

EE/CSE 474 Lab Report
Project 3



Radleigh Ang

Abdul Katani

Daniel Snitkovskiy

TABLE OF CONTENTS

ABSTRACT	4
1. INTRODUCTION	6
2. DESIGN SPECIFICATIONS	7
2.1 System Requirements	7
2.1.1 General Summary of Satellite Management and Control System	7
2.1.2 Subsystems Overview	8
2.1.3 Scheduler for these Tasks	10
2.2 Specific Details and Requirements	10
2.2.1 Tasks and Task Control Blocks	10
2.2.2 Requirements for Power Subsystem	11
2.2.3 Requirements for Thruster Subsystem	13
2.2.4 Requirements for Satellite Coms	13
2.2.5 Requirements for Console Display	14
2.2.6 Requirements for Warning Alarm	15
2.2.7 Requirements for Solar Panel Control	15
2.2.8 Requirements for Console Keypad	16
2.2.9 Requirements for Vehicle Comms	16
3. SOFTWARE IMPLEMENTATION	17
3.1 High-Level Block Diagram	17
3.2 General UML Diagrams	18
3.3 UML Activity Diagrams	21
3.3.1. Warning Alarm	21
3.3.2. Power Subsystem	22
3.3.3. Thruster Subsystem	23
3.4 UML Sequence Diagram	24
3.5 Function Prototypes for Tasks	25
3.6 Data Structs Defined for Application	26
3.7 Pseudocode	28
3.7.1 WarningAlarm Pseudocode	28
3.7.2 SatelliteComs Pseudocode	30
3.7.3 ConsoleDisplay Pseudocode	31

3.7.4 Power Subsystem Pseudocode	31
3.7.5 Thruster Subsystem Pseudocode	35
3.7.6 solarPanelControl Pseudocode	37
3.7.7 keyboardConsole Pseudocode	38
3.7.8 vehicleComs Pseudocode	39
3.7.9 Dynamic Task Queue Pseudocode	40
3.7.10 Startup Task Pseudocode	43
3.7.11 Scheduler Pseudocode	44
4. TEST PLAN	46
5. TEST SPECIFICATION	47
6. TEST CASES	49
7. PRESENTATION AND RESULTS	54
7.1 Output of the System	54
7.2 Results for Task Execution Time	56
7.3 Lab Spec Questions	58
8. ANALYSIS OF ANY RESOLVED ERRORS	59
8.1 Measuring Errors for files	59
8.2 GPIO Pin Error in measurement	61
9. ANALYSIS OF UNRESOLVED ERRORS	62
9.1 Task Time Standard Deviation	62
9.2 Battery Level Updating Error	63
10. SUMMARY	63
11. CONCLUSION	64
12. APPENDICES	65
13. CONTRIBUTION	65
14. TIME SPENT WITH PROJECT	66
15. SIGNATURE	66

ABSTRACT

The objective of this lab was to refine the design of the Satellite Management and Control System begun in Project 2. The additions of this project included:

1. A system initialization function.
2. A dynamic task queue.
3. Several external peripherals into the system (i.e PWM and ADC).
4. Serial communication capabilities between the mining vehicle and Satellite Management and Control System.
5. Various amendments of Project 2 functions.

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. Implement and test the dynamic task queue and system initialization function using the existing Project 2 infrastructure.
2. Amend existing communication TCBs related to output/communication (Satellite Comms and Console Display), and integrate these with the new “Vehicle Comms” TCB to facilitate serial communication with the mining vehicle.
3. Implement and test simple functions that interface with the PWM and ADC.
4. Integrate PWM and ADC functionality with existing internal state management TCBs (Thruster Subsystem and Power Subsystem), adding in the new “SolarPanel Controls” TCB in the process.

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 eight 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. Lastly, the input terminal can control the speed at which the motor drive operates the solar panels. If the battery or fuel is low, the system can flash leds at specific intervals on the panel and serve as a warning.

Internally, the system’s thrusters are controlled by a random 16 bit number, 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 gained a deeper understanding of the C language, refined our use of formal design tools, and utilized several pieces of external sensors and devices to further model the behavior 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 I's design, adding functionality that would improve the safety and reliability of the system. Some features were introduced to make the design of the system more robust, such as the system initialization task and the dynamic task queuing; other features were used to more accurately model the behavior of the target system, such as introducing an analog-to-digital converter (ADC) to simulate the battery, and pulse width modulation (PWM) to simulate the thruster/solar panel motors. These features were conceptualized, 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 Ω and 100 Ω)
- Potentiometer with max resistance of 1 k Ω
- Breadboard Wires

2. DESIGN SPECIFICATIONS

This section shows the design specifications for Project 3.

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).
 - *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 the information in the form of status, annunciation, and alarm through 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 level and fuel level. 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 one 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 and Satellite Comms).

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 eight TCB elements in the queue correspond to tasks managing the powerSubsystem, solarPanelControl, thrusterSubsystem, satelliteComms, vehicleComms, consoleDisplay, consoleKeypad, and warningAlarm. Currently, all tasks are to be executed normally except for the solar panel control and console keypad tasks. These two tasks are only to be executed during solar panel deployment.

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

The following gives a C declaration for such a TCB.

```
struct MyStruct
{
    void (*myTask)(void*);
    void* taskDataPtr;
    struct MyStruct* next;
    struct MyStruct* prev;
};
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

Data transfer from the satellite to the earth shall be the following status information:

- Fuel Low
- Battery Low
- Solar Panel State
- Battery Level
- Fuel Level
- Power Consumption
- Power Generation
- Vehicle Status

The thrust command (a randomly generated 16-bit number) shall be interpreted as follows:

<i>Thruster ON</i>	<i>Bits 3-0</i>
<i>Left</i>	<i>0</i>
<i>Right</i>	<i>1</i>
<i>Up</i>	<i>2</i>
<i>Down</i>	<i>3</i>
<i>Magnitude</i>	<i>Bits 7-3</i>
<i>OFF</i>	<i>0000</i>
<i>Max</i>	<i>1111</i>
<i>Duration - sec</i>	<i>Bits 15-8</i>
<i>0</i>	<i>0000</i>
<i>255</i>	<i>1111 1111</i>

This command is to be sent to the thruster subsystem to be used.

2.2.5 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

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

- Fuel Low
- Battery Low

2.2.6 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.

2.2.7 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.

2.2.8 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.

2.2.9 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.

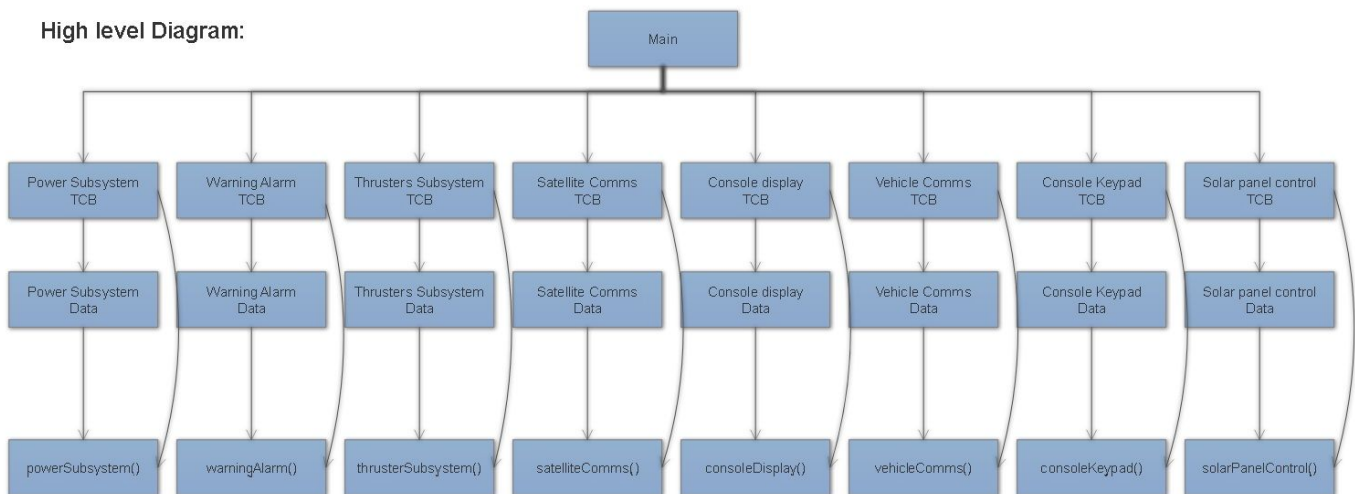
For the present phase, the following commands and responses will be implemented:

- Commands:
 - F Forward
 - B Back
 - L Left
 - R Right
 - D Drill down – Start
 - H Drill up – Stop
- Response: A <Solar Panel Command sent>

3. SOFTWARE IMPLEMENTATION

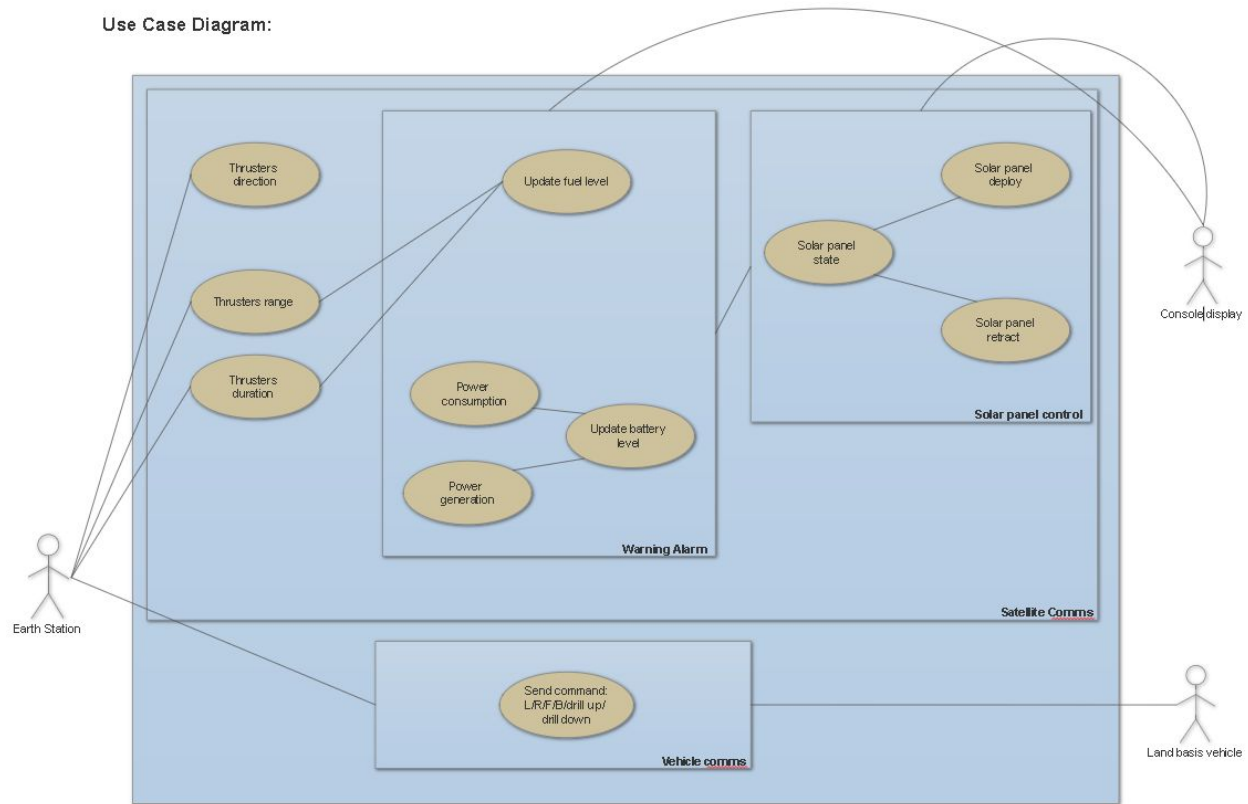
The following section describes in detail how the specifications was implemented in this project. First, a top level view of the code will be explored through a high-level block diagram, and the individual TCB methods will be modeled via UML Activity Diagrams. Then, the relationship and sequence of all 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



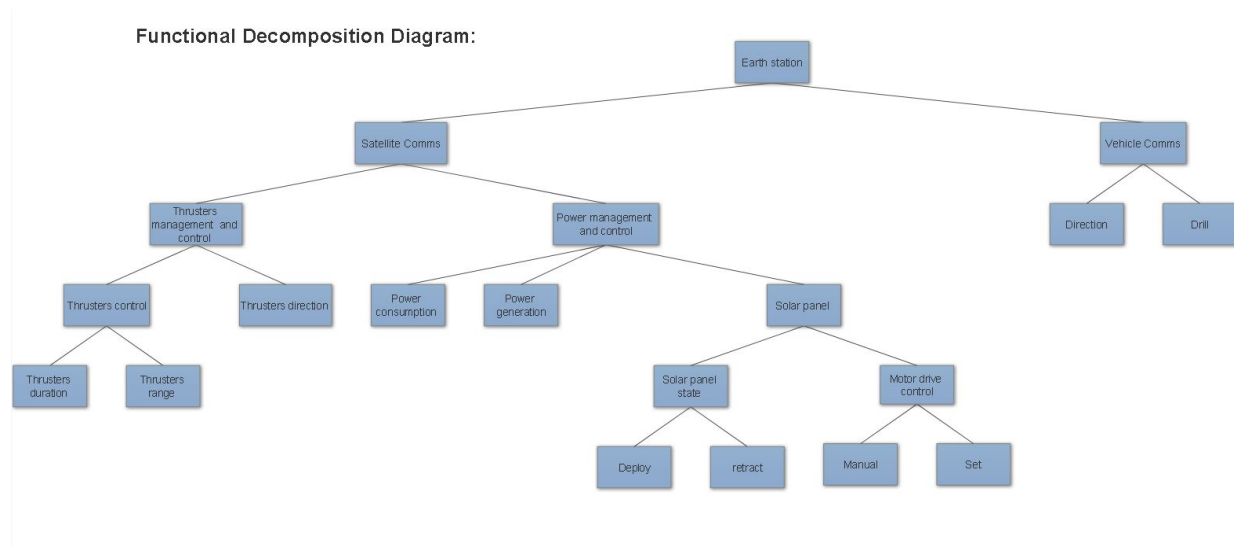
From this diagram, we see that the main function coordinates the eight different TCBs. Within each TCB is a reference to a struct with the shared variables and a function that uses them. For example, we see that main creates a Power Subsystem TCB which points to shared data. Both the TCB and the shared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

3.2 General UML Diagrams



(Figure 1: Use Cases Diagram)

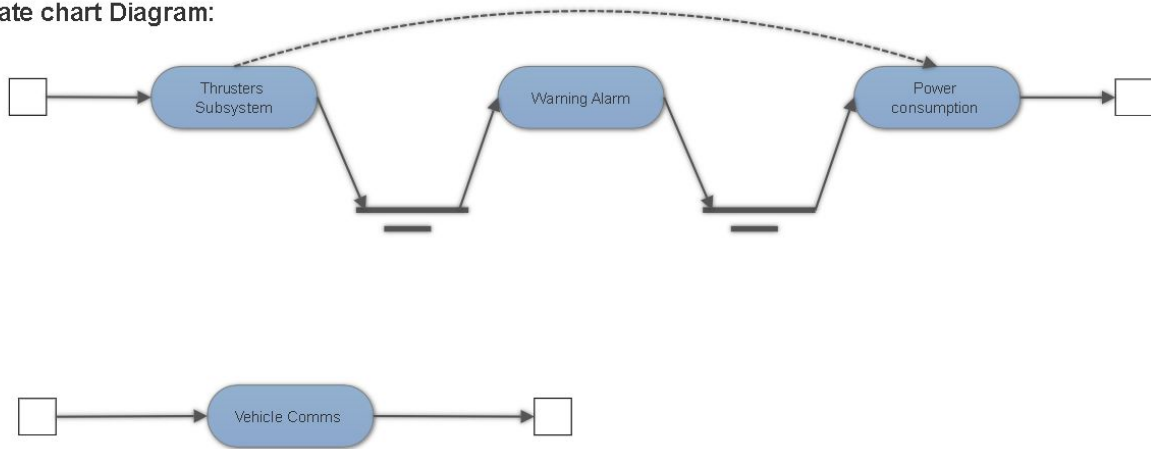
This use case diagram visualizes the interactions between the Earth station, the vehicle, and the satellite. Here, we see that the system objectives are divided into four sections: (i) warning alarm, (ii) solar panel control, both of which are contained in (iii) satellite coms, and (iv) vehicle comms.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



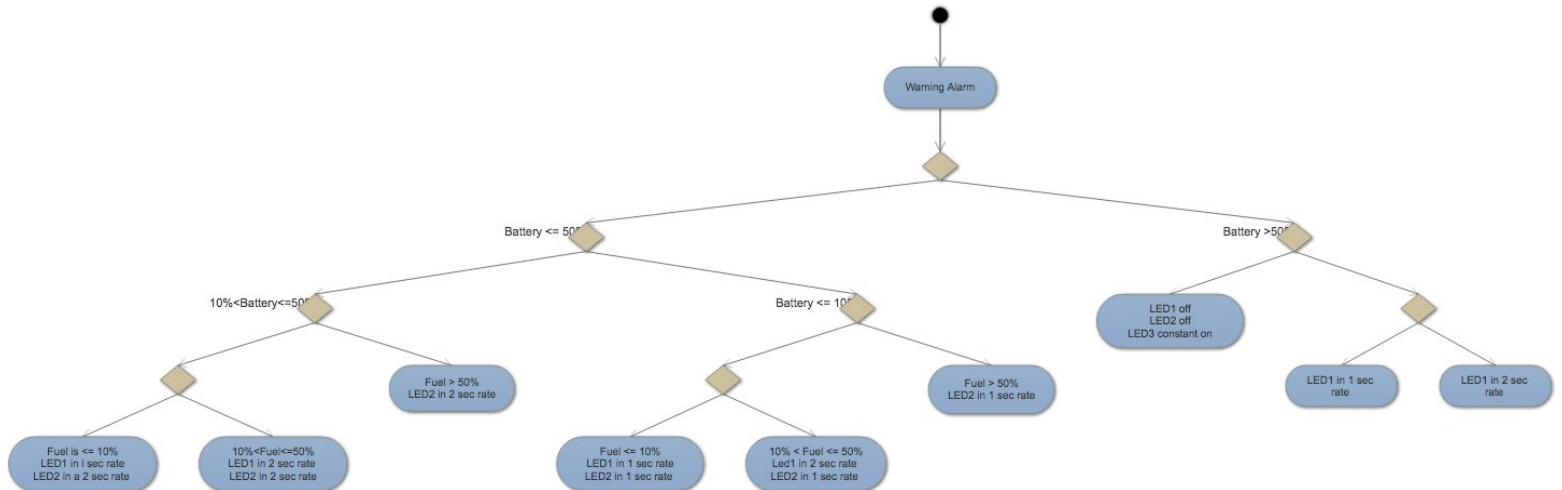
(Figure 3: State Chart Diagram)

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

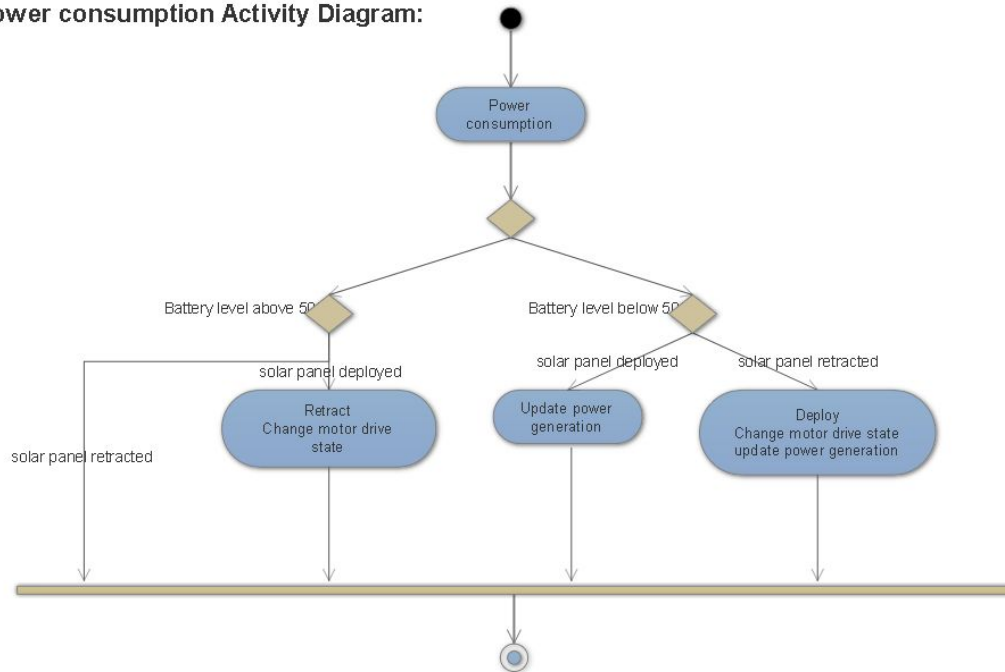


This figure demonstrates the decision points that the warning alarm function encounters.

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.

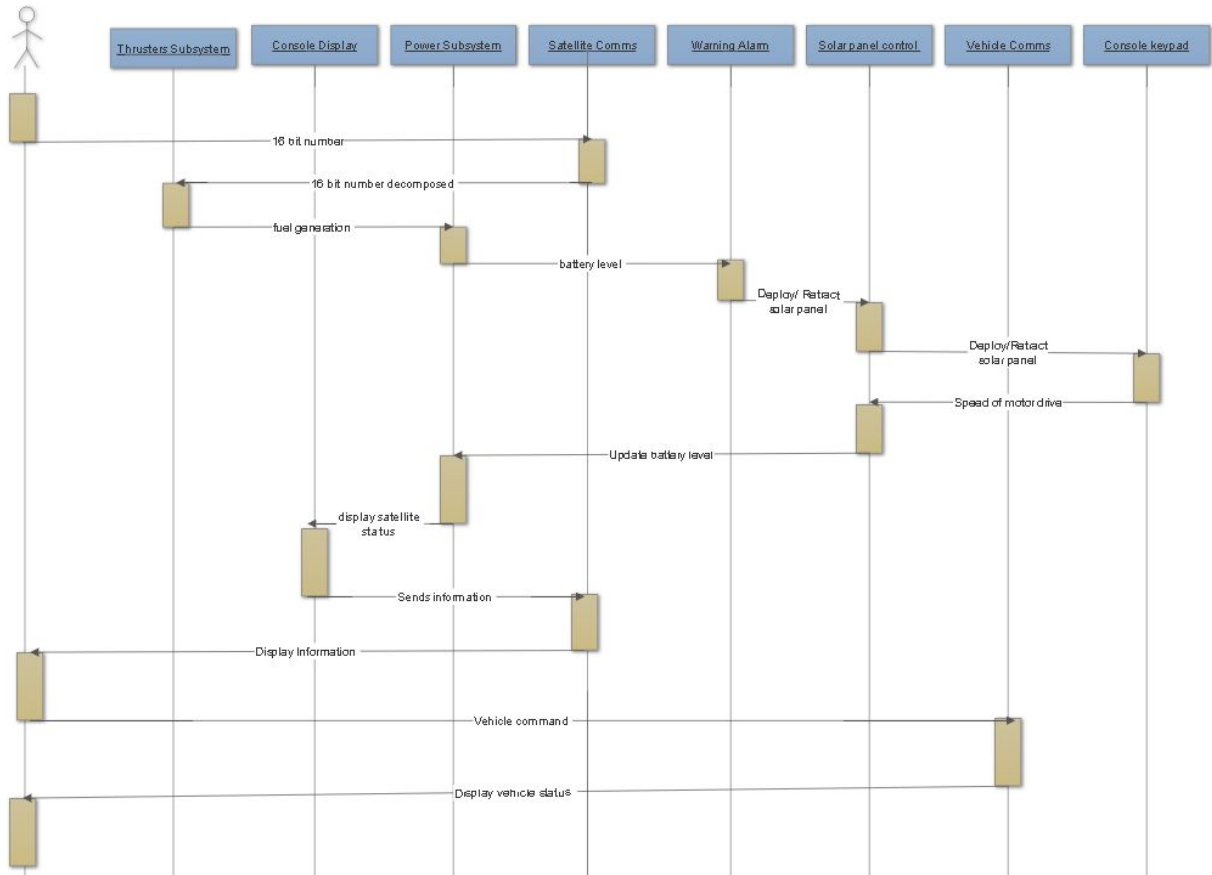
3.3.3. Thruster Subsystem



This activity diagram shows 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.

3.4 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 solarPanelControl(void *solarStruct);*
 - Controls the solar panels.
- *void keyboardConsole(void *keyboardStruct);*
 - Controls the keyboard console.
- *void vehicleComms(void *vehicleStruct);*
 - Controls communication with the vehicle.
- *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

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;
```

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

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

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

3.7 Pseudocode

The following sections provide pseudocode implementations of the 8 TCB functions. For a complete implementation in C, see the attached “a3.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_diration = 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 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 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 all eight TCBs.
```

ActivateTimeBase():

```
set GlobalCounter = 0;
```

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.

3.7.11 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 (should add/remove SolarPanelControlTCB and KeyBoardConsoleTCB) {
        if (append == 1) {
            Add SolarPanelControlTCB and KeyBoardConsoleTCB to TaskQueue;
            set append = 1, remove = 0;
        } else { // remove == 1
            Remove SolarPanelControlTCB and KeyBoardConsoleTCB to TaskQueue;
            set append = 1, remove = 0;
        }
    }
    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 every TCB excluding Warning Alarm, Satellite Comms, Vehicle Comms and Console Display.

Design Decisions:

1. Utilizing flags to denote whether SolarPanelControlTCB and KeyBoardConsoleTCB 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.

4. TEST PLAN

In this Project, complexity of the software and hardware components increased, meaning unit and integration testing that isolated key functions was essential. To meet this added complexity, each 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, and Warning Alarm. 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, and Warning Alarm. 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 2. Power generation 3. Solar panel state 4. Solar panel deploy 5. Solar panel retract 6. Battery level	1. $0 \leq x \leq 12$ 2. $x \geq 0$ 3. TRUE/FALSE 4. TRUE/FALSE 5. TRUE/FALSE 6. $0 \leq x \leq 1800$	1. $x = 0, x = 12$ 2. $x = 0$ 3. N/A 4. N/A 5. N/A 4. $x = 0, x = 1800$
Thruster Subsystem	1. Thruster Command 2. Fuel Level 3. PWM: duty 4. PWM: period	1. 16-bit int 2. $0 \leq x \leq 100$ 3. $0 \leq x \leq 16$ 4. $0 \leq x \leq 16$	1. N/A 2. $x = 0, x = 100$ 3. $x = 0, x = 16$ 4. $x = 0, x = 16$
Satellite Comms	1. Thruster Command 2. Response	1. 16-bit int 2. "A <valid cmd>"	1. N/A 2. 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. $x = 0, x = 500000$

6. TEST CASES

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

TCB	How It Was Tested	Tools/Strategies (if applicable)
Power Subsystem	<p>1. Ensuring Power Consumption remains in a valid state. We tested this utilizing unit tests to compare the expected output with the resulting output of the TCB function of the the four conditions in which Power Consumption would update its value: reversed condition vs. non-reversed condition and even vs. odd function call number.</p> <p>2. Ensuring Power Generation remains in a valid state. Similarly to Power Consumption, unit tests were used to 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>3. Ensuring the Solar Panel variables remain in valid states. As with both 1 and 2, unit tests were used to compare the expected output when solar panels were deployed vs. retracted and the state ON vs. OFF.</p> <p>4. Testing the ADC interface The ADC interface (contained in adc_utils.*), was tested using a toy program designed to test the basic functions (initialization/measurement). This file can be found in adc_test.c within a3.zip.</p>	Google Test C++ Testing Framework, and toy programming

Thruster Subsystem	<p>1. Verifying that the thruster command was being properly decoded. To test this, a unit test was constructed that went through all the permutations of thruster_dirations (up, down, left, right), magnitudes (0-100), and durations (0-255) via nested for-loops.</p> <p>2. Verifying that the cost was being properly calculated. To test this, a unit test was constructed to compare a known command with a precalculated cost was compared with the output of the system.</p> <p>3. Testing the PWM interface 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 a3.zip.</p>	Google Test C++ Testing Framework, and toy programming
Satellite Comms	<p>1. Random integer is no greater than maximum for a 16-bit number ($2^{16}-1$) We tested this by printing the thrusterCommand integer onto the terminal and noted if any of the numbers exceeded $2^{16}-1$.</p> <p>2. Integer's third and fourth bit are masked to 0 correctly Printed the integer after it's been masked. Converted it to binary to make sure the third and fourth bit gets 0.</p> <p>3. Satellite Status and annunciation are printed correctly. Used simple print and check for all the values. Inspected the printed values to ensure they were updating correctly.</p> <p>4. Vehicle Response and Command are shared correctly with vehicleComs. Assigned and printed the response and command variables onto the console.</p>	Output inspection & Print Statements

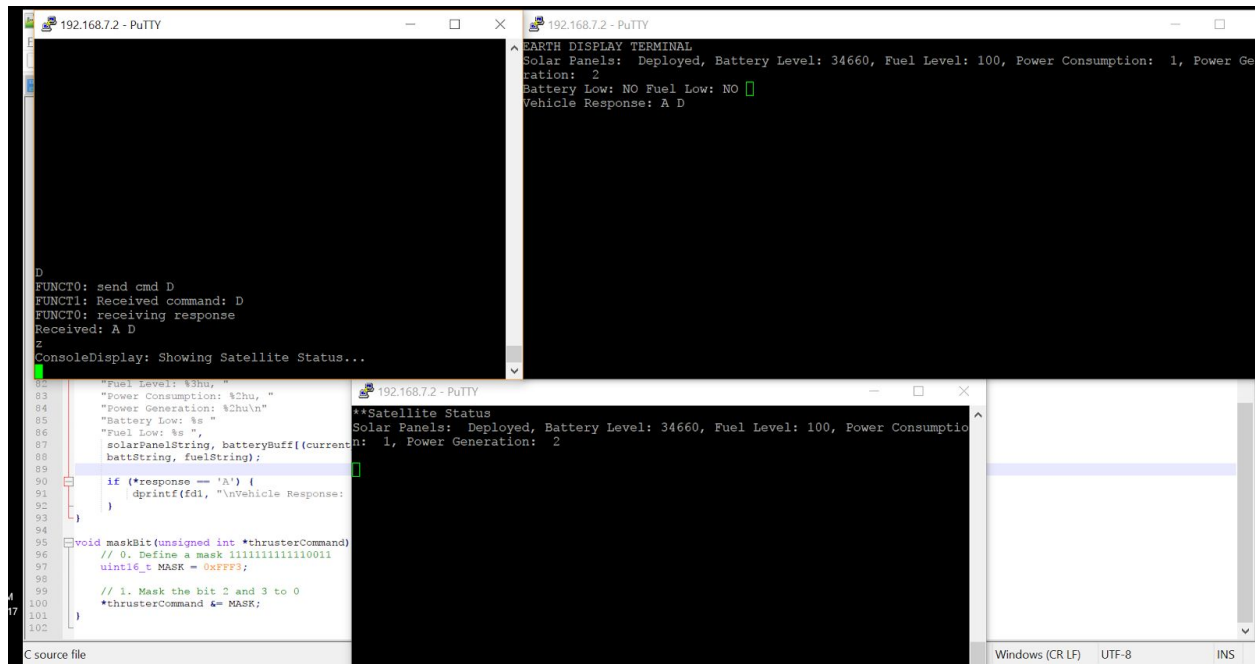
Warning Subsystem	<p>1. Battery low and fuel low are true when their values go below 10%. Initialized battery level and fuel level to just above 10%, and observed whether the fuel low and battery low values change when the levels change.</p> <p>2. Leds are flashing correctly based on battery and fuel level. Initialized battery and fuel level to appropriate testing value (above 50, in between 10 and 50, and below 10). Observed 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>1. All commands are printed correctly on the satellite terminal First, printed the values onto the controlling terminal. Then, printed the values onto the pseudo terminal. Verified that the values were updating.</p> <p>2. Switching modes operated as intended Printed the character that was read in from stdin onto the console to ensure the right character is stored. Used simple print statements to print out the mode based on character input. Also, tested that there is no output if an invalid character is received.</p>	Output inspection & Print Statements

Vehicle Coms	<p>1. Pipe is communicating as intended Assigned a local character variable to write into the pipe. Used 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". Used similar print statements to track the response from the vehicle to when it's read in the main vehicleComs task function.</p> <p>2. Reading user input is functioning correctly Printed the character read from the stdin buffer to check if it's storing it correctly. Tested to make sure invalid commands are ignored and that no output is produced. Fed user input into pipe and printed every step of the way (similar to the first test case).</p>	Output inspection & Print Statements
Keyboard Console	<p>1. Reading user input is functioning correctly Printed the character read from the stdin buffer onto the terminal.</p> <p>2. MotorInc and MotorDec update based on user input Printed the state of MotorInc and MotorDec constantly. While these variables are printed, we enter user input commands to make sure the variables are printing the correct updates.</p>	Output inspection & Print statements

Solar Panel Control	<p>1. Checking if the state needs to be updated Printed the command given and the associated state, checking if it is the consistent across the solar panel control sub functions.</p> <p>2. Checking MotorInc/MotorDec was read and duty was updated accordingly. 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>3. Testing the PWM interface See “Thruster Subsystem”</p> <p>4. Verifying that the caps for the PWM function is being maintained. 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>1. Verifying individual methods within the TaskQueue ADT 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>1. Verifying the correct initial values of the shared variables were being initialized correctly 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

7. PRESENTATION AND RESULTS

7.1 Output of the System



The screenshot displays three PuTTY terminal windows. The top-left window, titled '192.168.7.2 - PuTTY', shows a sequence of commands and responses: 'D' is entered, followed by 'Received command: D', then 'receiving response', and finally 'Received: A D'. The top-right window, also titled '192.168.7.2 - PuTTY', displays the output of the 'z' command: 'EARTH DISPLAY TERMINAL', 'Solar Panels: Deployed, Battery Level: 34660, Fuel Level: 100, Power Consumption: 1, Power Generation: 2', 'Battery Low: NO Fuel Low: NO', and 'Vehicle Response: A D'. The bottom window shows a C source file with code for handling satellite status and thruster commands. The code includes comments and function definitions like 'maskBit' and 'maskBit'. The status bar at the bottom indicates 'Windows (CR LF)', 'UTF-8', and 'INS'.

```
D
FUNCT0: send cmd D
FUNCT1: Received command: D
FUNCT0: receiving response
Received: A D
z
ConsoleDisplay: Showing Satellite Status...

// Fuel Level: %hu, "
// Power Consumption: %hu, "
// Battery Low: %s "
// Fuel Low: %s "
solarPanelString, batteryBuff[current],
battString, fuelString);
}

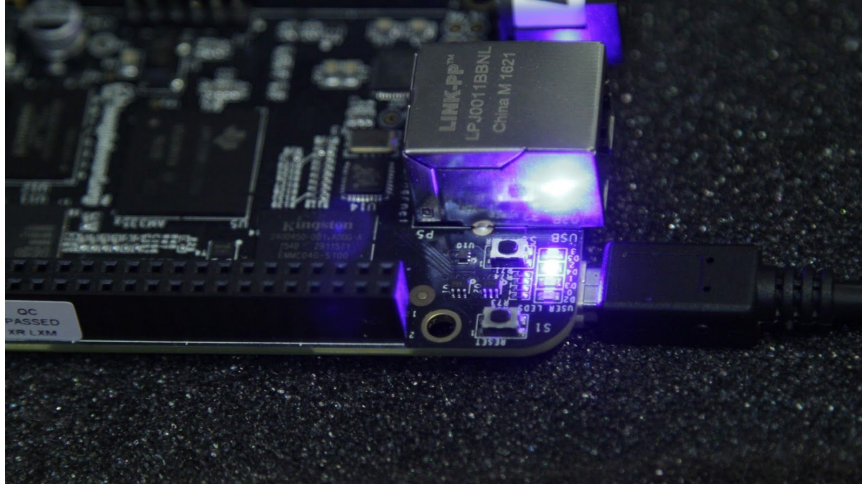
if (*response == 'A') {
    dprintf(fdi, "\nVehicle Response:
}

void maskBit(unsigned int *thrusterCommand)
// 0. Define a mask 1111111111110011
uint16_t MASK = 0xFFFF;

// 1. Mask the bit 2 and 3 to 0
*thrusterCommand &= MASK;
```

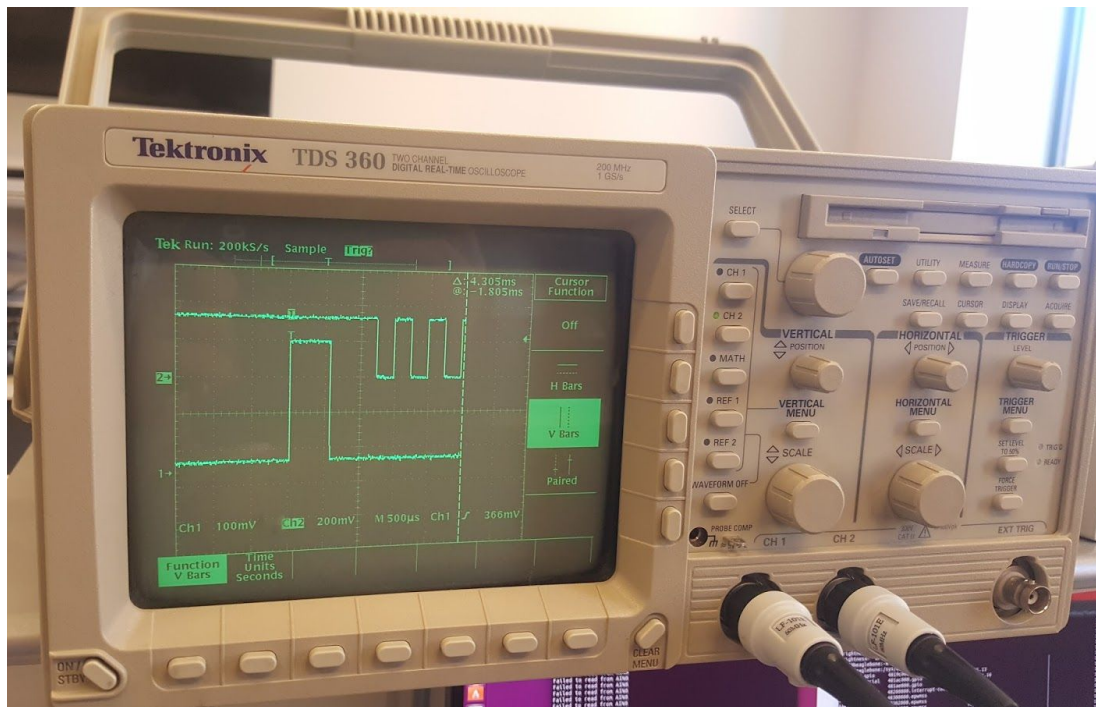
This screenshot shows three terminals that encapsulates the output of the system:

- (Top left) The Earth input terminal. Shown is an example of an input command “D” to the vehicle. The lines printed afterwards show the bidirectional communication between the satellite and the vehicle.
- (Top right) The Earth display terminal. The satellite and annunciation modes are being constantly streamed to the terminal. Also, notice that it also prints the vehicle response from the earth input terminal. This is to simulate information being relayed from the vehicle to the satellite to the earth station.
- (Bottom) The satellite terminal. Notice the input terminal took in the command “z” (an arbitrarily assigned command) and displayed the satellite status onto the terminal.



Another output of the system is the functionality of the LEDs to annunciate when fuel and battery are low. The picture shows the LED2 light up and flash when battery level reaches between 50% and 10%. However, since the battery level is a fixed number in the current phase, the LEDs must be set manually by the battery.

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.

How Long Each Task Takes in μ s								
	<i>power Subsystem</i>	<i>thruster Subsystem</i>	<i>satellite Comms</i>	<i>console Display</i>	<i>warning Alarm</i>	<i>solarPanel Control</i>	<i>vehicle Coms</i>	<i>console Keypad</i>
Trial 1	3380	928	4320	6640	5400	565	5820	2930
Trial 2	5100	932	4120	6040	4040	795	5660	2810
Trial 3	5340	912	4280	6840	5240	545	5100	2930
Trial 4	4980	916	4120	5960	480	575	5540	2910
Trial 5	5420	840	4160	5760	5680	645	5780	3090
Avg	4844	905.6	4200	6248	4168	625	5580	2934
Error rate (std dev)	837.42	37.59	93.81	465.96	2154.70	102.23	289.83	100.40
Major Cycle	29504.6		Delay Major Cycle	70495.4				
Minor Cycle	20196		Delay Minor Cycle	79804				

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

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 8 tasks;
init taskIndex to current index of array containing all 8 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. Referring back to the presentation and result, an abridged version of the table is provided for convenience:

How Long Each Task Takes in μ s								
	<i>power Subsystem</i>	<i>thruster Subsystem</i>	<i>satellite Comms</i>	<i>console Display</i>	<i>warning Alarm</i>	<i>solarPanel Control</i>	<i>vehicle Coms</i>	<i>console Keypad</i>
Avg	4844	905.6	4200	6248	4168	625	5580	2934
Error rate (std dev)	837.45	37.6	93.8	465.96	2154.7	102.2	289.8	100.4

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.

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.

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 discusses 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 II of this project in which we add battery level ADC functionality, vehicle communication, and a solar panel control driven with PWM.

(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 II. Moreover, the pseudocode demonstrate the various design decisions specific to each task.

(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.

(iv) The presentation and results section show how the output of the three terminals communicating with each other (Earth input terminal, Earth display terminal, and satellite terminal). Moreover, we saw the results from the tasks being measured on the oscilloscope. Thus, this section served to demonstrate the working aspects of our system.

(v) The errors section demonstrate the resolved and unresolved issues we came across during the implementation of the system. The resolved section goes through each task and discusses the main errors that were dealt with. The unresolved section examines the errors that were not resolved during the completion of this phase, namely, the deviation of times that were measured across multiple trials for each task and the battery level measurements not updating on the terminal.

11. CONCLUSION

In conclusion, this lab helped us gain a better understanding of pipes, cross terminal communication, analog to digital converter, voltage divider circuits, and PWM on the beaglebone. Moreover, we broadened our understanding of task queues by using a doubly linked list, and improved our understanding of pointers and UML diagrams which helped us evolve and refine our design, improving the safety and reliability of our 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.

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

13. CONTRIBUTION

Name	Contribution
Abdul Katani	Created the skeleton for the solar panel control and implemented it also contributed to thrustersSubsystem.c, powerSubsystem.c. 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
Radleigh Ang	Created skeleton for overall system, contributed to main.c and the dynamic queue, implemented satelliteComs.c, consoleDisplay.c, vehicleComs.c. Contributed to report.
Daniel Snitkovskiy	Updated powerSubsystem.c and thrusterSubsystem.c, added PWM and ADC functionality, and implemented the Dynamic Task Queue and the Startup Task.

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