Cornell University
Space Systems Design Studio
Electrolysis Propulsion Research Project

Cornell University
Cislunar Explorers

Alex Wong Spring 2017 End-of-Semester Report
CDHTC Subsystem

249 Upson Hall • Ithaca, NY 14853

Contents

Abstract

The Cornell Cislunar Explorer uses the Raspberry Pi Model A+ as its Flight Computer and uses a Linux-based operating system called Raspbian (default distribution on all Pi models). The Flight Software of the spacecraft leverages on the simplicity of Linux and the abundant libraries native to Linux and Python development. At its core, a thread manager is used to manage the multiple subsystems by using a simple queue-based scheduler with optional timing features. A dedicated command-handling thread is used to receive commands from the AX5043 transceiver chip and route it to its specified subsystem. A mode controller CSC works in conjunction with a health and status manager to provide protection for all of the mission's invariants. In addition, each specific subsystem has an interface file to interface with any external hardware and also any other subsystems that it has to communicate with.

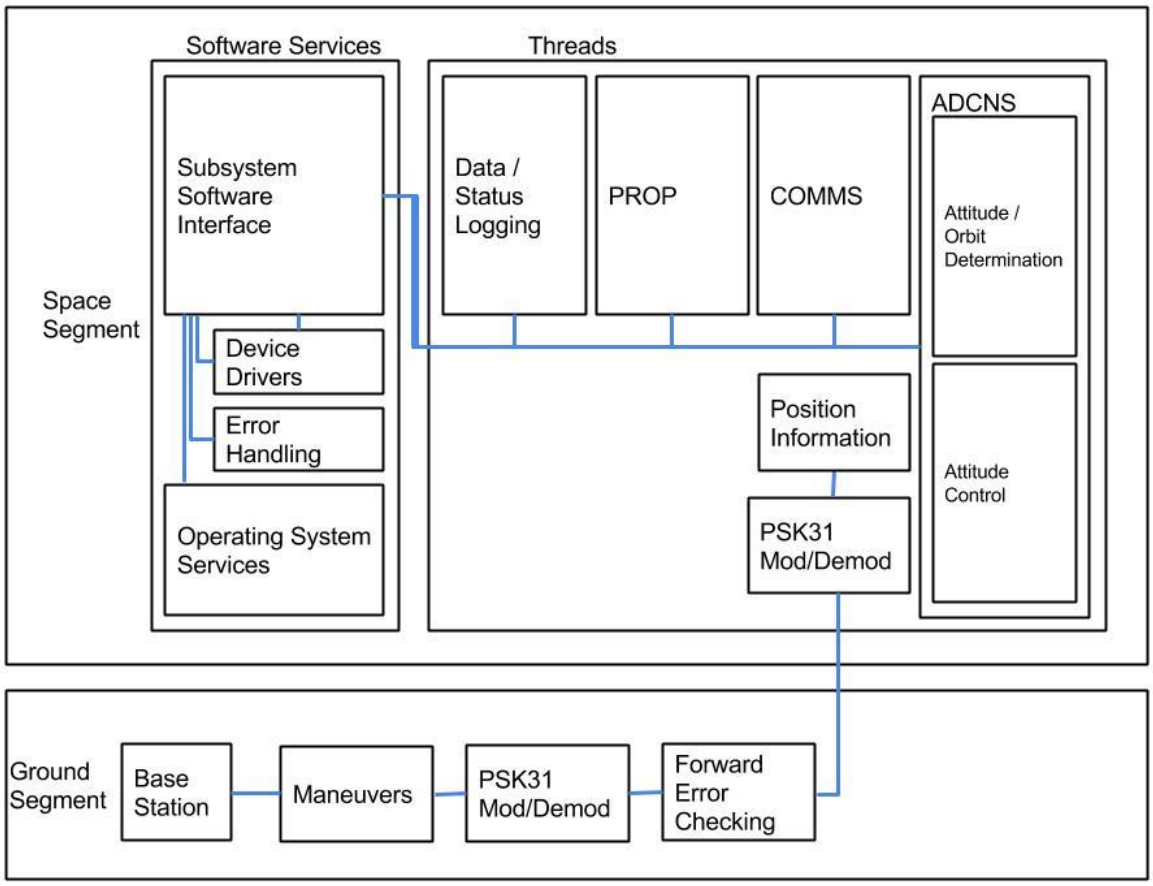

Figure 1: Basic Flight Software Interface Guidline

Csilunar Explorers Flight Software Background

The Flight Software is currently in the final stages of development with the core application-level operating system finished. The thread manager and scheduler can spawn new threads and is directly controlled from a command router. And new CSCs such as the mode controller were added to provide protection for the invariants set by the mission. Overall, the remaining steps

involve software interfacing with the communications (Transceiver chip specific) subsystem and the ADCNS subsystem as well as defining responses for the mode controller.

Thread Controller

This CSC acts as the central thread manager and scheduler for the application-level operating system. This CSC will interface directly with the command router to receive commands from the ground station and spawn new threads with function pointers and arguments specified by the commands. In addition, there are several special commands which perform specific flight computer related tasks such as resetting and powering off as shown below.

Power off command:
00110001 00100000 00110001 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000
00100000 00110000 00100000 00110001 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000

The thread manager works stand-alone in that all of the data flow comes out of the CSC. In other words, the individual subsystems must provide an API containing app-ids, commands, and arguments. The only flow out of this CSC is that it provides an initialization function to be used by the System Startup Controller to spawn all the initial threads: command reading thread, telemetry handling thread, and a health & status thread.

In terms of testing, this CSC has been unit tested using modular string testing. In other words, hard-coded commands were sent to the Pi to be parsed via the command router and then used to spawn new threads which would print an identification string. An example of a testing procedure is shown below.

• Start water electrolysis: Apid = 2, Opcode = 2
Packet Sent: 00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000
00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000
00110000 00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

Packet Received: 00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000
00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000
00110000 00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

Thread Function Pointer: propulsion_controller.electrolyzerControl(1,0)

System Startup Controller

This CSC acts as the startup function for the software. A callable main function will be housed here which calls all of the initializations for each CSC. This will then spawn all of the required

threads and also initialize any global variables or global data structures to be used by the software.

Health & Status Manager
This CSC collects all of the health & status onboard the spacecraft. This includes telemetry generated to be sent to the ground station as well as software messages which relay information about the architectural state of the software. In terms of functionality, this CSC currently only houses the most up-to-date information by using a C-type structure and simply updating fields accordingly. This CSC interfaces directly to the communication controller in order to send out telemetry. In terms of testing, unit testing was done to show that the structure could be updated to house the most up-to-date information and pass it to the communication controller via an inter-file queue.

Command Router
The command router handles all commands and parsing. This CSC will have a direct interface to the communication subsystem with a downward flow of received command from the ground station being sent to the command router. In addition, it will also have a direct interface to the thread manager with an upward flow of parsed commands being sent to the thread manager for spawning of command handling functions. In terms of functionality, the CSC mainly consists of two functions: a handler and a receive callback. The handler is a function which is used to continually read off an inter-file queue shared between it and the communication controller. This function is used by the thread manager's initialization function as it spawns a thread with a function pointer to it. The receive callback provides an API to the thread manager to use to receive commands from the command router.

In terms of testing, this CSC has been unit tested in conjunction with the thread manager as their operation is so closely coupled.

Communication Controller
This CSC acts as the higher level radio abstraction. This will provide an API to communicate with the ax5043 transceiver chip and perform general sends and receives. It also provides a receive callback function to be used by the command router to read off of their shared inter-file queue.

ax5032 Interface
This CSC acts as the direct interface to the ax5043 transceiver chip. This CSC houses all of the configuration functions for the ax5043 and performs all of the direct UART communication to this chip.

Power Controller
This CSC acts as the direct interface to the GOMSPACE power board. This CSC houses all of the configuration functions for the GOMSPACE board and performs all of the direct SPI communications to this chip.

Vision Controller
This CSC acts as the direct interface to the ACS subsystem.

All of these CSCs will be discussed in later sections in regards to the current design of the FSW which has mainly been designed to act as a software flatsat with a basic UI command interface and basic thread management.

Mode Controller
This CSC provides a protection service for the entire FSW. It houses a mapping between sensor values and its desired threshold for each mode of operation during the mission. In other words, during different stages of operation during the mission, different sensor values are expected. This CSC will check the mapping and determine if we are within a reasonable threshold. If so, everything works as expected. If not, it will execute a safety response depending on the severity of the conflict. This CSC interfaces directly with the health & status manager to determine if the values are safe. In addition, it sets flags in the CSCs to restrict function execution depending on the current mode.

The specific functionalities of the FSW will be discussed in later sections by exploring specifics concerning these CSCs discussed above.

Subsystem Command Interface

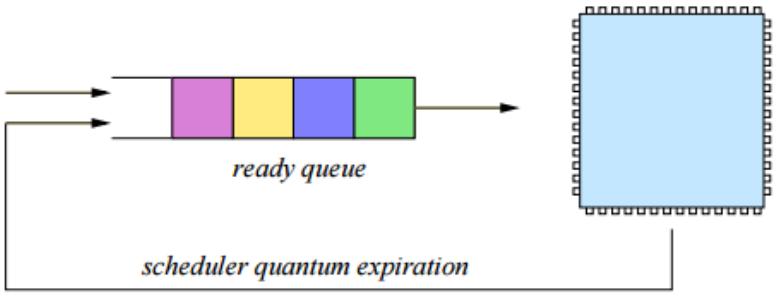

Figure 2: Basic queue-based thread scheduling algorithm

The basic command interface hasn't been changed since conception. It is currently implemented as a command line interface in the Raspberry Pi's command prompt. The current command interface is written in a single script which loops continuously on the Pi upon first initialization. Currently, the implemented commands deal with either propulsion, communication, power management, health & status checking, and mode transitioning. The most basic commands are as follows:

• Start water electrolysis: Apid = 2, Opcode = 2
Packet Content: 00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000 00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

• Fire the spacecraft main thruster: Apid = 2, Opcode = 3
Packet Content: 00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000

00110000 00100000 00110000 00100000 00110011 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

• Fire the Cold Gas Thruster: Apid = 2, Opcode = 1
Packet Content: 00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000
00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000
00110000 00100000 00110000 00100000 00110001 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

• Send status report: Apid = 3, Opcode = 0
Packet Content: 00110001 00100000 00110001 00100000 00110000 00100000 00110011 00100000
00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110011 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

• Obtain navigation fix: Apid = 3, Opcode = 1
Packet Content: 00110001 00100000 00110001 00100000 00110000 00100000 00110011 00100000
00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000
00110000 00100000 00110000 00100000 00110001 00100000 00110011 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110000 00100000 00110000 00100000

The current interface runs in the main function in the Thread Manger CSC which houses the basic thread managing and scheduling software used in the flight software. Upon system startup calling the main function, the software will initialize a command handling, health & status, and thread manager thread and idle them. In addition, it will continuously poll the user to enter one of the commands given above along with a time of execution. The thread manager will then either spawn a thread for execution or stall execution until the allotted time has run.

Flight Software Architecture

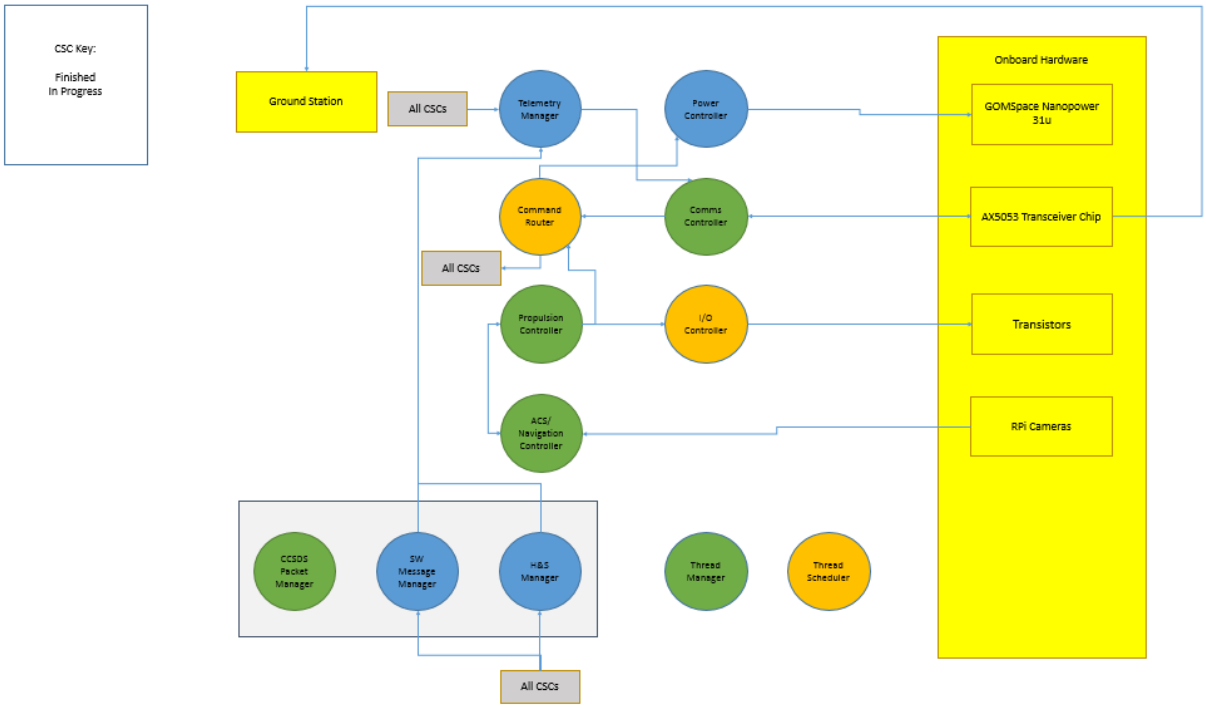

Figure 3: Flight Software Architecture Diagram

Thread Management and Scheduling

The thread management software is housed in the Thread Manager CSC and acts simply as a thread spawning and thread scheduling utility. This CSC heavily uses the Python Threading and Python Queue libraries to execute different commands in multiple threads and uses queues to simulate a simple scheduler.

Currently, the thread manager has a thread pool with a maximum thread count of 20. This means there can be 20 threads running concurrently at the same time. Despite this, we are currently only using single-core execution to simplify data-handling and reduce the risk of data coherency errors.

A queue data structure was implemented in order to receive user-entered commands and process them through spawning tasks. In other words, the data flow of the Thread Manager is it first initializes by spawning 20 utility threads in its thread pool. It then spin waits until the user enters a valid command where it will then add a thread to a thread queue to be pulled off by the main thread and be serviced.

Command & Data Handling

The command and data handling software is housed in the Command Router CSC as well as the Communications Controller CSC and Thread Manager CSC. These two CSCs work in

conjunction to receive commands from the on-board ax5043 transceiver chip and delegate work to the other subsystems.

The command handling software is mainly implemented in the Command Router CSC. This CSC has a global queue which the communications subsystem has access to and directly writes to upon receiving any messages by calling the Communications Controller CSC's receive callback function. The Command Router CSC will then read off this queue and service it by sending it to the private command queues for each subsystem. Because there is a dedicated thread for servicing received commands via the command router and the scarcity of commands in-flight, it is unlikely that this queue will be filled.

Continuing the use of CCSDS headers on all packets sent through the system, we will mainly deal with commands as a type-encoded message with the different 2-byte opcodes representing different, unique commands for each subsystem.

On the other side of this subsystem is the Communications Controller CSC which interfaces on a software layer to the ax5043. This CSC works by continuously polling the receive buffer on the ax5043 to read incoming messages. If there are any messages on the FIFO, it will simply copy the message into the queue to be processed by the Command Router CSC. This method of having a dedicated receive and dedicated service thread works well in this case by mitigating possible packet drops.

Communication Interface

The communications interface consists of the Communications Controller CSC and the ax5043 interface file. The former interfaces directly with the FSW and plays the crucial role of reading off the transceiver chip's receive FIFO as described above. The latter simply acts as the direct hardware interface to the ax5043. This software was originally written in C by Filipe, however, it has been translated to Python to run natively on the Raspberry Pi and interface better with the Flight Software by Alex Trestyn. The vast majority of this file consists of definitions of different variables for register use and also functions for reading and writing different registers of the chip.

Power Interface

The power interface consists of the Power Controller CSC which directly communicates to the GOMSPACE power board. Like the ax5043 interface file, most of this consists of low-level functions to directly configure the registers on the power board. This file is also used to directly send encoded commands to the power board.

Health & Status Manager

The health and status controller consists of a Health & Status Manager CSC which has access to all of the health and status structures for each subsystem. This CSC is responsible for keeping track of possible software hazards and exceptions for either debugging or for downlinking to a ground station.

Mode Controlling

The mode controller was the most recently developed CSC and allows the spacecraft to work in conjunction with the actual mission. The CSC has a dictionary which maps different I/O values and sensor readings to correct thresholds. It simply looks at the struct in the health & status manager which houses the most up-to-date information and compares it to the threshold values. Depending on how close the values are, it will either do nothing or perform a safety response which is yet to be defined. In addition, the mode controller houses a global variable which defines the current mode. All functions will then check against this mode to see if the function can be executed. In other words, the mode controller defines flags for each function which allows execution.

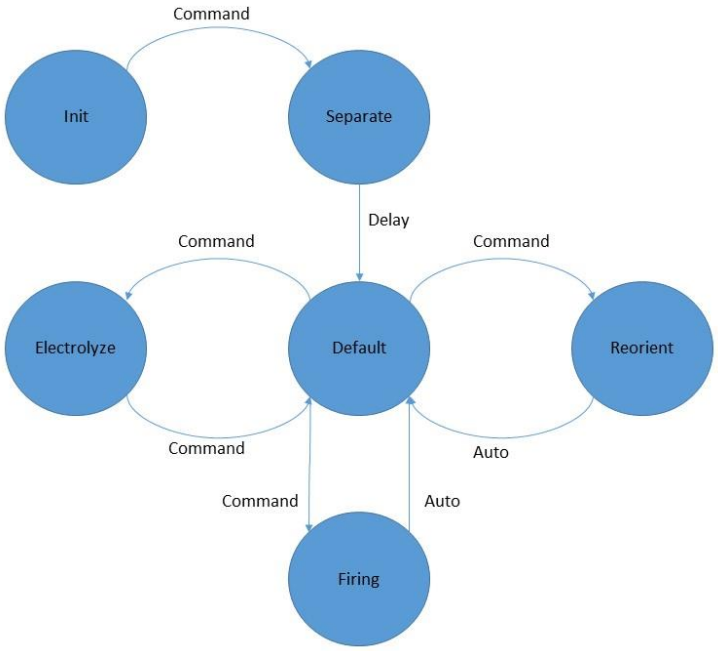

Figure 4: Mode Transition Diagram

Correct mode transitions have been tested as shown below:

Mode initialized to Init and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000 00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000 00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly switched to separate and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

00110010 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly switched to electrolyze and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000
00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000

Mode correctly switched to default and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110101 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110011 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly switched to reorient and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000
00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000

Mode correctly switched to default and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110101 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110100 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly switched to firing and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000
00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000

Mode correctly switched to default and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110101 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110010 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly switched to electrolyze and the following packet was sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110101 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000
00110100 00100000 00110101 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000

Mode correctly rejected firing command and correctly switched to default, the following packet was then sent.
Packet Sent:
00110001 00100000 00110001 00100000 00110000 00100000 00110010 00100000 00110000
00100000 00110000 00100000 00110001 00110000 00110010 00110100 00100000 00110000
00100000 00110000 00100000 00110010 00100000 00110010 00100000 00110000 00100000
00110000 00100000 00110000 00100000 00110000 00100000 00110000 00100000 00110000
00100000 00110000 00100000

Mode correctly rejected separate command and stayed in default.

Interfacing

Currently, a basic software flatsat has been constructed which has the capability to receive user commands through the Pi's command line and spawn threads with a time delay to service these commands. These commands are sent directly to the respective subsystem's command queue for servicing.

While this software hasn't been integrated and fully-tested with the hardware due to the lack of a physical electrical harness for most of the semester, there are currently placeholder GPIOs in the code which can be readily changed for use with any reasonable harness. In addition, as most of the GPIOs will actually be used from the power board rather than the Pi, these don't have to be as rigorously tested as they rely on byte-encoded packets which have already been tested. In other words, the main interfaces have already been well developed and tested as SPI is working on the Pi and the commands to the power board have also been already tested.

Division of Work

The vast majority of the work done over the semester was to interface the various CSCs to provide an application-level OS to act as the FSW. This was mostly done in the beginning of the semester. Afterwards, rigorous unit testing was done for each CSC using modular string testing and basic verification methods. This would reveal many bugs in the software which would then be subsequently fixed. Functional-level testing would then be done on the FSW which involves multiple different CSC working in conjunction. The end of the semester mainly consisted of development of the mode controller as well as kernel-level modifications to the underlying Raspbian OS. In terms of division of work, all of the software except the ax5032 code and the power subsystem code was developed by Alex Wong.

Future Work

There are a variety of goals for the future in order to complete the FSW and prepare it for the spacecraft. They will be split by semesters to provide a sample time-frame.

Summer Goals

Unit testing:

More unit testing of each individual CSC is required especially once it is interfaced with the Flatsat. This testing will be similar to what has already been done (ie: modular string testing). It may be helpful to write a script which generates random tests and logs results in a text file in order to amass a large number of test results.

Mission-Ops software development:

The software must be interfaced with the specific mission plan outlined for the spacecraft. In other words, the mode controller must be further improved to hold threshold values for each mode and also provide correct usage depending on mode (currently undefined behavior placed by software lead). In addition, correct safety responses must be defined and implemented in software (ie: conflict in mode controller dictionary leads to transition from certain mode along with data flush, etc..).

Add basic commands:

Basic commands such as reset and power-down are finished but there may be other basic commands which may need to be implemented. This is up to the mission-ops lead.

Startup scripts and kernel modification:

SH scripts must be made so that the Pi initializes the FSW correctly. Currently a script loops the execution of the main thread manager script. However, if the spacecraft were to power-off there aren't any scripts which can handle this. In addition, the kernel of the underlying Raspbian OS should be modified for the FSW needs (ie: removing several unnecessary drivers, etc.).

Interface with power board:

This is mostly done, but the interface should be better defined.

Testing outline by Dean Larson:

A test outline has been provided by Dean Larson which involves several testing stages. Currently the structured walk-through has been finished but the rest of the schedule is TBD.

Data Verification and Validation: 15 May 2017. Verifying the all internal data has been entered correctly. Any data used in the code is validated against the appropriate, real-world standard. All results to be documented.
Individual module testing: 19 June 2017. Verifying all individual software modular functions work correctly by comparing the expected computed result based on known input data, to the results generated by module under test. All results to be documented.
Software merge and cross-communications check: 3 July 2017. All software is loaded and executing as it will be in flight. Verify there are no problems with communications interfaces between the Command & Control, Optical Navigation, and Communications software.
Modular String Testing: 1 August 2017. Verifying that all possible commands and paths through the code work as expected, starting with the commands expected to be used most often. This involves determining the path to be checked, determining realistic input values, and calculating the expected result at the end of the path without

using the software under test. A verification test is then run to compare the code output with the calculated result. All results to be documented.

Pre-Vibration Test Verification: 1 September 2017. Execute each planned post-vibration test once to ensure that 1) we can expect the tests to be successful after vibration testing and there are no software issues to prevent successful post-vibration-test testing and 2) no corrections made to the software to date have unintentionally introduced errors the software. All results to be documented.

Ax5043 python code & comms integration:
The translated Ax5043 code has been handed off to Alex Trestyn to work on. This must be finished and integrated with the rest of the FSW.

ADCNS integration:
This will mainly be handled by Doga and her team. On the FSW side, her functions must be placed in a thread which will run automatically at pre-defined rates.

Memory storage:
This is a large milestone which must be accomplished. We have to decide on either using the Pi's flash or an external SD card to store data. Ultimately, a Data Logger CSC must be made to handle how we will store data and in what format. There are currently very few design constraints in this implementation.

Fall Goals

Stress-testing Pi:
The FSW should be stress-tested while running on the PI to determine some baseline operation benchmarks during the mission.

Interfacing with new hardware:
Several new pieces of hardware were acquired during this semester and probably in the next few months. These will need to be interfaced.

Handling radiation-induced faults:
Currently, there are no methods to deduce faults on the spacecraft side. While CRCs are attached to each incoming and outgoing packet, once the packet reaches the spacecraft, it is hard to tell if particular packets are corrupted or not. Simple checks include parity checking which has a 1-step accuracy (can only determine if 1 bit-flip occurred). More complicated measures include Hamming Codes.

Ground station software:
A CLUI should be developed to act as the ground station software. Time permitting this should be updated to an executable GUI.

Kernel optimization:
The kernel of the underlying Raspbian OS should be modified for the FSW needs (ie: removing several unnecessary drivers, etc.).

End-to-end Flatsat testing:
Involves full testing of the FSW as well as the rest of Dean Larson's test outline. **Pre-Vibration**

Test Verification: 1 September 2017. Execute each planned post-vibration test once to ensure that 1) we can expect the tests to be successful after vibration testing and there are no software issues to prevent successful post-vibration-test testing and 2) no corrections made to the software to date have unintentionally introduced errors the software. All results to be documented.

After Vibration Testing: 1 October 2017. Execute all scheduled post vibration testing verification tests. All results to be documented.

Command Data Verification: Verify that all Commands are transmitted correctly and commands with mistakes in them are rejected with an error message to the operator. All results to be documented.

Before Releasing Spacecraft to NASA: Redo all reliability tests and all post-vibration-testing tests at least once. All results to be documented.

References

All code will be housed on a private repository in GitHub. Currently, the repository isn't well structured. I will make sure to reorganize the repository and push the new code onto there once that is done.

- https://github.com/cislunar-explorers/avionics
- https://github.com/cislunar-explorers/comms
- https://github.com/cislunar-explorers/propulsion
- MCOTEA manual