
    third_character_of(ack) = STOP_COMMAND;
} else if (command == DISPLAY_COMMAND) {
    first_character_of(ack) = 'A';
    third_character_of(ack) = DISPLAY_COMMAND;
    transmit = ' '; // no payload
    flip (display); // if true -> false, and vice-versa
    if (display) { // display mode ON
        if (consoleDisplayTCB is not in the TaskQueue) {
            Add consoleDisplayTCB to the TaskQueue;
        }
    } else { // display mode OFF
        if (consoleDisplayTCB is in the TaskQueue) {
            clear Display;
            Remove consoleDisplayTCB from the TaskQueue;
        }
    }
} else { // unrecognized command
    first_character_of(ack) = 'E';
    third_character_of(ack) = BAD_COMMAND;
    transmit = ' '; // no payload
}
commandOn = false;

```

*AddMeasurementTasks():*

```

if (!enableSignals()) { // unsuccessfully enabled data interrupts
    return false;
}
if (!not_paused) { // see pseudocode for 'main.c'
    Add all measurement tasks to TaskQueue; // (e.g. thrusterSubsystemTCB)
    set vehicleCommsInTheQueue = true; // is vehicleComms is in the TaskQueue
}
not_paused = true; // see pseudocode for 'main.c'
return true;

```



*RemoveMeasurementTasks()*:

```
if (!disableSignals()) { // unsuccessfully disabled data interrupts
    return false;
}
if (not_paused) { // see pseudocode for 'main.c'
    Remove all measurement tasks to TaskQueue; // (e.g. thrusterSubsystemTCB)
    set vehicleCommsInTheQueue = false; // is vehicleComms is not in the TaskQueue
}
return true;
```

### Design Decisions

1. Factoring out adding/removing tasks into separate functions (i.e. AddMeasurementTasks and RemoveMeasurementTasks) increases modularity by allowing for what START\_COMMAND and STOP\_COMMAND does to change in the future.
2. Adding/removing consoleDisplayTCB within commandParser (rather than in the main scheduler) allows for displaying even if STOP\_COMMAND was previously issued without creating added complexity within the main scheduler, increasing code clarity.
3. Setting the acknowledgement strings (to be printed as-is) within commandParser (rather than in consoleDisplay) localizes command status to commandParser, abstracting other functions from how/whether a command succeeded or not, increasing code concision.
4. Setting the measurement payload as a “confirmation” rather than the actual values from the respective TCB limits the external dependencies within commandParser, and allows for consoleDisplay to utilize this confirmation to get the actual values with minimal changes to existing code, which substantially increases code independence.

### 3.7.8 keyboardConsole Pseudocode

The keyboardConsole is split into (i) input character is not a solarPanel command and (ii) input character is a solarPanel command.

(i) input character is not a solarPanel command

```
read input and store character in variable c;  
if (c is not solarPanel command) {  
    motorInc = false;  
    motorDec = false;  
}
```

(ii) input character is a solarPanel command

```
else {  
    if (c == increase motor speed) {  
        motorInc = true;  
        motorDec = false;  
    }  
    else if (c == decrease motor speed) {  
        motorInc = false;  
        motorDec = true;  
    }  
}
```

#### Design Decisions

1. Splitting keyboard console into these two sections increases readability, and allows for faster evaluation if c is not a valid solarPanel command.

### 3.7.8 vehicleComs Pseudocode

Vehicle Comms manages serial communication with the Earth terminal and the land-based vehicle.

#### (i) Read character input

```
read input and store character;  
if (valid vehicleCommand) {  
    store it;  
} else {  
    return it to the buffer or discard it;  
}
```

#### (ii) Initiate pipe communication

```
open pipe to vehicle;  
send command to vehicle;  
read command from vehicle;  
write response back;  
read response from satellite;
```

#### Design Decisions:

1. The pipe communication was placed within the vehicleComs task to promote simple information exchange by isolating this functionality within the same module.

### 3.7.9 Dynamic Task Queue

The Dynamic Task Queue was implemented as a first-class ADT (“Abstract Data Type”). That is, a construct with data members and a set of operations. The following struct definition represents the data members:

```
typedef struct taskqueue {  
    TCB *head, *tail; // Head/Tail of the task queue  
    unsigned int num_tasks; // Current number of tasks in the queue  
} *TaskQueue;
```

As one can see, the Dynamic Task Queue has access to the beginning (head) and end (tail) of the queue, as well as a running count of the total number of elements in it.

The following function prototypes represent the set of operations defined for a Dynamic Task Queue:

```
// Initializes queue to an empty Dynamic Task Queue  
void InitializeTaskQueue(TaskQueue queue);  
  
// Adds a TCB to the end of queue  
int AppendTCB(TaskQueue queue, TCB *node);  
  
// Adds a TCB to the beginning of queue  
int PushTCB(TaskQueue queue, TCB *node);  
  
// Removes a TCB from queue  
TCB* RemoveTCB(TaskQueue queue, TCB *node);  
  
// Removes the last TCB from the queue  
TCB* SliceTCB(TaskQueue queue);  
  
// Removes the first TCB from the queue  
TCB* PopTCB(TaskQueue queue);  
  
// Report the number of TCBs currently in the queue  
unsigned int NumTasksInTaskQueue(TaskQueue queue);
```

The pseudocode implementations for AppendTCB, RemoveTCB, SliceTCB, and PopTCB are provided below. Since PushTCB was not used in this project, and NumTasksInTaskQueue is a simple accessor, the pseudocode has been omitted for space.

*AppendTCB(TaskQueue queue, TCB \*node):*

```

init num_tasks = queue.num_tasks;
if (num_tasks == 0) {
    set queue.head & queue.tail = node;
    set node.next & node.prev = NULL;
} else {
    set node.next = NULL;
    set node.prev = queue.tail;
    set queue.tail.next = node;
    set queue.tail = node;
}
queue.num_tasks = queue.num_tasks + 1;

```

*RemoveTCB(TaskQueue queue, TCB \*node):*

```

init TCB *curr = queue.head;
while (curr != node) {
    set curr = curr.next;
}
if (curr == queue.head) { // node is located at the front of queue
    return PopTCB(queue);
}
if (curr == queue.tail) { // node is located at the back of queue
    return SliceTCB(queue);
}
// node is located in the middle of queue
set curr.prev.next = curr.next; // skip over curr.next
set curr.next.prev = curr.prev; // reset curr's previous pointer
set queue.num_tasks = queue.num_tasks - 1;
return curr;

```

*PopTCB(TaskQueue queue):*

```

init TCB *old_head = queue.head; // save the old head
queue.head = queue.head.next; // skip over the old head
queue.num_tasks = queue.num_tasks - 1;

```

*SliceTCB(TaskQueue queue):*

```
init TCB *old_tail = queue.head; // save the old tail
queue.tail = queue.tail.prev; // skip over the old tail
queue.num_tasks = queue.num_tasks - 1;
```

### Design Decisions:

1. Implementing the Dynamic Task Queue as a first-class ADT allows for greater flexibility of any client using the Dynamic Task Queue (in this case, the scheduler) by abstracting the “next/prev” pointers of TCBs completely.
2. Having an “InitializeTaskQueue” method allows for a standardized way of creating static Task Queues, and greatly simplifies the associated initialization within the startup task.
3. Maintaining pointers to the head/tail of the task queue simplifies the Pop/Slice/Push/Append methods, as well as increasing access speeds for important sections of the task queue.
4. Maintaining a running count of the number of tasks allows for greater control/diagnostic information about the task queue.
5. Separating different cases for removing from (Push/Slice/Remove) and adding to (Append/Push) different sections of the queue gives clients specialized functions for manipulating the task queue, increasing efficiency and modularity.

### 3.7.10 Startup Task Pseudocode

The Startup Task declares all shared variables, default initializes them, and starts the System Time Base.

*startup:*

```
declare all shared global variables;  
Initialize();  
ActivateTimeBase();
```

*Initialize():*

```
set initial values for all shared global variables;  
initialize the Dynamic Task Queue containing warningAlarm and satelliteComs;  
initialize PWM and ADC;
```

*ActivateTimeBase():*

```
set GlobalCounter = 0;
```

*signalHandler(int signal\_num):* // Defined but used externally (see commandParser)

```
if (signal_num == USER_DEFINED_SIGNAL) {  
    if (attempting to open ADC) { // Handles connections with arbitrary ADC channels  
        set stable = true; // used in readADC() of adc_utils.c  
    }  
    if (attempting to overdrive the solar panel PWM) { // Prevents motor overdrive  
        set endOfTravel = true; // used in solarPanelControlTCB()  
    }  
    if (executed transport distance) { // handles transportDistance docking  
        print ("Received T");  
        call vehicleComs();  
        print ("Received D");  
        call vehicleComs();  
    }  
}
```

```

if (signal_num == CTRL_C) {
    set addTransportDistanceToQueue = true; // used in main scheduler
}
if (!adc_initialized) {
    adc_initialized = true; // used in readADC() of adc_utils.c
    print ("Successfully connected ADC");
}
if (!pwm_initialized) {
    pwm_initialized = true; // used in setPWMProperty() of pwm_utils.c
    print ("Successfully connected PWM");
}

```

### Design Decisions:

1. Separating the declaration and initialization of global variable allows for easily modifying default initialization to reflect future changes to the specification.
2. Separating the initialization of the GlobalCounter supports future modifications to the current system timing scheme.
3. Defining the signalHandler within startup abstracts commandParser from the interactions of different TCBs with software interrupts, allowing for changes to the ISRs without breaking commandParser.
4. Assuming pwm/adc have been properly initialized outside of the signalHandler shortens the ISR, allowing for minimal computation within the interrupt.
5. Performing function calls and printing within the context of transportDistance simplifies the control flow between transport distance, vehicle comms, and the scheduler.
6. Overloading CTRL\_C (typically used to terminate a program) as a signal for transportDistance allows for a more realistic, “push-button” - like, interface (a “push-button”, which can be easily overridden upon sending the STOP command to commandParser.



### 3.7.11 Scheduler Pseudocode

The scheduler coordinates the execution of the TCBs, dispatching tasks from the Task Queue, adding tasks as necessary, and updating the System Time Base.

*main:*

```
Startup();
declare TCB *aTCBPtr;
init static append = 0, remove = 1;
while (true) {
    if (not_paused) { // user has entered 'P' through commandParser (or 'S' never inputted)
        if (solarPanelState == ON) {
            if (solarPanelDeployed) { // Charging
                if (batteryTempTCB is not in the TaskQueue) {
                    Add batteryTempTCB to the TaskQueue;
                } else { // Solar Panel switch state
                    if (solarPanelControlTCB is not in the TaskQueue) {
                        Add solarPanelControlTCB and keyBoardConsoleTCB to TaskQueue;
                    }
                }
            }
        } else { // solarPanelState == OFF
            if (!solarPanelRetracted) { // Solar Panel switch state
                if (solarPanelControlTCB is not in the TaskQueue) {
                    Add solarPanelControlTCB and keyBoardConsoleTCB to TaskQueue;
                }
            } else { // Not Charging
                if (batteryTempTCB or solarPanelControlTCB in the TaskQueue) {
                    Remove batteryTempTCB, solarPanelControlTCB, and keyBoardConsoleTCB
                    from the TaskQueue;
                }
            }
        }

        if (transportDistance_should_be_scheduled) {
            if (transportDistanceTCB is not in the TaskQueue) {
                Add transportDistanceTCB to the TaskQueue;
            }
        } else { // transportDistance_should_not_be_scheduled
            if (transportDistanceTCB is in the TaskQueue) {
                Remove transportDistanceTCB from the TaskQueue;
            }
        }
    }
}
```

```

    if (pirateDetected) {
        if (pirateManagementTCB is not in the TaskQueue) {
            Add pirateManagementTCB to the TaskQueue;
        }
    } else { // !pirateDetected
        if (pirateManagementTCB is in the TaskQueue) {
            Remove pirateManagementTCB from the TaskQueue;
        }
    }
}
}
if (commandParserTCB_should_be_scheduled) {
    if (commandParserTCB is not in the TaskQueue) {
        Add commandParserTCB to the TaskQueue;
    }
} else { // commandParserTCB_should_not_be_scheduled
    if (commandParserTCB is in the TaskQueue) {
        Remove commandParaserTCB from the TaskQueue;
    }
}
}
aTCBPtr = get_beginning_of_TaskQueue;
execute(aTCBPtr);
if (a cycle has been completed) {
    if (GlobalCounter % MajorCycle == 0){
        delay with MajorDelay;
    } else{
        delay with MinorDelay;
    }
}
GlobalCounter = GlobalCounter + 1;
add aTCBPtr to the end of TaskQueue;
}

```

*Non-critical TCB functions:*

```

init static start = 0;
if ((GlobalCounter - start) % MajorCycle != 0 ){
    do not run this function;
}
assign start to GlobalCounter;

```

Note: “Non-critical TCB functions” include TCB excluding Warning Alarm, Satellite Comms, Vehicle Comms, Pirate Detection and Management, Image Capture, and Console Display.

### Design Decisions:

1. Utilizing flags to denote whether ad-hoc tasks (e.g. SolarPanelControlTCB) or commandParser should be added or removed allows for logical alteration of the task queue and increased modularity.
2. Removing TCBs from the beginning of the TaskQueue before execution, and adding TCBs to the end of the TaskQueue after execution ensures an expected ordering of task execution, increasing predictability/safety.

### **3.7.12 Image Capture Pseudocode**

Image Capture reads images (ADC voltage), performs a Fast Fourier Transform on the raw data, and stores the frequencies in a buffer of size 16.

*imageCapture:*

```
initialize ADC
for loop 16 times:
    for loop 256 times:
        read raw ADC value
        store into real buffer
    fft(real, samples)
    find index of max magnitude
    use bin wang's algorithm
    store final frequency into 16 length buffer
```

### Design Decisions:

1. We decided to calculate all 16 frequencies in one task cycle. This would be much easier to store all frequencies when the user wishes to request them, despite taking slightly longer to execute.

### 3.7.12 Transport Distance Pseudocode

Transport distance tracks the distance between the satellite and a vehicle wishing to dock.

*transportDistance:*

- (i)
  - open 9 gpio pins
  - reset the hardware counter
  - set gpio input to 1
  - usleep(DELAY)
  - set gpio input to 0
  
- (ii) *after receiving hardware counter count*
  - read all 7 bits
  - convert 7 bits into int gpioBinary
  - freq = gpioBinary \* time of delay;
  - if freq > 2100 {
    - freq = 2100}
  
- (iii) *after calculating frequency*
  - distance = 2100 - frequency; // inverse relationship
  - cap distance between 100 and 2000
  - store distance into 8 bit buffer

#### Design Decisions

- *Section (i):*
  - 1. Initializing the gpio pins within transportDistance keeps it localized and modular.
- *Section (ii):*
  - 1. Since the frequency values can go above 2100, we cap it at 2100 to prevent overflow.
- *Section (iii):*
  - 1. The distance to frequency linear equation was made to maximize the range of frequencies we can use, which is from 0 to 2000.

### 3.7.12 Pirate Detection Pseudocode

*pirateDetection:*

```
rawDistance = readADC;
pirateDistance = rawDistance / 9;

if (pirateDistance <= 100)
    pirateDetected = true;
else
    pirateDetected = false;
```

#### Design Decisions:

1. Since the ADC ranges from 0 to 1800 mV, we scaled the pirateDistance down by 9 to get better resolution for when the pirates are under 5 units away.

### 3.7.12 Pirate Management Pseudocode

*pirateManagement:*

```
get rawDistance from pirateDetection;
get pirateDetection bool;

if(pirateDistance <= 30){
    get the character from user input

    if (pirateDistance <= 5 && character is 'o'){
        initiate the photon bitch
        turn LED on the sleep for 0.2 sec 5 times
    }
    else if(pirateDistance > 5 && character is 'p'){
        initiate the phasor
        turn yellow LED on for 1 sec
    }
}
```

#### Design Decisions:

1. We added delays to when the photon and phasors are initiated to give time for the external LED to turn on and off.

## 4. TEST PLAN

In these Projects, as with the previous one, software and hardware features were added, which increased the complexity of the complete system. Building off of the strategy from the previous project, each new TCB was tested separately, as well as the added peripheral devices/sensors, if applicable, forming three distinct categories of functions: communication based, hardware dependant, and scheduler based.

The communication based functions include Vehicle Comms, Satellite Comms, Console Display, Keyboard Console, Warning Alarm, and Command Parser. The main strategy used to test these functions was output inspection and print statements of internal values. The hardware dependant functions include Power Subsystem, Thruster Subsystem, Solar Panel Control, Warning Alarm, Pirate Detection, Image Capture, Transport Distance, and Battery Temperature. To test these functions, toy programs were used to test the interface with the external devices, while the internal data management of the shared variables was tested via Google Test C++ Testing Framework (for test code, see files labeled with the “.cc” extension). The scheduler based functions included the Dynamic Task Queue and Startup Task. To test these functions, a set of traditional unit tests using the Google Test C++ Testing Framework was leveraged. Sections 4 and 5 outline the test specification and test cases, which provide greater detail about both the “what” and the “how” of this project’s approach to testing.

With this approach, the robustness of the system was tested under normal and abnormal conditions, time-constraints were validated, and the behavior of the system was verified under the terms of the specification.

## 5. TEST SPECIFICATION

The following table outlines the expected values of the data items within each TCB, outlining the invariants that must be preserved during execution.

TCB	Primary Data	Valid State	Boundary Values
Power Subsystem	1. Power consumption	1. $0 \leq x \leq 12$	1. $x = 0, x = 12$
	2. Power generation	2. $x \geq 0$	2. $x = 0$
	3. Solar panel state	3. TRUE/FALSE	3. N/A
	4. Solar panel deploy	4. TRUE/FALSE	4. N/A
	5. Solar panel retract	5. TRUE/FALSE	5. N/A
	6. Battery level	6. $0 \leq x \leq 1800$	4. $x = 0, x = 1800$
Thruster Subsystem	1. Thruster Command	1. 16-bit int	1. $0 < x < 2^{16} - 1$
	2. Fuel Level	2. $0 \leq x \leq 100$	2. $x = 0, x = 100$
	3. PWM: duty	3. $0 \leq x \leq 16$	3. $x = 0, x = 16$
	4. PWM: period	4. $0 \leq x \leq 16$	4. $x = 0, x = 16$

Satellite Comms	1. Thruster Command 2. Response 3. Transmit 4. Ack 5. Received	1. 16-bit int 2. "A <valid cmd> 3. Defined in satComsParse.h 4. Defined in satComsParse.h 5. Defined in satComsParse.h	1. $0 < x < 2^{16} - 1$ 2. N/A 3. N/A 4. N/A 5. N/A
Warning Subsystem	1. Fuel Level 2. Battery Level 3. Fuel Low 4. Battery Low	1. $0 \leq x \leq 100$ 2. $0 \leq x \leq 100$ 3. TRUE/FALSE 4. TRUE/FALSE	1. $x = 0, x = 100$ 2. $x = 0, x = 100$ 3. N/A 4. N/A
Console Display	1. Fuel Level 2. Battery Level 3. Fuel Low 4. Battery Low 5. Power consumption 6. Power generation	1. $0 \leq x \leq 100$ 2. $0 \leq x \leq 100$ 3. TRUE/FALSE 4. TRUE/FALSE 5. $0 \leq x \leq 12$ 6. $x \geq 0$	1. $x = 0, x = 100$ 2. $x = 0, x = 100$ 3. N/A 4. N/A 5. $x = 0, x = 12$ 6. $x = 0$



VehicleComs	1. Command 2. Response	1. F, B, L, R, D, H 2. A <valid cmd>	N/A
Keyboard Console	1. MotorInc 2. MotorDec	1. TRUE/FALSE 2. TRUE/FALSE	N/A
Solar Panel Control	1. Solar panel state 2. Solar panel deploy 3. Solar panel retract 4. MotorInc 5. MotorDec 6. PWM: duty	1. TRUE/FALSE 2. TRUE/FALSE 3. TRUE/FALSE 4. TRUE/FALSE 5. TRUE/FALSE 6. $0 \leq x \leq 500000$	1. N/A 2. N/A 3. N/A 4. N/A 5. N/A 6. $x = 0, x = 500000$

Command Parser	1. ThrusterCommandPtr 2. Received 3. Transmit 4. CommandOnPtr 5. Ack 6. DisplayPtr	1. 16-bit int 2. Defined in satComsParse.h 3. Defined in satComsParse.h 4. TRUE/FALSE 5. Defined in satComsParse.h 6. TRUE/FALSE	1. $0 < x < 2^{16} - 1$ 2. N/A 3. N/A 4. N/A 5. N/A 6. N/A
Image Capture	1. Process Image	1. Valid address of the 16 bit buffer	N/A
Transport Distance	1. Transport Distance	1. $100 < x < 2000$	1. $100 < x < 2000$
Pirate Detection/ Management	1. Pirate Distance 2. Pirate Detected	1. $0 < x < 200$ 2. TRUE/FALSE	1. $x > 0$ 2. N/A

## 6. TEST CASES

The following table outlines the test cases for task and the subsequent methodology for testing said cases.

TCB	How to Test	Tools/Strategies (if applicable)
Power Subsystem	<p><b>1. Ensuring Power Consumption remains in a valid state.</b> Running the unit tests to compare the expected output with the resulting output of the TCB function, the four conditions in which Power Consumption would update its value is correct: reversed condition vs. non-reversed condition and even vs. odd function call number.</p> <p><b>2. Ensuring Power Generation remains in a valid state.</b> Similarly to Power Consumption, running the unit tests should simulate the five conditions in which Power Generation would change: battery level either below 50, between 50 and 95, or greater than 95, and even vs. odd function call number.</p> <p><b>3. Ensuring the Solar Panel variables remain in valid states.</b> As with both 1 and 2, running unit tests should produce the expected output when solar panels were deployed vs. retracted and the state ON vs. OFF.</p> <p><b>4. Testing the ADC interface</b> Running the toy program for the ADC interface (contained in adc_utils.*) should see the correct basic functions (initialization/measurement) are implemented. This file can be found in adc_test.c within a5.zip.</p>	Google Test C++ Testing Framework, and toy programming

<p>Thruster Subsystem</p>	<p><b>1. Verifying that the thruster command was being properly decoded.</b> Running a unit test should yield all correct permutations of thruster_dirations (up, down, left, right), magnitudes (0-100), and durations (0-255) via nested for-loops.</p> <p><b>2. Verifying that the cost was being properly calculated.</b> Running a unit test, the user should see that known command with a precalculated cost matched the output of the system.</p> <p><b>3. Testing the PWM interface</b> The PWM interface (contained in pwm_utils.*), was tested using a toy program designed to test the basic functions (initialization/property modification). This file can be found in pwm_test.c within a5.zip.</p>	<p>Google Test C++ Testing Framework, and toy programming</p>
<p>Satellite Comms</p>	<p><b>1. Random integer is no greater than maximum for a 16-bit number (<math>2^{16}-1</math>)</b> The user can print the thrusterCommand integer onto the terminal and noted if any of the numbers exceeded <math>2^{16}-1</math>.</p> <p><b>2. Integer's third and fourth bit are masked to 0 correctly</b> The user can print the integer after it's been masked. The user should see that it converted it to binary and see that the third and fourth bit gets 0.</p> <p><b>3. Satellite Status and annunciation are printed correctly.</b> The user should see that the printed values are correctly updated</p> <p><b>4. Vehicle Response and Command are shared correctly with vehicleComs.</b> The user should see that the response and command variables are printed correctly onto the console.</p>	<p>Output inspection &amp; Print Statements</p>

Warning Subsystem	<p><b>1. Battery low and fuel low are true when their values go below 10%.</b> The user can observe whether the fuel low and battery low values change when the levels change.</p> <p><b>2. Leds are flashing correctly based on battery and fuel level.</b> If initialized battery and fuel level to appropriate testing value (above 50, in between 10 and 50, and below 10), the user can observe whether the correct led is turned on, and used a timer + metronome to determine if the led is flashing at a 1 and 2 second rate.</p>	Output inspection (via metronome + stopwatch)
Console Display	<p><b>1. All commands are printed correctly on the satellite terminal</b> The user can observe that all commands are printed onto the pseudocode terminal correctly and were updating.</p> <p><b>2. Switching modes operated as intended</b> The user can see that the mode is printed correctly based on character input. Also, the user should not see any output if an invalid character is received.</p>	Output inspection & Print Statements
Vehicle Coms	<p><b>1. Pipe is communicating as intended</b> The user should see the print statements to trace the character variable from the time it's being written into the pipe, to when it's being read from the "vehicle". Similar print statements allow the user to track the response from the vehicle to when it's read in the main vehicleComs task function.</p> <p><b>2. Reading user input is functioning correctly</b> The user can see the character read from the stdin buffer to check if it's storing it correctly. The user should also see that invalid commands are ignored and that no output is produced.</p>	Output inspection & Print Statements

Keyboard Console	<p><b>1. Reading user input is functioning correctly</b> The user should see the character read from the stdin buffer onto the terminal is correct.</p> <p><b>2. MotorInc and MotorDec update based on user input</b> The user should see that the state of MotorInc and MotorDec are consistent with user input.</p>	Output inspection & Print statements
Command Parser	<p><b>1. Well formed commands were being responded to appropriately</b> First, print out the state of “Ack” upon entering a valid command, and verify that it contains ‘A’ as the first character, and the valid command as its third. For START/STOP, print out the state of the Task Queue before and after execution to ensure the number of tasks and the tasks contained within the Task Queue are correct. For DISPLAY, inspect the output terminal after successive toggles of the DISPLAY. For THRUSTER, print out the state of thruster command and verify that the bitmask occurred with assert statements. Finally, for MEASURE, ensure that the “transmit” and the measurement request in question match.</p> <p><b>2. Invalid commands were being caught and flagged</b> First, ensure that commands (i.e. the first character of user input) that are outside of the defined commands receive an error acknowledgement followed by ‘BAD_COMMAND’. For valid commands with invalid payloads (e.g. for THRUSTER/MEASURE), ensure that the error acknowledgement is still indicated, followed by the valid command.</p>	

Solar Panel Control	<p><b>1. Checking if the state needs to be updated</b> Printed the command given and the associated state, checking if it is the consistent across the solar panel control sub functions.</p> <p><b>2. Checking MotorInc/MotorDec was read and duty was updated accordingly.</b> Read the value from the motor increase and the motor decrease variables to see if the duty needed to increase or decrease. If it needed to increase, we added 5% of the original duty to the duty that we have if needed to decrease we subtracted 5% of the original duty. We tested this by seeing if the PWM duty cycle was changed using an oscilloscope probe.</p> <p><b>3. Testing the PWM interface</b> See “Thruster Subsystem”</p> <p><b>4. Verifying that the caps for the PWM function is being maintained.</b> We made sure that the function does not reach the caps by putting a limit to the duty at the period and at zero. We printed out the duty that was echoed on the system to ensure that it never exceeded this limit.</p>	Toy programming; Function generator, wires, and oscilloscope.
Dynamic Task Queue	<p><b>1. Verifying individual methods within the TaskQueue ADT</b> This was tested utilizing unit tests that did output comparison of expected values and TaskQueue state after each of the operations (AppendTCB, PopTCB, etc) were called with dummy TCB values.</p>	Google Test C++ Testing Framework
Startup Task	<p><b>1. Verifying the correct initial values of the shared variables were being initialized correctly</b> This was tested utilizing unit tests that externed each of the shared variables from the “startup.c” source file, and comparing the expected values of the individual variables with the actual value after both the Initialize() and ActivateTimeBase() functions have been called.</p>	Google Test C++ Testing Framework

Image Capture	<p><b>1. Verifying that the ADC was being correctly read</b> The values were printed to a txt file and imported into excel. The user should see a consistent sine wave of adc values that were sampled.</p> <p><b>2. Verifying that the image frequencies were stored correctly</b> After the images were calculated from the fft, the user can run a for loop through the entire buffer and see the values are both printed correctly, and closely match the frequency from the function generator.</p>	File I/O verification and print statements
Transport Distance	<p><b>1. Verify that the circuit was working</b> The user can use an oscilloscope and probe each output to see if a square wave is being produced at each critical section. The counter value should also be printed correctly and is consistent.</p> <p><b>2. Ensure that the distance stays in bounds.</b> When the user changes the frequency of the square wave, the distance does not exceed 2000 and go below 100.</p>	Print statements Oscilloscope
Pirate Detection/ Management	<p><b>1. Verifying the correct reading from the ADC</b> the values was checked with the voltmeter and made sure that they are equal knowing that the the value is in millivolts from the ADC PIN</p> <p><b>2. Verifying that the ADC conversion is correct</b> divided the output by 9 and read what it is made sure that it is going above the required range to start pirat detection</p> <p><b>3. Making sure that the leds is light correctly and according to the desired</b> checked what is the distance if it was less than 5 or greater than 5 with the correct LED flash</p>	multimeter, print statement

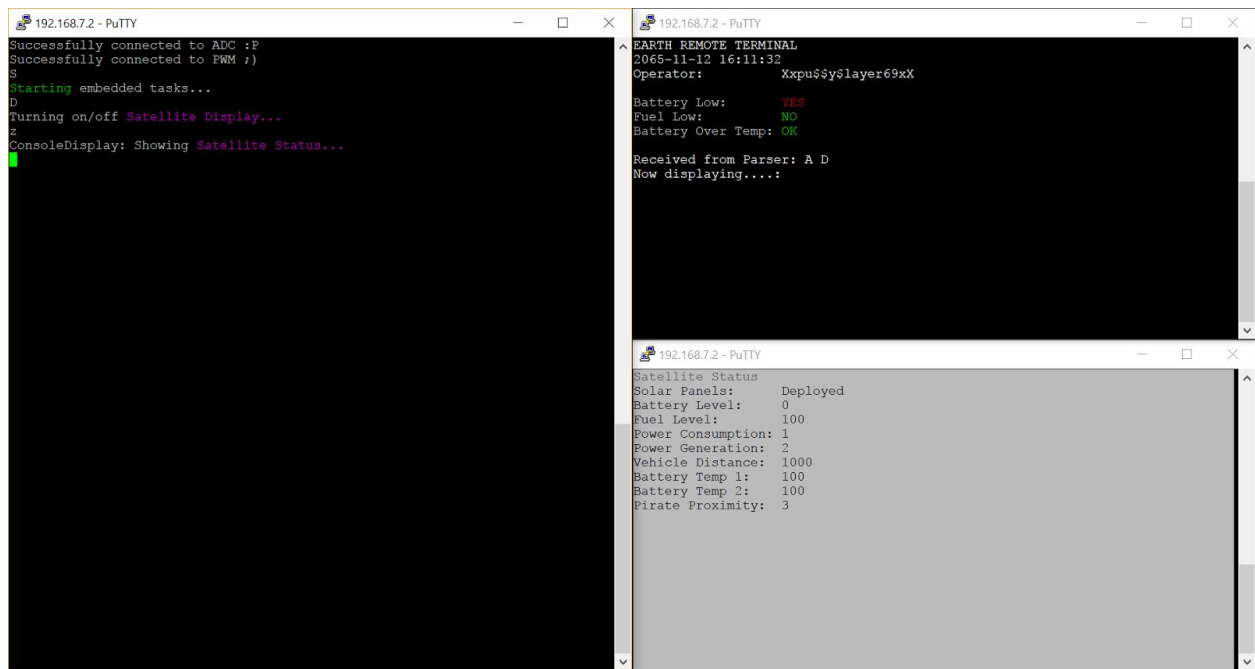


## 7. PRESENTATION AND RESULTS

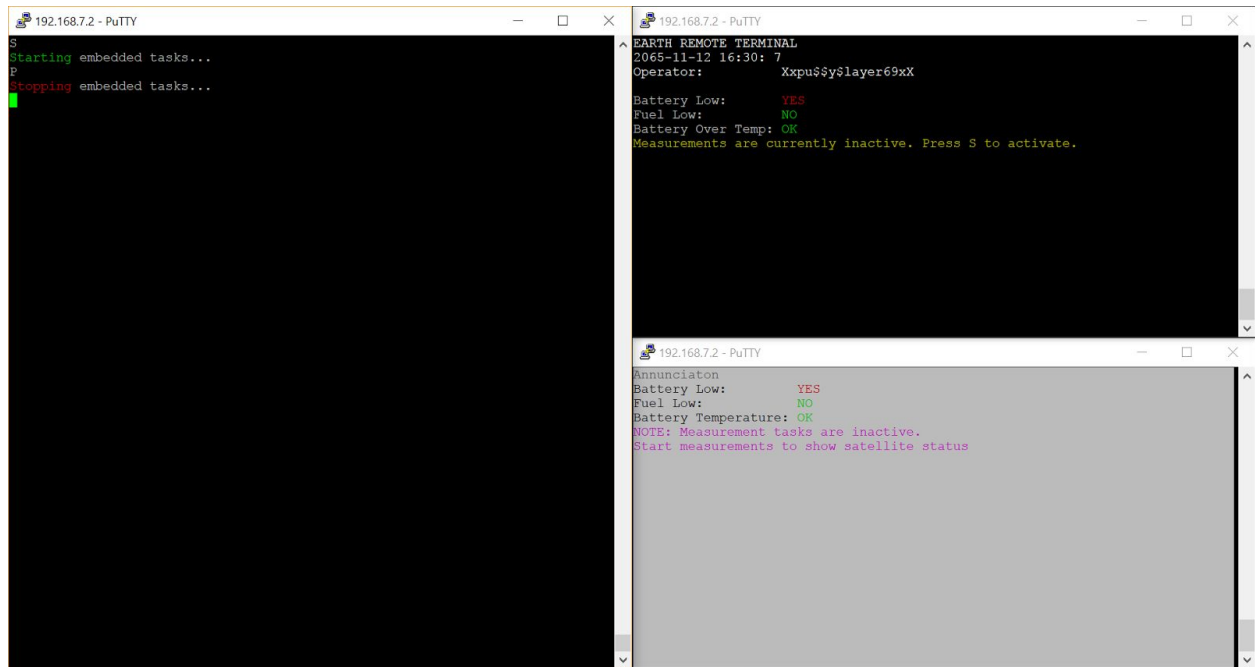
### 7.1 Output of the System

This subsection will present each task result in detail with associated screen captures and images.

#### 7.1.1 Terminal Interface (Start tasks and stop tasks)

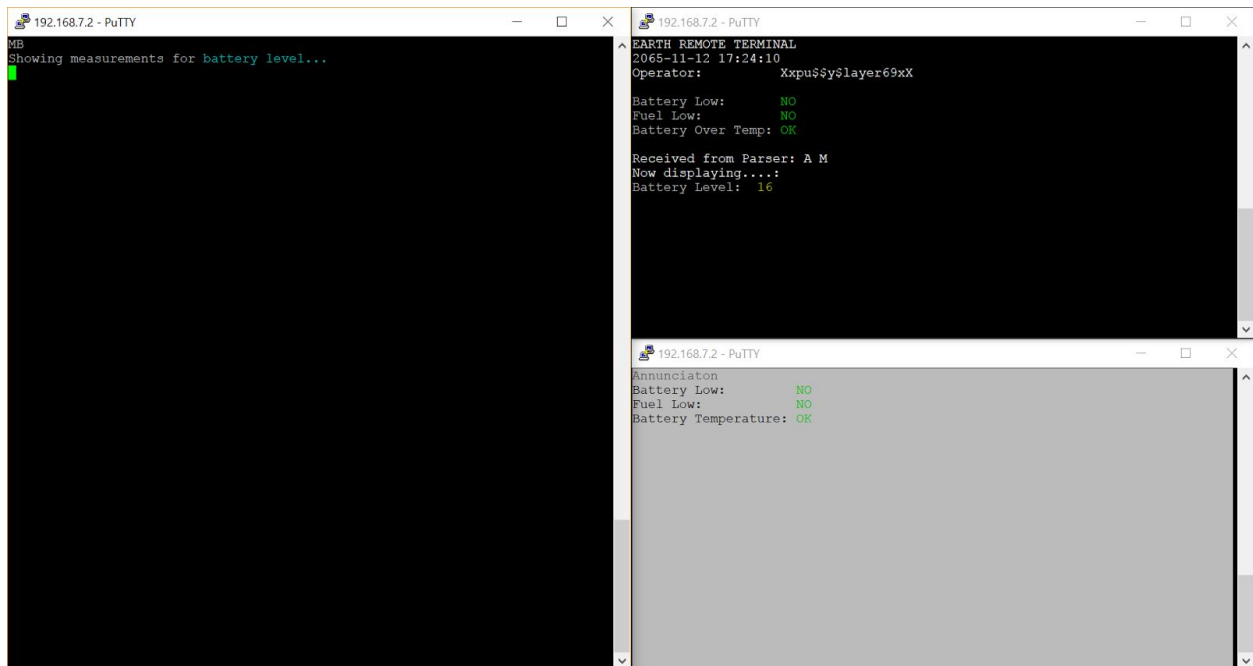


This screen capture shows our interface. The left window (i) is the earth input terminal, the top right window (ii) is the earth remote terminal, and the bottom right window (iii) is the satellite terminal. This state shows what happens when we enter “S” to start up the embedded tasks and show the display.



Conversely, this state shows what happens when we press “P” to stop the embedded tasks. Notice in (ii) and (iii) that it no longer displays any measurement information.

## 7.1.2 Battery Level



The image displays three terminal windows from a PuTTY session connected to 192.168.7.2. The leftmost window shows the command 'MB' being entered, followed by the prompt 'Showing measurements for battery level...'. The top-right window shows the output of the command, including system status (EARTH REMOTE TERMINAL, 2065-11-12 17:24:10), operator information (Operator: Xxpu\$\$y\$layer69xX), and battery status (Battery Low: NO, Fuel Low: NO, Battery Over Temp: OK). It also shows a message 'Received from Parser: A M' and 'Now displaying....: Battery Level: 16'. The bottom-right window shows a summary of the battery status (Annunciator: NO, Battery Low: NO, Fuel Low: NO, Battery Temperature: OK).

```
192.168.7.2 - PuTTY
MB
Showing measurements for battery level...

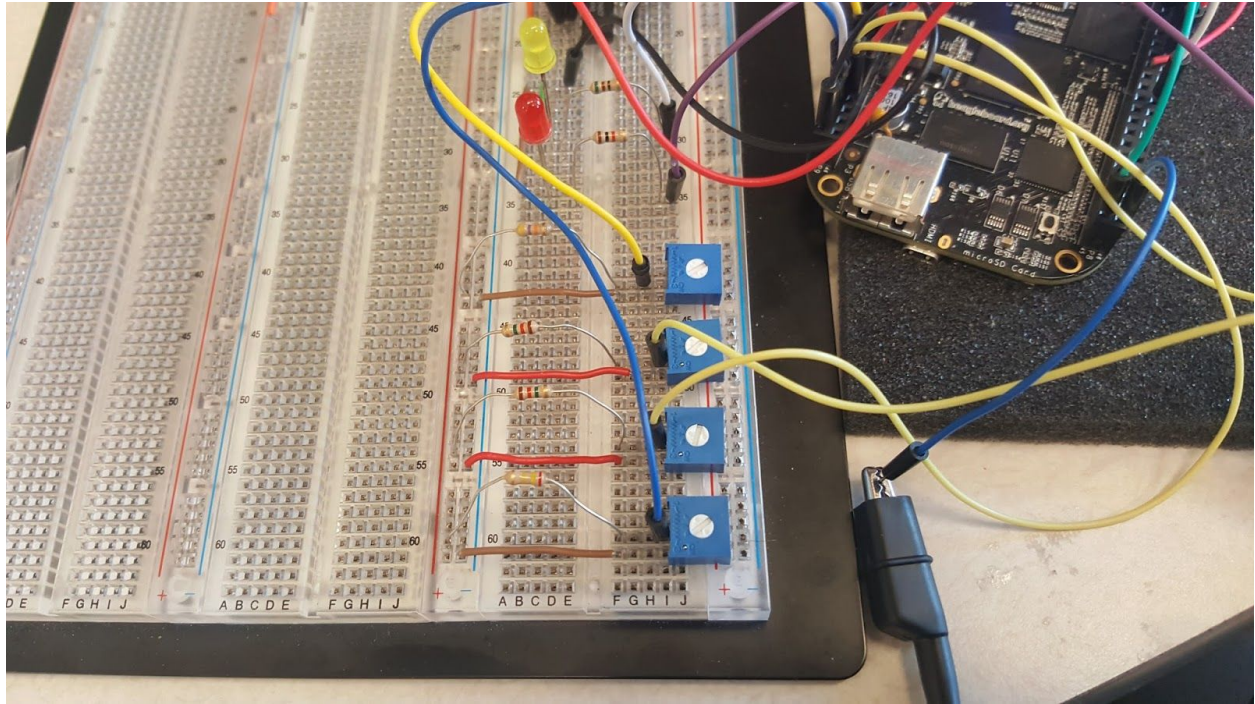
192.168.7.2 - PuTTY
EARTH REMOTE TERMINAL
2065-11-12 17:24:10
Operator: Xxpu$$y$layer69xX

Battery Low: NO
Fuel Low: NO
Battery Over Temp: OK

Received from Parser: A M
Now displaying....:
Battery Level: 16

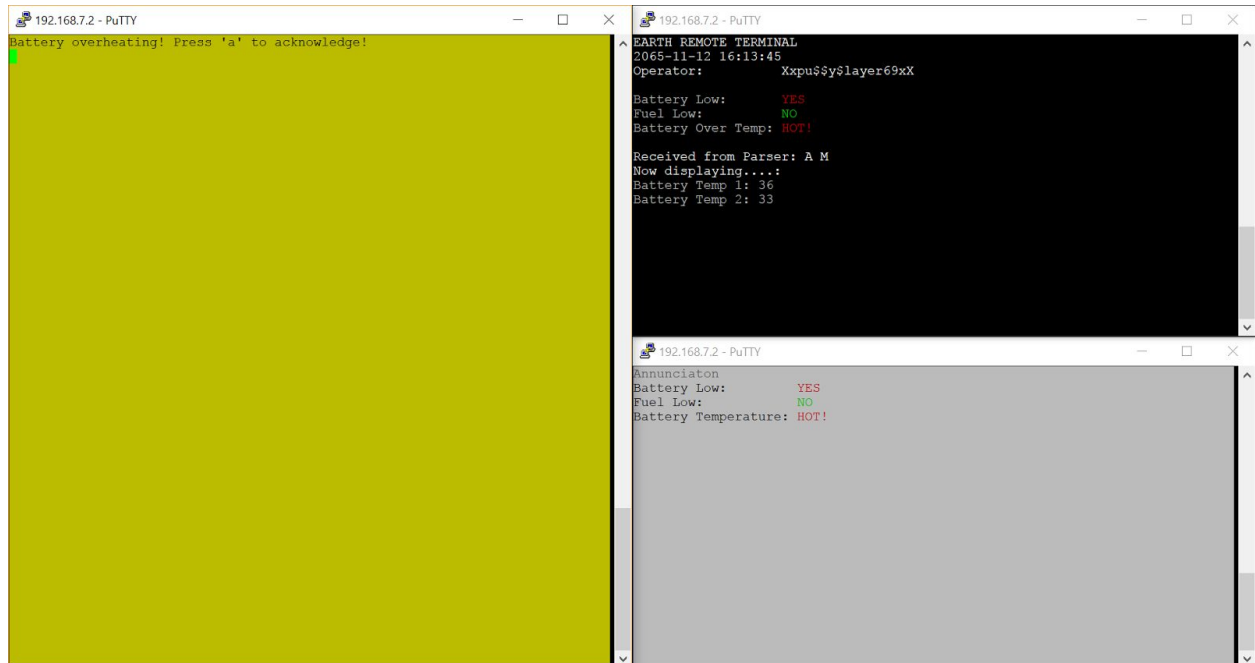
192.168.7.2 - PuTTY
Annunciator
Battery Low: NO
Fuel Low: NO
Battery Temperature: OK
```

This terminal window shows the control of battery level. We display it by entering “MB” in (i), and the measurement is shown in (ii) with a value in this case of 16.

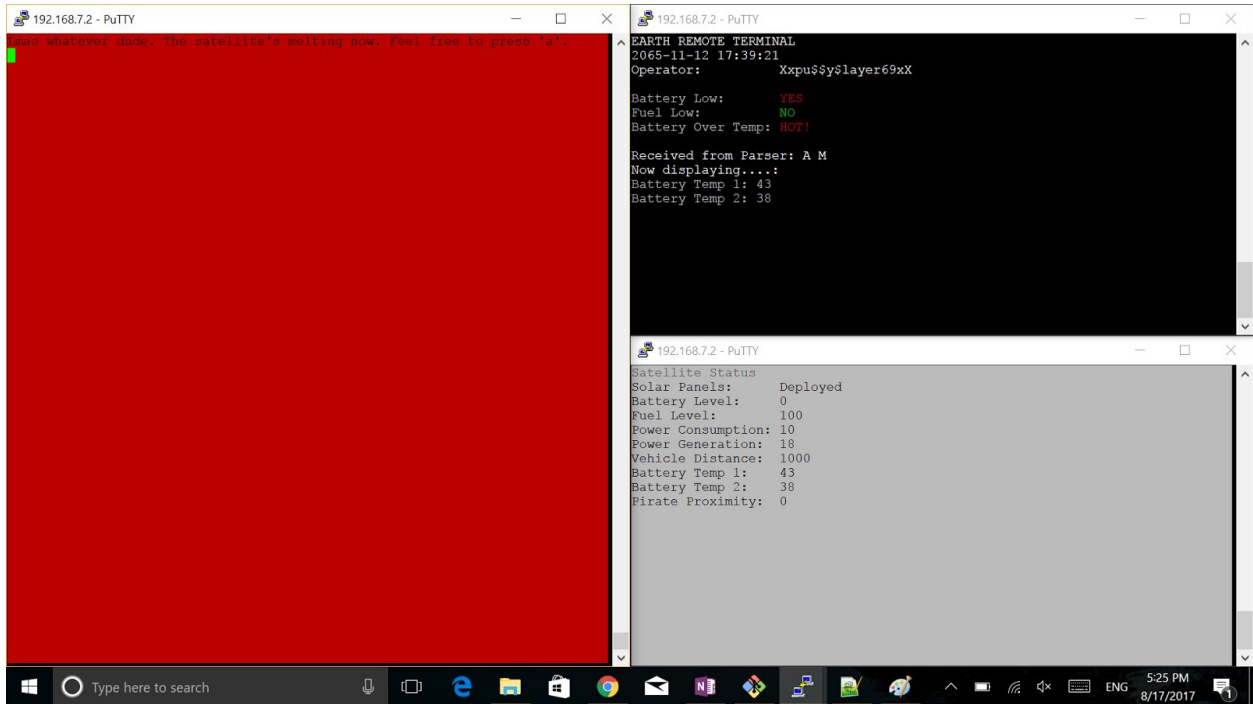


This value is obtained by the ADC, in which its voltage is varied by the potentiometer (closest to us in the picture).

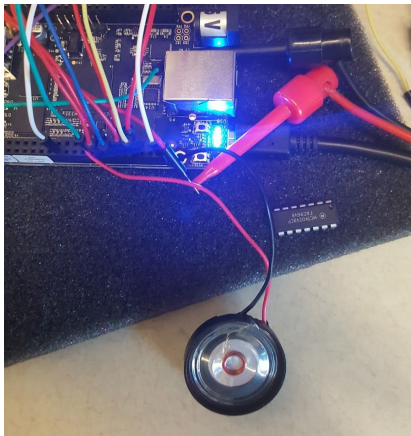
### 7.1.3 Battery Temperature/Warning Alarm



This terminal shows the overheating temperature event. When overheating occurs, meaning that either the battery temp 1 or battery temp 2 transducer's value has varied by over 10% of it's last measurement, (i) is initially colored as yellow with a warning.



If 15 seconds pass unacknowledged, (i) begins to flash red and yellow with a slightly more passive-aggressive warning.



Moreover, when the overheating event occurs, the audible buzzer at the bottom of the picture is sounded using a PWM generated by the beaglebone.

Also, once we are in this 15 second unacknowledged stage, the BeagleBone's LEDs flash as shown.

## 7.1.4 Vehicle Comms

```
192.168.7.2 - PuTTY
F
PIPE - VEHICLE: Received command: F
PIPE - RESPONSE = A F
B
PIPE - VEHICLE: Received command: B
PIPE - RESPONSE = A B
L
PIPE - VEHICLE: Received command: L
PIPE - RESPONSE = A L
R
PIPE - VEHICLE: Received command: R
PIPE - RESPONSE = A R
Y
PIPE - VEHICLE: Received command: Y
PIPE - RESPONSE = A Y
H
PIPE - VEHICLE: Received command: H
PIPE - RESPONSE = A H
U
PIPE - VEHICLE: Received command: U
PIPE - RESPONSE = A U
Running Image Capture...
Frequencies Capture = 16 / 16
Image Status: W
I
Image Status: F
PIPE - VEHICLE: Received command: I
PIPE - RESPONSE = A I
```

In this screen capture, (i) is showing all the commands that are being sent to the pipe and getting a response back from the vehicle.

**Note:** Y replaces “D” for drill down and U replaces “S” for take image due to conflicting user inputs with commandParser.

## 7.1.5 Transport Distance

```
192.168.7.2 - PuTTY
MD
Showing measurements for transport distance...
^C
Calculating transport distance...
Received T...
RESPONSE = K
Received D...
RESPONSE = C
Latest distance measurement: 100
^C
Calculating transport distance...
Latest distance measurement: 1500
Latest distance measurement: 1500
Latest distance measurement: 1500
Latest distance measurement: 1700
Latest distance measurement: 700
Latest distance measurement: 700
Received T...
RESPONSE = K
Received D...
RESPONSE = C
Latest distance measurement: 100
```

```
192.168.7.2 - PuTTY
EARTH REMOTE TERMINAL
2065-11-12 16:29: 1
Operator: Xxpu$Sy$layer69xX
Battery Low: YES
Fuel Low: NO
Battery Over Temp: OK
Received from Parser: A M
Now displaying....:
Transport Distance: 100
```

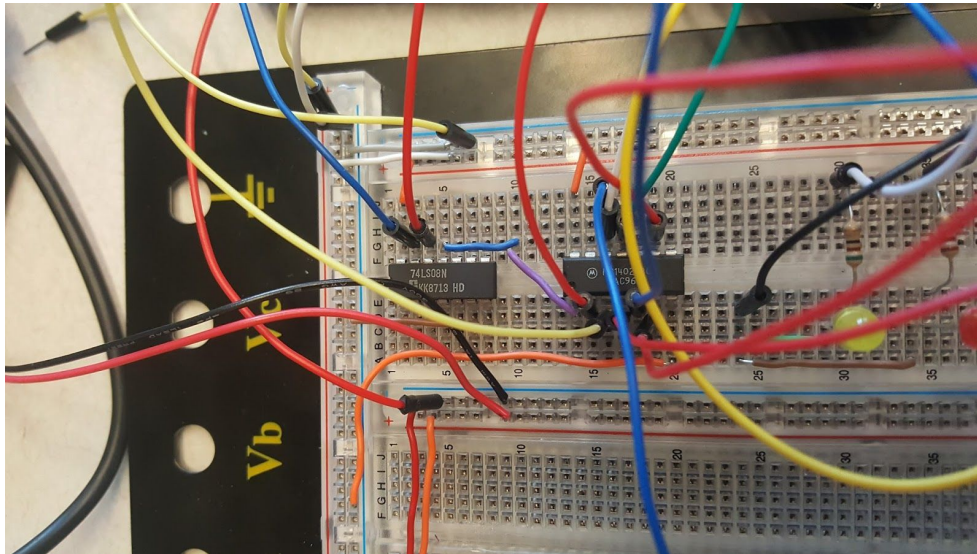
```
192.168.7.2 - PuTTY
Satellite Status
Solar Panels: Deployed
Battery Level: 0
Fuel Level: 100
Power Consumption: 7
Power Generation: 42
Vehicle Distance: 100
Battery Temp 1: 41
Battery Temp 2: 33
Pirate Proximity: 1
```

This screenshot shows the output for transport distance. We simulate the transport coming within 1km by using SIGINT as our “push button”. Once transport distance is active, it will start reading frequencies from the function generator and displaying their corresponding distances on all three terminals. Once distance reaches 100, the Transport Distance task sends out a T and D to vehicle Coms, which responds with K and C respectively, and stopping transport distance.





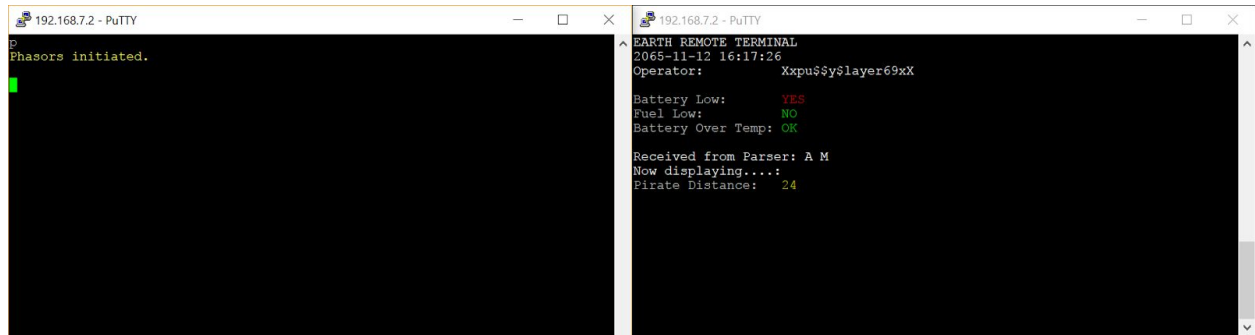
The distance is varied by the frequency from square wave of the function generator, which in this case is 600 Hz. (The higher the frequency, the shorter the distance).



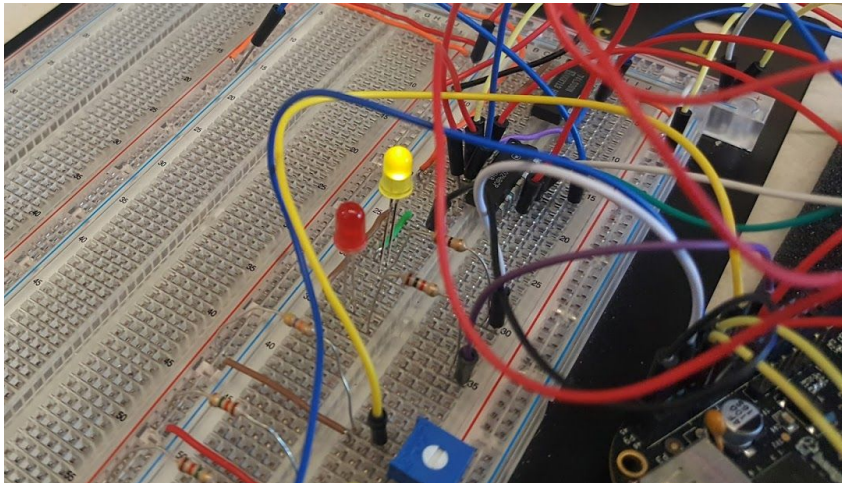
The circuit is controlled by two chips, an AND gate (74LS08N) and a 7 stage ripple counter. The function generator and a beaglebone gpio pin go into the AND inputs. The output goes into the 7 stage ripple counter. Each output bit goes into a gpio pin to the beaglebone. Thus, whenever the gpio pin into the AND gate is echoed as a 1, the counter will increment every negedge of the function generator square wave. Consequently, we can obtain the frequency from the known time and the count.



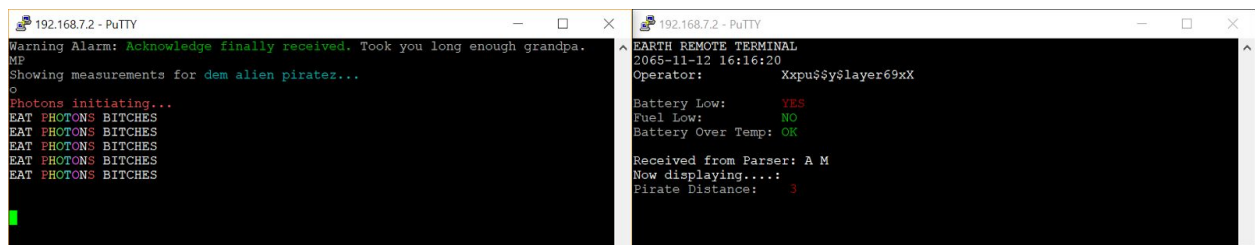
### 7.1.6 Pirate Detection/Management



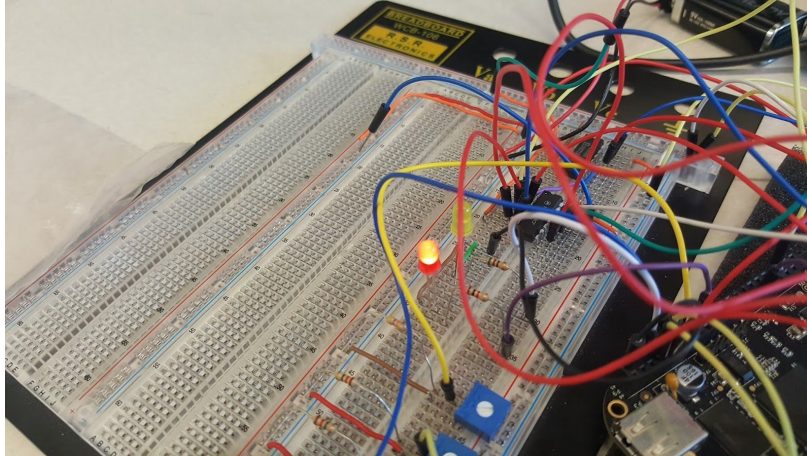
This screenshot shows the terminal output for when the pirates are less than 30 but greater than 5 units from the satellite: (i) shows “Phasors initiated”. Notice that the pirate distance in (ii) shows 24 units.



Additionally, when the phasors are initiated, a yellow LED lights up for a second as shown.



This screenshot shows the terminal display for when the photons are fired. (i) prints out a series of tasteful statements. Note that (ii) shows the pirate distance to be 3 units, under the 5 unit threshold to activate the photons.



Additionally, when the photons are initiated, the red led on the breadboard is flashed several times to simulate the firing of the photons.

### 7.1.7 Image Capture

```

192.168.7.2 - PuTTY
MI
Showing measurements for image frequency...
U
PIPE - VEHICLE: Received command: U
PIPE - RESPONSE = A U
Running Image Capture...
Frequencies Capture = 8 / 16

```

```

192.168.7.2 - PuTTY
EARTH REMOTE TERMINAL
2065-11-12 16:21:27
Operator: Xxpu$$y$layer69xX

Battery Low: YES
Fuel Low: NO
Battery Over Temp: OK

Received from Parser: A M
Now displaying....:
No image captured

```

This screenshot demonstrates the terminal's output midway through screen capture. Here, we see that the vehicle received a user input command "U" (which was changed from "S"), which initiated the image capture to run.

```

192.168.7.2 - PuTTY
MI
Showing measurements for image frequency...
U
PIPE - VEHICLE: Received command: U
PIPE - RESPONSE = A U
Running Image Capture...
Frequencies Capture = 16 / 16
Image Status: N
I
Image Status: P
PIPE - VEHICLE: Received command: I
PIPE - RESPONSE = A I

```

```

192.168.7.2 - PuTTY
EARTH REMOTE TERMINAL
2065-11-12 16:24: 9
Operator: Xxpu$$y$layer69xX

Battery Low: YES
Fuel Low: NO
Battery Over Temp: OK

Received from Parser: A M
Now displaying....:
Image Frequencies:
322 315 311 311
311 315 311 311
311 315 311 311
311 315 311 311

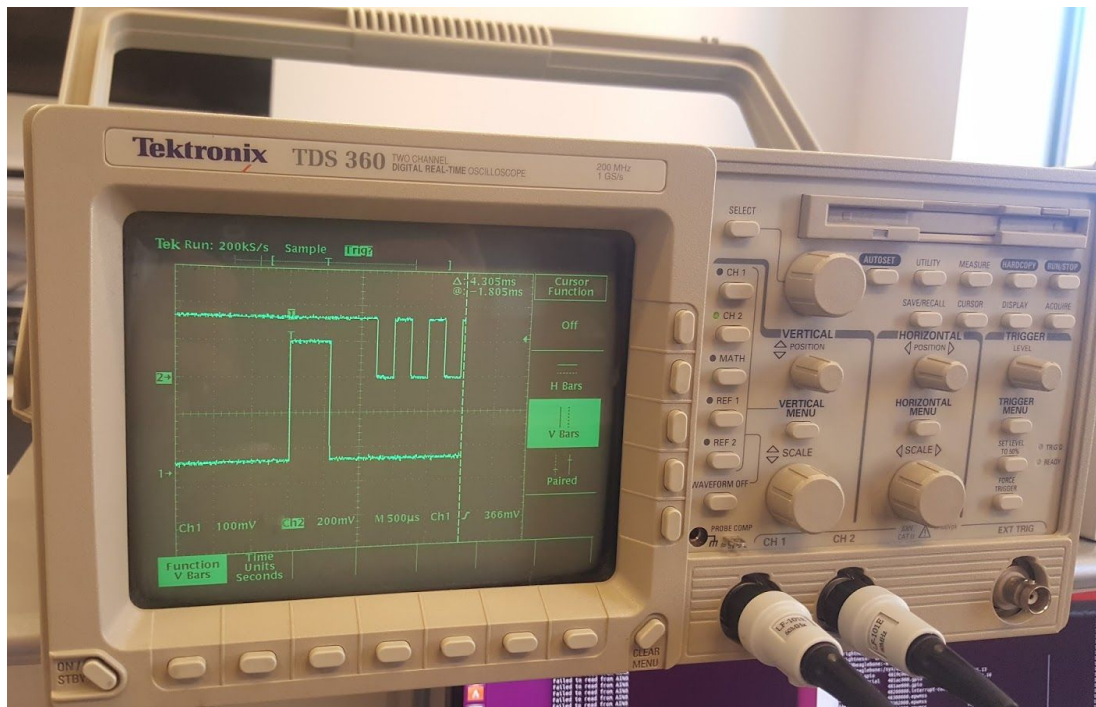
```

This screenshot shows the terminal after the image capture is complete and after the user requests the data transfer to the earth remote terminal. Notice that in (ii), the 16 frequencies are displayed, which is about 311 Hz.



This image shows the frequency of the sine wave that was being sent through the ADC to the image capture task. Here, the original frequency was about 310Hz, which was close to what the terminal displayed, about 311 Hz.

## 7.2 Results for Task Execution Time



The snapshot shows two waves: the top wave shows the PWM for the solar panel control. The Bottom wave is a square wave which indicates the length of the task. When the voltage measured by the GPIO pin was high, the task was initiated. When the voltage goes back low, the task ended. Thus we can measure the time interval for how long the voltage of the GPIO pin was high. Moreover, the duty cycle of the PWM was successfully changed by the motor speed increase and decrease of the function.

	<i>powerSubsystem</i>	<i>thrusterSubsystem</i>	<i>satelliteComms</i>	<i>consoleDisplay</i>	<i>warningAlarm</i>	<i>solarPanelControl</i>	<i>vehicleComs</i>	<i>consoleKeypad</i>	<i>imageCapture</i>	<i>transportDistance</i>	<i>pirateDetection</i>	<i>pirateManagement</i>
<b>Trial 1</b>	3380	928	7110	6640	7010	565	5820	2930	330000	6010	5820	5130
<b>Trial 2</b>	5100	932	7230	6040	4040	795	5660	2810	350000	7030	5660	9020
<b>Trial 3</b>	5340	912	6980	6840	7200	545	5100	2930	340000	6510	5100	5110
<b>Trial 4</b>	4980	916	7310	5960	7060	575	5540	2910	370000	6030	5540	7010
<b>Trial 5</b>	5420	840	7410	5760	5110	645	5780	3090	300000	6910	5780	3010
<b>Average</b>	4844	905.6	7208	6248	6084	625	5580	2934	338000	6498	5580	5856
<b>Error rate (stddev)</b>	837.4246235	37.58723187	168.2854717	465.9613718	1430.220263	102.2252415	289.8275349	100.3992032	25884.35821	476.9905659	289.8275349	2265.277025
<b>Major Cycle</b>	390362.6		<b>Delay Major Cycle</b>	609637.4								
<b>Minor Cycle</b>	25120		<b>Delay Minor Cycle</b>	974880								

The table shows our trial measurements for how long each task runs. Then, we calculated the standard deviation for each task and determined the major and minor cycles. In this case, we considered warning alarm and satellite comms as part of the minor cycle.

### **7.3 Lab Spec Questions**

**8. If a stealth submersible sinks, how do they find it?**

Sheer willpower and luck.

**9. Does a helium filled balloon fall or rise south of the equator?**

Rise.

**10. If you fly faster than the speed of sound, do you have to slow down every now and then to let the sound catch up?**

Probably.

**11. If you fly really really fast around the world, can you catch up to the sound before it catches up to you and answer a question before you hear it?**

Sure.

**12. If you don't answer a cell phone call, where does it go? Is it just sitting there waiting for you?**

Were you drunk when you wrote these questions? No beers until after you're done.

## 8. ANALYSIS OF ANY RESOLVED ERRORS

In this section, we discuss the errors that we encountered and how we resolved them.

### 8.1 Measuring Errors for files

Filename	File Description	Error (s)	Cause (s)	Resolution
Console Display	Displays onto satellite terminal	Wasn't switching modes	Didn't implement reading input and not passing it to terminal coms	Used ungetc to put character into buffer if not a valid mode switching command
Terminal Coms	Sets up another terminal, reads a file and dprints the contents	N/A File didn't change from last phase	N/A	N/A
Satellite Coms	Sends information to terminal Coms and obtains thruster command from random number generator	Did not print to the earth display terminal	Didn't implement another pseudo-terminal	Opened another /dev/pts/ terminal to be available to print

Warning Alarm	Flashes corresponding LEDs based on battery level and fuel level	LEDs were not turning on and off correctly	Didn't save the initial values every cycle of the global counter correctly	Used additional static variables to indicate whenever the state of an LED changed
Vehicle Coms	Controls the interface between the vehicle and satellite	Response from vehicle was not stored into the response variable correctly	Pointer was not updating in the main task function but was updating in the local function	Set response pointer in the main function
Keyboard Console	Reads user input	N/A	N/A	N/A
Solar Panel Control	Updates the solar panel state	N/A	N/A	N/A
PWM	The general PWM from the pins of the beaglebone	Couldn't run both PWM pins at the same time on the beaglebone	Douglas Smith, the demo reviewer for our project, revealed that two PWMs can't run simultaneously on the pins	Tested the thruster subsystem and solar panel control PWM individually
ADC readings	Volt measurements read by the ADC of the beaglebone	The ADC values were random	Negative end was not grounded to the Ground ADC port on the GPIO	Grounded it



Hardware Counter	Used in transport distance to measure the frequency	The counter kept climbing in value every measurement	The reset of the counter was not working	Used the 3.3 V from the beaglebone to power the counter instead
------------------	---	--	--	---

## 8.2 GPIO Pin Error in measurement

This section is a followup to the error for the empirical task measurement in the presentation and results section. The following pseudocode demonstrates our measurement tactic in *gpio.c*:

*gpio*:

```

open GPIO pin stream;
init array containing all 11 tasks;
init taskIndex to current index of array containing all 11 tasks;
init timeTask to task to be measured; // based on taskIndex
while(true) {
    if (timeTask == taskIndex) {
        write 1 to gpio stream // task has started
    }
    run task we want to measure
    if (timeTask == taskIndex) {
        write 0 to gpio stream // task has ended
    }
}

```

We used the oscilloscope probes, at 10x, to measure the voltage difference when the gpio is set to 1 and then at 0. We then paused the waveform and used the scope's vertical cursors to determine the time interval, which estimates the task's execution time.

## **9. ANALYSIS OF UNRESOLVED ERRORS**

### **9.1 Task Time Standard Deviation**

The main unresolved error was the slight imprecision when measuring the task length.

Ideally, we want the standard deviation to be perfectly 0, meaning the five trials we did for each task would yield the exact same number. To further minimize the error, by the Law of Large Numbers, more trials would have given results that approached the “true” execution time.

Moreover, since some of the tasks are dependent on user input, when no user input is received, the task time usually takes much shorter. Thus, for a more consistent task value, we could be more pedantic and split the times into subcategories: “user input” and “no user input”, thus having more precise values and consequently reducing error.

And finally, due to the nature of more tasks being dynamically scheduled, this significantly varies the major cycle delay clock. We can fix this by adjusting the delay based on every permutation of the queue.

### **9.2 Battery Level Updating Error**

We were unable to correctly display battery level updates on the terminal. We believe that the variables were not shared correctly across the power subsystem task and terminal display tasks. Due to time constraints, we were not able to implement the fixes. However, for future projects, we will utilize GDB, print statements, and observation within the power subsystem and display tasks to isolate and correct the error.

### 9.3 Image Capture Limitations

Due to the slow speed of the sysfs, the sampling frequency limits the image capture to a frequency range between 0 and 500 Hz. Thus, we weren't able to capture the 30 to 3.75kHz range as specified in the spec. Moreover, the captured frequencies deviate by a couple Hz compared with the frequency generator. This is dependent on the precision of the ADC. Next time, we could have fixed these errors if we didn't use sysfs, but instead used a more direct memory map. This would have been faster, albeit more complex to implement.

## 10. SUMMARY

To summarize the report, we will quickly examine the main ideas from each of the major sections: (i) design specification, (ii) software implementation, (iii) tests, (iv) presentation and (v) errors.

(i) The report discussed the design specifications for the *Satellite Management and Control System*, from the high level system requirements to the low level details for each task. In addition, the specifications discuss Phase III-IV of this project in which we extend the satellite communication to earth through a command parser, add further battery level ADC functionality, extend vehicle communication, a PWM driven solar panel control and thruster subsystem, image capture from the mining vehicle, manage transport distance with the transport vehicle, and add pirate detection/management.

(ii) In this section, we demonstrate our design process in the form of UML diagrams and pseudocode. We explained how these UML diagrams helped us formulate an underlying structure and strategy for implementing Phase III-IV. Specific diagrams include the high level block diagram of our system, which showed the 11 tasks and how each TCB was implemented. We created a use case diagram which showed the interactions between the earth station, the

satellite, and the mining vehicle. We also created UML diagrams for warning alarm, power subsystem, thruster subsystem, pirate detection/management, and including how each function is updated. Lastly, the pseudocode implementation for all 11 tasks, along with the associated design decisions have been provided.

(iii) The test sections in this report discuss what and how we tested each task separately to make sure each piece of the system was functioning properly. In particular, the addition of several hardware peripherals (e.g. added ADC channels, hardware counter, external LEDs, etc) required more isolated unit testing to separate hardware bugs from software bugs.

(iv) The presentation and results section outlined what the output was for each task with associated screenshots and images. In these subsections, the report explained what each terminal looked like, and how the hardware and circuitry influenced what was being displayed. Moreover, the report details the results from the tasks being measured on the oscilloscope.

(v) The errors section demonstrate the resolved and unresolved issues we came across during the implementation of the system. The resolved section discusses the main errors that were dealt with. These errors include the implementation of each task, as well as certain hardware issues with the PWM, ADC, and hardware counter. The unresolved section examines the errors that were not resolved. Specifically, there were the deviation of times that were measured across multiple trials for each task, and the image capture was limited due to the slow execution speed of the sysfs to capture adc values.

## 11. CONCLUSION

In conclusion, this lab helped us gain a better understanding of voltage divider circuits, hardware counters, and protocol based communication on the beaglebone. Moreover, we broadened our understanding of pipes, scheduling, pointers, UML diagrams, and hardware integration; all of which helped us further develop and refine our design, improving the safety, reliability, and realism of the target system.

## 12. APPENDICES

We adapted certain supplementary programs written by Jim Peckol and other authors to aid us in getting our system to work. We adapted the following code to use for our system:

- *rand2.c* to generate a 16 bit number for the thruster command.
- *terminalComs0.c* to allow communication between two open Linux terminals.
- *blinkusr.c* and *libBBB.c* by Gavin Strunk for Beaglebone specific interfacing functionality. (i.e. leds, PWM, and ADC)
- *Kernel3.c* to build our scheduler.
- *rw0.c* to build the pipe.
- *Float\_FFT* to capture images.

These files can be found under Peckol's Archive of Code [here](#).

Color was used to enhance the information displayed. These codes were taken from websites that provided the ANSI escape sequences.

### 13. CONTRIBUTION

Name	Contribution
Abdul Katani	Created the skeleton for the solar panel control and implemented it also contributed to thrustersSubsysem.c, powerSubsystem.c., pirate subsystes, image capture and the lab report. Worked with the team on the GPIO A/D functionality plus the PWM implementation. also built the circuit and tested it functionality for the A/D and transport distance
Radleigh Ang	Created skeleton for overall system, contributed to main.c and the dynamic queue, implemented satelliteComs.c, consoleDisplay.c, vehicleComs.c, pirate subsystem, image capture, transport distance. Created user interface. Contributed to report.
Daniel Snitkovskiy	Updated powerSubsytem.c and thrusterSubsystem.c, added PWM and ADC functionality, implemented the Dynamic Task Queue, Startup Task, solarPanelControl and commandParser. Integrated software interrupts to all parts of the system.

## Time Spent with Project

Estimate on Time Spent	Hours
Design	15
Coding	40
Test/Debug	85
Documentation	20

## Signature

The undersigned testify to the best of their knowledge that this report and its contents are solely their own work and that any outside references used are cited.

Radleigh Ang

---

Author 1

Daniel Snitkovskiy

---

Author 2

Abdul Katani

---

Author 3