**The Thesis Committee for Shaina Ashley Mattu Johl**
**Certifies that this is the approved version of the following thesis:**

# A Reusable Command and Data Handling System for University CubeSat Missions

**APPROVED BY**
**SUPERVISING COMMITTEE:**

E. Glenn Lightsey, Supervisor

Wallace T. Fowler

# A Reusable Command and Data Handling System for University CubeSat Missions

**by**

**Shaina Ashley Mattu Johl, B.A.S.**

**Thesis**

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

**MASTER OF SCIENCE IN ENGINEERING**

**The University of Texas at Austin**
**December 2013**

# Acknowledgements

I would first like to thank my advisor Dr. Glenn Lightsey for his instruction, encouragement, and guidance on my graduate work these past years. I would like to thank my second reader, Dr. Fowler for providing me with insight that helped me finish this thesis. I would also like to thank my fellow lab mates at the Texas Spacecraft Laboratory. I look forward to coming to the lab every morning knowing I have supportive colleagues with whom I enjoy exploring new ideas and solving hair-pulling software bugs, all the while sharing some great laughs. I look forward to celebrating successful satellite missions in the upcoming years with all of you.

Finally, I would like to thank my parents for supporting my move to Texas and my passion for aerospace engineering. Your constant love and encouragement means the world to me.

# Abstract

## A Reusable Command and Data Handling System for University CubeSat Missions

Shaina Ashley Mattu Johl, M.S.E
The University of Texas at Austin, 2013

Supervisor: E. Glenn Lightsey

A Command and Data Handling (C&DH) system is being developed as part of a series of CubeSat missions being built at The University of Texas at Austin's Texas Spacecraft Laboratory (TSL). With concurrent development of four missions, and with more missions planned for the future, the C&DH team is developing a system architecture that can support many mission requirements. The presented research aims to establish itself as a reference for the development of the C&DH system architecture so that it can be reused for future university missions. The C&DH system is designed using a centralized architecture with one main flight computer controlling the actions and the state of the satellite. A Commercial Off-The-Shelf (COTS) system-on-module embedded computer running a Linux environment hosted on a custom interface board is used as the platform for the mission software. This design choice and the implementation details of the flight software are described in detail in this report. The design of the flight software and the associated hardware are integral components of the spacecraft for the current missions in the TSL which, when flown, will be some of the most operationally complex CubeSat missions attempted to date.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Small satellites have been an emerging class of spacecraft in the satellite industry for the past several years. Satellites classified under this title are considered those with a mass of less than 180 kilograms, and include commonly named satellite terms such as micro- (10-100 kg), nano- (1-10 kg), and picosatellites (0.001 – 1kg) (National Aeronautics and Space Administration 2013). There has been growing interest in using small satellites for civil, commercial and military space purposes. A study identified 33 potential markets for low-cost small satellites in these sectors, and six markets that are likely near-team users (Foust 2010):

- Military science and technology
- Intelligence, surveillance, and reconnaissance
- Remote site communications
- Polling of unattended sensors
- High-resolution Earth observation
- Landsat-class data for environmental monitoring

Technological advancement over the past decades has allowed the size of the payloads and instruments for space missions to continue to decrease (Toorian, Diaz and Lee 2008). This has made the use of the CubeSat form factor, a type of small satellite on the smaller end of the size scale, more common. This chapter introduces the CubeSat and its C&DH subsystem. The motivation behind this thesis and its structure is then presented.

## 1.1 CubeSat Form Factor

The CubeSat was developed in 1999 by California Polytechnic State University's Multidisciplinary Space Technology Laboratory (MSTL) and Stanford's Space Systems Development Laboratory (Toorian, Diaz and Lee 2008). A satellite is designated a CubeSat if it meets the requirements outlined in the CubeSat Design Specification (California Polytechnic State University 2009). A 1U CubeSat form factor is 10 cm x 10 cm x 10 cm. However, CubeSats can be 1U, 2U, 3U, 6U or other sizes, but must weigh

less than 1.33 kg per U under the current standard. The standardized CubeSat deployment system is called the Poly Picosatellite Orbital Deployer (P-POD). The P-POD acts as the interface between the launch vehicle and the satellite, and is capable of carrying up to 3 1U satellites (or 1 3U satellite) in a single deployer.

Figure 1. P-Pod Deployer

CubeSats provides several favourable attributes over their larger counterparts, namely development time to launch and cost. CubeSats can be developed faster than larger spacecraft. CubeSat missions can go from conception to delivery in as little as a few years. This is partially due to CubeSats having less complex missions and shorter lifetimes. Another contributing factor is that CubeSats can be assembled using COTS components, thus eliminating the time that would be required to design and test components that would be fabricated in-house, and only leaving the time needed for proper interfacing with the COTS components.

CubeSats also have a lower cost for access to space than larger spacecraft. Due to their small size, CubeSats can be launched as secondary payloads on launch vehicles dedicated to a larger satellite, or by integrating the CubeSat into the larger satellite and being launched from it. There are currently a number of programs that provide ridesharing for CubeSats, such as the University NanoSatellite program (UNP), and NASA's CubeSat Launch Initiative (CSLI).

The ability to quickly develop and deploy CubeSats, along with significant flight heritage, makes them an attractive form of spacecraft for many types of missions. The

first CubeSat missions were launched in 2003, and since then there has been over 70 U.S. companies, 50 U.S. universities and 41 foreign universities that have worked on building and flying these spacecraft (National Reconnaissance Office 2013).

### 1.2 Command and Data Handling Subsystem

As the CubeSat industry continues to grow, there will be a larger demand for CubeSats to handle more complex mission and operational requirements. These requirements flow down to affect the Command and Data Handling (C&DH) subsystem of the satellite. The C&DH subsystem acts as the "brain" of the spacecraft. It consists of the hardware, including the main flight computer, and the software that controls the operations of the satellite.

### 1.3 Motivation

The goal of this thesis is to document the work done in developing the C&DH subsystem used for the current missions in the Texas Spacecraft Lab (TSL) at the University of Texas at Austin (UT). The TSL has flown two satellites, and is currently working on three additional satellites that will use the C&DH system. The experience gained by past missions has made obvious the need to develop a re-usable C&DH system for CubeSats. This thesis aims at describing this effort and promoting the reuse of the C&DH system. The thesis acts as a guide for the design, implementation, and testing process of the components that comprise the C&DH system, with an emphasis on the development of the flight software. Prior to this research, the TSL did not have a reusable architecture for the C&DH system. The research done for this thesis aims to establish a standard for the development of the C&DH system architecture so that it can be reused for all future TSL missions.

### 1.4 Thesis Structure

The layout of the thesis is as follows. Chapter 2 introduces the reader to the TSL at UT and provides background information on past and current missions designed and supported by the lab. The design requirements of the common C&DH system are

introduced in Chapter 3, and a description of the C&DH hardware and software is presented. The current system in development is also compared to the C&DH systems from previous missions in the TSL, namely FASTRAC and Bevo-1, which served both as a starting point for the design of the current system and as a knowledge bank which provided guidance throughout the development process. Chapter 4 focuses on the architecture of the flight software currently being implemented and tested for the current TSL missions, as well as the C&DH software infrastructure put in place to aid in the development of the code. Chapter 5 then describes the main features of the implementation of the flight software. Information on the methods and the results from testing the C&DH software, including the flight software running on the integrated satellite, is given in Chapter 6. Finally, recommendations are made on what the focus should be on for future work on the C&DH system and presented in Chapter 7. Collectively, the topics discussed in this thesis were steps taken in the design, implementation and testing of the C&DH system and flight software developed for current and future missions in the TSL.

# Chapter 2: Background

The C&DH system being developed, while it is the focus of this thesis, is only one key component in the satellite design work being performed in the Texas Spacecraft Laboratory at the University of Texas at Austin. The work performed in the TSL involves the application of skills and knowledge from many different fields of engineering, all of which contribute to the development of small satellites.

## 2.1 Texas Spacecraft Laboratory

The technical staff of the TSL at UT-Austin consists of a group of roughly thirty graduate and undergraduate students who work together on the lab's satellite missions. The students are involved in all steps of the satellite fabrication process including the design, build, test and operation of the spacecraft.

Since 2007, the lab has launched three satellites into orbit, FASTRAC-1, FASTRAC-2 and Bevo-1. The TSL is currently working on three additional satellites that will fly within the next two years, Bevo-2, ARMADILLO (Atmosphere Related Measurements and Detection submiLLimiter Objects) and RACE (Radiometer Atmospheric CubeSat Experiment).

## 2.2 Past Missions

The TSL is a multi-purpose facility. Here, students combine past experience, heritage designs, COTS hardware, and new ideas to develop concepts for new satellites and missions. In an environment where students graduate and take their knowledge gained at UT-Austin with them, it is important to make provisions for ensuring that the lessons learned throughout the satellite development process are recorded. Documenting lessons learned is critical for maintaining progress in a lab where there is a large turnover rate every semester. However, this can be challenging in a university setting where there is less manpower and monetary resources, and generally a shorter project lifetime than in industry. Keeping accurate records and preserving knowledge through documentation is especially critical for software implementation in the lab as it is very difficult to read and

understand someone else's code. The documentation of the software from the previous TSL missions was instrumental in providing a starting point for the development of the current C&DH system. The design of the software for the current TSL missions began with an analysis of the lessons learned from FASTRAC and Bevo-1. A brief overview of these past two missions will be given in the proceeding sub-sections.

### 2.2.1 FASTRAC

FASTRAC, (Formation Autonomy Spacecraft with Thrust, Relnav, Attitude, and Crosslink) was a satellite built by the TSL for which work began in 2003. It was the winning entry of the University Nanosat-3 Competition in 2005. The University NanoSatellite Program, sponsored by the Air Force Research Laboratory (AFRL), gives university students hands-on experience in designing and constructing satellites in a two-year concept-to-flight-ready competition. While working with personnel at AFRL, the two FASTRAC satellites, known as Sara Lily and Emma, were prepared for flight after several component and hardware redesigns and modifications, as well as extensive environmental testing which lasted until February 2010 (Munoz, Hornbuckle and Lightsey 2012). FASTRAC was successfully launched in November 2010, and the separated satellites are currently still operating in orbit. As of April 2012, over 16 000 beacon messages as well as telemetry data such as health, GPS, thruster and IMU messages had been received by the Operations team.

The FASTRAC project consisted of two nearly identical NanoSatellites, as shown in Figure 2, with three primary mission objectives. The FASTRAC satellites are the two stacked hexagonal objects in the foreground of the figure.

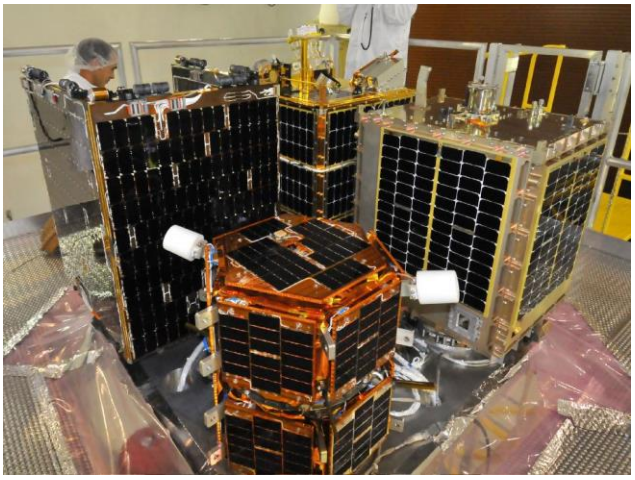

Figure 2. FASTRAC (Center) mated onto the adapter plate of STP-S26 (Hernandez, 2011)

The first mission objective entailed establishing an autonomous crosslink between the two satellites. The second objective involved performing on-orbit real-time GPS relative navigation. The final objective demonstrated autonomous thruster firing logic based on the on-orbit real-time single antenna GPS attitude determination solution (Munoz, Hornbuckle and Lightsey 2012).

The FASTRAC mission provided the TSL with valuable experiences and lessons learned on the development, implementation and operation of student-built satellites.

### 2.2.2 BEVO-1

The Bevo-1 satellite was built by the TSL as the first of four missions as part of the LONESTAR (Low Earth Orbiting Navigation Experiment for Spacecraft Testing Autonomous Rendezvous and docking) program. This program, sponsored by NASA's Johnson Space Center (JSC), is a collaborative project between the TSL at UT-Austin and the AggieSat Lab at Texas A&M University (Department of Aerospace and Engineering Mechanics at the University of Texas at Austin 2013). Its aim is to promote aerospace engineering education and to provide an opportunity for research in low-cost autonomous rendezvous and proximity operations techniques (AggieSat Lab 2010). Each mission is comprised of one satellite built by each school with the mission objectives

increasing in complexity. The first three missions lead up to the final mission objective of demonstrating autonomous rendezvous and docking between the two cooperative spacecraft. Each of the missions of the program demonstrates new technologies and operations that are necessary to achieve the final mission. The mission objective for Bevo-1 was to collect and downlink two orbits of GPS data to validate NASA JSC's DRAGON (Dual RF Astrodynamic GPS Orbital Navigator) GPS receiver (Johl and Imken 2012). Bevo-1 along with AggieSat2 by Texas A&M, depicted in Figure 3, were launched together aboard the Space Shuttle Endeavour in July 2009. Bevo-1 is shown on the left of the figure, and AggieSat2 is on the right of the figure. The two satellites were designed to push apart and separate completely from each other upon launch. However, they failed to separate upon deployment. Bevo-1 never powered on, and contact was never established with the satellite. The satellites reentered in early 2010. Despite the failure to achieve the mission objectives, Bevo-1 provided valuable experience and perspective on best engineering practices in a university low budget hardware environment.

Figure 3. Bevo-1 and AggieSat2 Satellites (AggieSat Lab 2010)

## 2.3 CURRENT MISSIONS

The TSL is currently working on three 3U CubeSat missions simultaneously, Bevo-2, RACE, and ARMADILLO. The design architectures of these three satellites are very similar. The structural layouts of Bevo-2 and ARMADILLO consist of three modules, the bus module, the ADC module, and the payload module. The bus modules of both satellites will be identical, but there will be differences in some components for the ADC and payload modules, as they are designed to meet very different requirements. RACE is also a 3U CubeSat, but with 1.5U dedicated to the radiometer instrument provided by JPL. A brief overview of these three missions is provided in this section.

### 2.3.1 BEVO-2

Bevo-2 is UT-Austin's satellite as part of the second mission of the LONESTAR program.

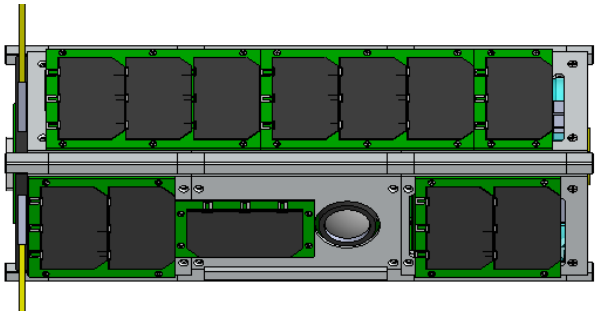

Figure 4. CAD Model of Bevo-2 Spacecraft

The goal of this second mission is to launch two satellites together, Bevo-2 and AggieSat-4, which will separate in orbit and perform proximity operations. For Bevo-2 specifically, the mission objectives are as follows (Texas Spacecraft Laboratory 2011):

- Evaluate sensors including but not limited to GPS receivers, IMUs, rate gyros, accelerometers
- Evaluate Reaction Control System (RCS).
- Evaluate GN&C system including guidance algorithms, absolute navigation, and relative navigation

- Evaluate communications capabilities between the two spacecraft and from each spacecraft to their ground stations.
- Evaluate capability to take video.

AggieSat-4 is an approximately 50 kg NanoSatellite built by Texas A&M. Bevo-2 will be stowed inside AggieSat-4 during launch. A JAXA (Japanese Aerospace Exploration Agency) airlock aboard the International Space Station (ISS) will be used to release AggieSat-4 into low Earth orbit, which will then discharge Bevo-2 (Kjellberg 2011). The Concept of Operations for Bevo-2 is shown in Figure 5.

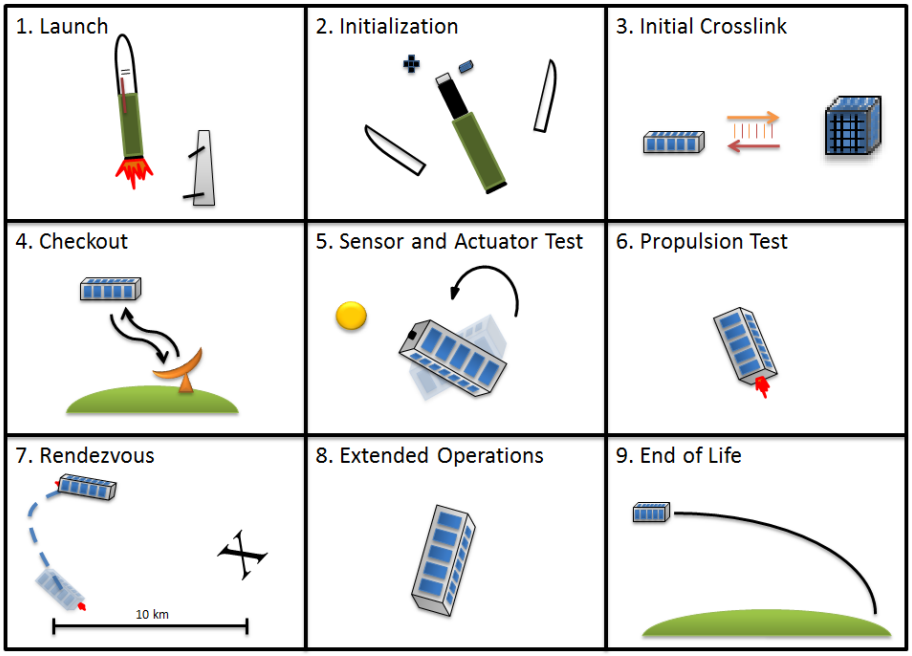


Figure 5. Illustrative View of Bevo-2 Concept of Operations (Texas Spacecraft Laboratory 2011)

Bevo-2 and AggieSat-4 will be launched into ISS orbit, and will have an estimated lifetime of 6 months. Upon separation, the two satellites will perform crosslink communication of GPS data.

Bevo-2 features an in-house miniaturized star tracker that will also be used to take images of AggieSat4.

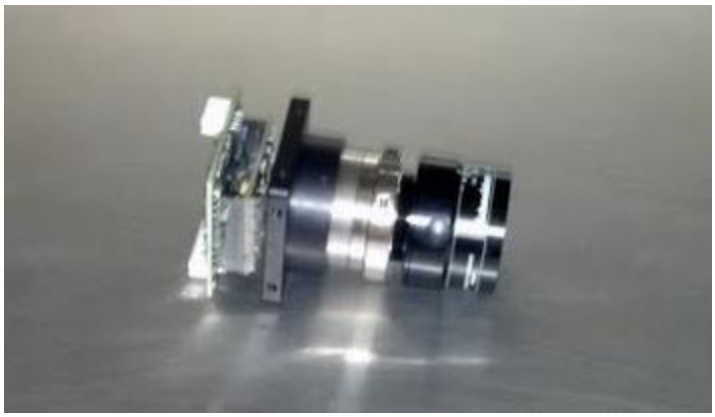

Figure 6. Star Tracker Camera to be used on Bevo-2 and ARMADILLO Spacecraft

Bevo-2 also features the same six degree-of-freedom ADC module as ARMADILLO, which, after the checkout stage, will be characterized by performing a series of sensor and actuator tests. Another component is an in-house designed cold gas thruster which will be used to perform a rendezvous maneuver to place the satellite in a pre-defined state in space. The LONESTAR-2 mission is planned for flight in 2014.

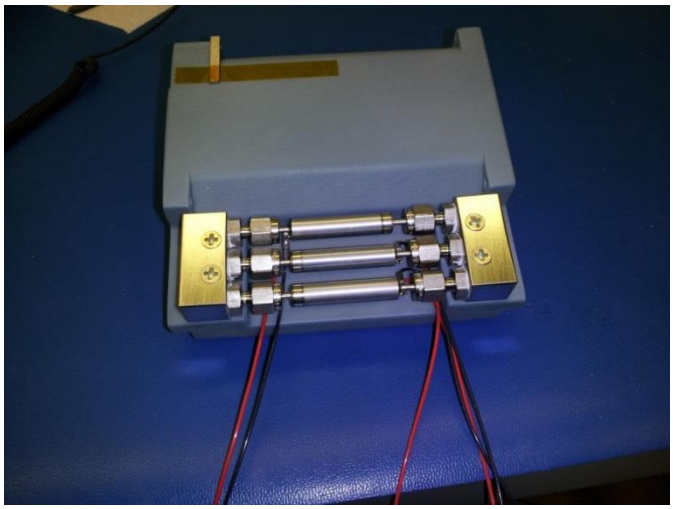

Figure 7 Cold Gas Thruster for Bevo-2 Spacecraft (Lightsey 2013)

### 2.3.2 RACE

RACE is a 3U CubeSat developed in collaboration with JPL, who will be providing the radiometer payload. The TSL's involvement in the RACE mission began in

April 2013. The primary mission objectives of RACE are to advance the technology readiness level of the radiometer instrument, thereby reducing the risk for future missions. The system includes a 35 nm Indium Phosphide low noise amplifier (LNA) at the front-end, and will be the first millimeter wave radiometer to be flown on a CubeSat (Jet Propulsion Laboratory 2013). Demonstrating the radiometer on a small and cost effective CubeSat will advance Earth science measurements for future missions. In addition, the data collected from the instrument will be used with weather prediction models to advance existing Earth climate system models.

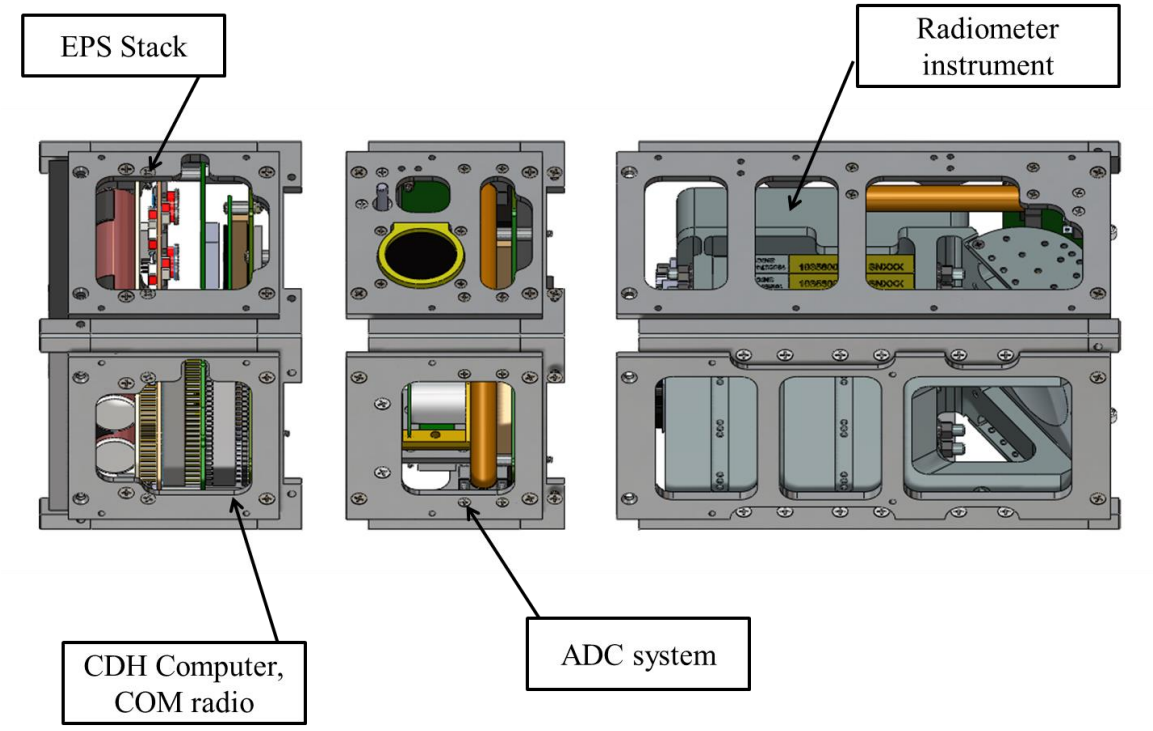


Figure 8. Modular CAD Model of RACE Spacecraft

While JPL is responsible for delivering the radiometer, the TSL is responsible for building and testing the CubeSat bus, and managing the payload integration. Upon launch in 2014, the TSL will also manage the ground segment, including data collection. RACE will be launched into an ISS altitude orbit, and will have a planned operational lifetime of approximately 6 months.

### 2.3.3 ARMADILLO

ARMADILLO is the TSL's winning entry into the UNP-7 competition sponsored by the US Air Force. The primary objective of this mission is to characterize sub-millimeter diameter dust and debris particles that are present in low Earth orbit. ARMADILLO features a Piezoelectric Dust Detector (PDD) being built by Baylor University that will detect the particles upon impact with the instrument. The impact will produce an electric charge which will be recorded and stored by the PDD until the C&DH computer queries the instrument. The data is then post-processed and provided to atmospheric models which will improve the knowledge of the sub-millimeter space debris environment (Brumbaugh 2012).

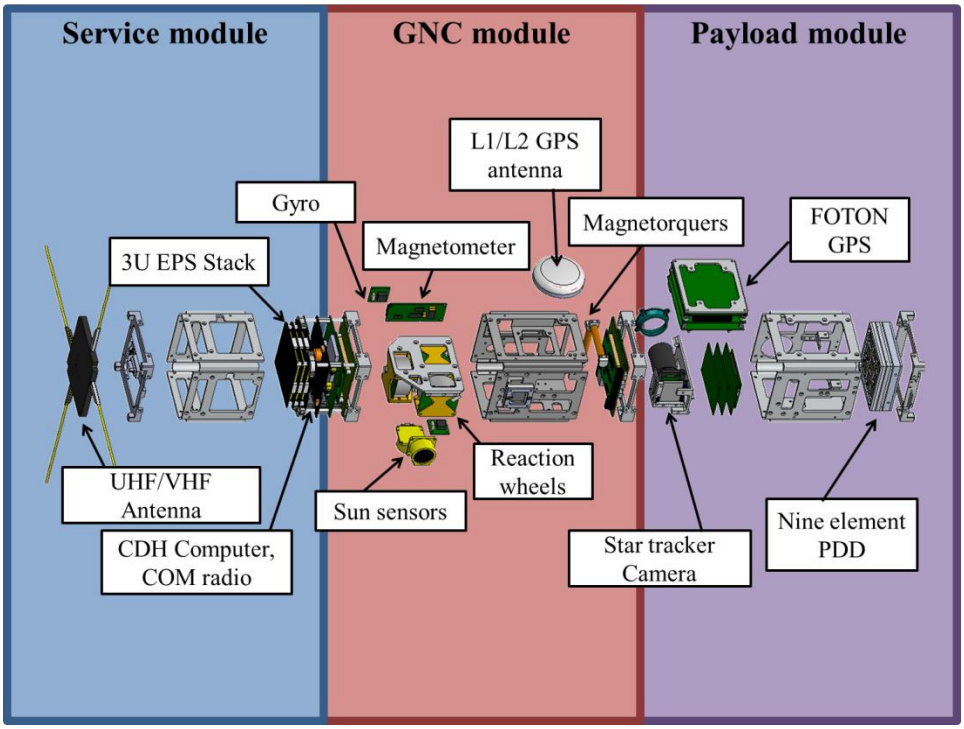


Figure 9. Exploded View of ARMADILLO (Brumbaugh 2012)

The secondary objective of ARMADILLO involves using a dual-frequency GPS receiver designed at UT-Austin called the FOTON (Fast, Orbital, TEC, Observables, and Navigation) to measure GPS radio occultations for studying the Earth's ionosphere. The

data collected by the FOTON will help increase the understanding and forecasting of space weather.

ARMADILLO also features a six degree-of-freedom ADC module developed in-house at TSL that provides arc-minute level 3-axis attitude control. The ADC will provide the pointing accuracy required by the PDD for data collection. The concept of operations for ARMADILLO is shown in Figure 10. As shown, the current plan is for ARMADILLO to be launched into an orbit with an altitude of 500 km, and to have an estimated mission lifetime of 2 years.

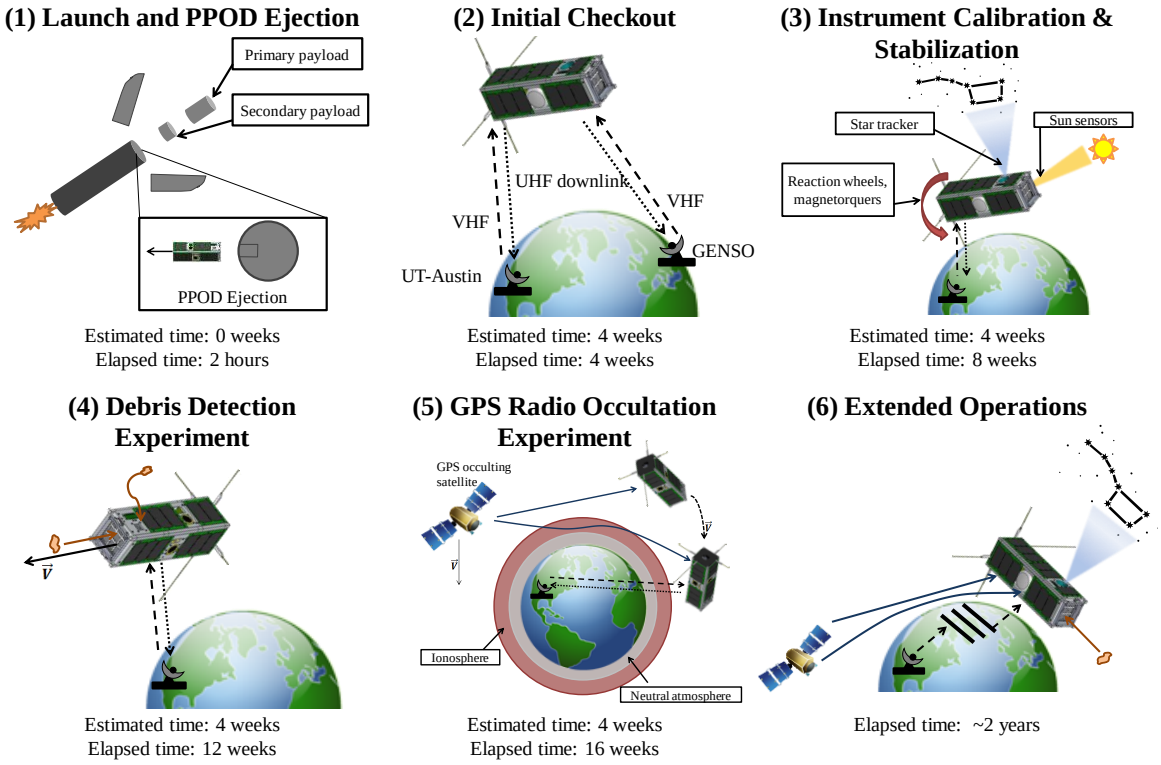


Figure 10. Illustrative View of ARMADILLO Concept of Operations (Brumbaugh 2012)

### 2.3.4 Current Status of Missions

As of fall 2013, Bevo-2 is currently in the integration and testing phase of its development cycle. The final preparations for running integrated tests on the flight version of the satellite are underway. The satellite is scheduled to be delivered to NASA in the first quarter of 2014, and will be launched later in 2014.

RACE is also manifested for launch in 2014 through NASA's CubeSat Launch Initiative program. All of the C&DH software and most of the overall Flight Software (FSW) testing being performed for Bevo-2 is directly applicable to the RACE mission. The EM radiometer was delivered by JPL to the TSL in November 2013. Full integrated satellite testing is underway.

Code development and testing is continuing to progress for the two ARMADILLO payload systems, the FOTON and the PDD. Certain components of the flight hardware need to be acquired before a flight build can begin, such as the UHF/VHF radio and the Electrical Power System (EPS) system. The ARMADILLO mission was manifested by the CubeSat Launch Initiative program for a launch in 2015.

In terms of the C&DH system for these missions, a version of the software common to all three satellites has been written. The software running on the integrated satellite for Bevo-2 has been tested through functional tests, and command execution tests. The next software version will be considered FSW for Bevo-2 once day-in-the-life testing has been completed. Specific subsystem-C&DH software interfaces for the respective payloads are needed in order to use this next version as the RACE and ARMADILLO missions' FSW.

# Chapter 3: Components of C&DH System

Even though Bevo-2, RACE and ARMADILLO have very different mission requirements, all subsystems developed in the TSL, with the exception of the payloads, are designed such that they are capable of completing all three missions' objectives. As the three satellites are scheduled to be delivered in the upcoming few years, this simultaneous development adds to the existing challenges of a student-run lab such as manpower, time, and resource constraints. Eliminating unnecessary re-engineering by developing modular subsystems that can be used on a variety of TSL CubeSat missions is a valuable concept to implement. Thus, the developed C&DH system discussed in this thesis was designed to be used for all three current missions of the TSL.

The C&DH system requirements common to all current missions are outlined in this chapter. These requirements were the driving force behind the selection of the C&DH hardware. The C&DH hardware used for the current missions in the TSL have not been used on a previous mission in this lab, as Bevo-2, RACE and ARMADILLO are the most complex missions the TSL has been involved with to date. Because of the increased complexity, higher computing requirements and more sophisticated software than previously used are needed to successfully complete each mission's requirements. The decisions for the choice of the flight computer's operating system and FSW are discussed in the later sections of this chapter. Also, a comparison between the new C&DH system architecture and those of the previous Bevo-1 and FASTRAC missions is made and discussed in this chapter.

## 3.1 Current Mission Requirements

As part of the mission design process for Bevo-2, RACE, and ARMADILLO, mission statements, objectives, and requirements were formed stating the goals of the missions and the criteria that define mission success. In addition, a set of requirements for each subsystem was formed and documented in a mission requirements verification matrix (RVM). These subsystem requirements were created to ensure that the higher-level mission requirements were met. The C&DH subsystem has five subsystem

requirements that are identical for all three missions. The requirements, the rationale behind each requirement, and the success criteria are listed below:

- The C&DH system shall provide 2 GB (unformatted) data storage

Each of the missions requires a substantial amount of on-board storage for the scientific data from the payloads and telemetry data from the other subsystems. In order to accommodate this need, it was deemed necessary to utilize external and non-volatile storage in the form of SD cards. This requirement is considered met if the C&DH system successfully provides 2 GB of storage. This requirement is fulfilled by ensuring that the C&DH computer can detect and mount an appropriately sized SD card during the boot up process and can write to and read from the card during the mission. It was initially decided that there would be two SD cards connected to the C&DH system, one acting as the primary storage, and the second card being used for data redundancy. However, a design change was made to only incorporate one SD card into the C&DH system as it was decided that the redundancy was unnecessary for these CubeSat missions.

- The C&DH system shall receive, process and execute commands within the window of a UT-Austin ground station pass

The missions are considered to be semi-autonomous. In other words, the satellites will be able to execute some actions autonomously such as turning on and off various components based on conditions such as power levels, or automatically downlinking data based on information gathered by an on-board GPS receiver. However, the satellites must also be able to process and execute commands that are uplinked from the ground station. They must be able to provide responses to these commands, if any, without a long time delay so that they are received by the commanding ground station within the same ground pass. This requirement is considered met if the C&DH computer can successfully detect when the satellite is within communication range of the UT-Austin ground station pass, and is able to receive and process commands during the detected pass. The satellite must

be able to receive ground commands, which would trigger the satellite to perform an action on-board specific to the command. A confirmation that the command was processed and satellite actions were taken to execute this command must be recorded, and can be sent down to the Austin ground station for proof of verification. An important factor in meeting this requirement is the defined interface between the C&DH and Communications (COM) subsystems.

- The C&DH system shall activate and begin executing commands upon separation from launch vehicle

It is imperative to overall mission success that the C&DH computer boots upon separation from the launch vehicle. If this does not happen, then none of the other components will receive the commands necessary for satellite operations. This requirement is considered met if the C&DH computer successfully enters the Startup mode (the initial mode) of the FSW after launch vehicle separation. This first involves the computer being able to start the collection of executables that together comprises the software running on the satellite upon bootup. The C&DH computer must then execute Hookem, the main executable of the FSW, and enter its initial operational mode. It is in this mode that the C&DH can begin executing commands. For the ARMADILLO mission, a built-in timeout period of 30 minutes must take place upon launch before deployment of the UHF/VHF antennas and before transmission can occur, allowing for proper separation distance between the satellite and the launch vehicle.

- The C&DH system shall accept and execute a command to reprogram satellite software

This requirement relates to the methods of handling any incorrect and erroneous behavior of the C&DH software. Even with extensive software testing and meticulous procedures and documentation generated for the C&DH software, there will still be bugs and unforeseen runtime errors. Some of these errors may be resolved through a reset of

the C&DH computer. However, other errors might continue to recur even after multiple resets, and may require software modification. Therefore, it is important to have the ability to repair the software after the satellite is in orbit to not jeopardize the mission success. It is also beneficial to have the ability to improve or adapt the software after launch. These capabilities would be useful in the case where unforeseen issues arise and the characteristics of the current software do not allow the successful completion of mission objectives.

This requirement is considered met if the C&DH computer successfully interprets commands to receive a new flight executable via the radio, stores the executable in the proper location, changes the startup script to the new executable, and reboots the computer to execute the new FSW.

- The C&DH system shall manage all commands governing the state and actions of the satellite

The main responsibility of the C&DH system is to execute all of the operations that control the spacecraft. The C&DH is the only subsystem that can change the state (physical and software) of the satellite. All other subsystems are delegated tasks to complete independently but remain under the control of the C&DH system. Therefore, the C&DH system has the responsibility of managing all other subsystems to execute the mission successfully. It must be able to interface with the various hardware components of the satellite by sending commands and receiving back acknowledgement of the requested actions, as well as health and scientific data.

### 3.2 C&DH HARDWARE

Following the Space Mission and Analysis Design (SMAD) approach in sizing the C&DH system, the first step in selecting the hardware is to identify the functions that need to be performed by the system, such as command processing, telemetry gathering and storage, and satellite time-keeping (Smith 2008). The subsystem requirements, which have been presented in the section above, and constraints, need also to be identified. This

aids in determining important characteristics needed from the hardware, such as performance, reliability, and radiation tolerance. The next steps are to determine and understand the level of complexity required by the identified functions so that a C&DH system that can perform these functions will be chosen. The level of complexity is dependent on such characteristics as the satellite's required rate of processing commands, the speed of processing telemetry data, and satellite time management. Finally, attributes such as size, mass and power of the hardware components that are being considered must be taken into account and prioritized based on their level of importance. For example, if designing a C&DH system for a large satellite, the size and mass of the C&DH system may not be as important as the overall power draw. In contrast, a smaller satellite such as a CubeSat will have much larger constraints on satellite mass and size, which would then impose constraints on the C&DH mass and size.

The steps outlined above were followed when deciding on the flight computer. A trade study was performed in order to select the computer used for the current satellite missions. The information gathered from the trade study and the description of the selected flight computer will be presented in the sections below.

The second major C&DH component is the hardware interface board. After consideration of available interface boards, a custom board was designed in-house called Kesler. The board houses the flight computer and connects it to the peripheral devices and other components of the satellite. The Kesler board was based on the needs of not only the C&DH system, but of the other subsystems as well. Kesler connects directly to the EPS, Attitude Determination and Control (ADC), Navigation Visual System (NVS) and Communications (COM) subsystems. The Kesler board also houses the SD card acting as the main on-board storage device. The Kesler board and the SD card will be discussed in more detail in the following sections.

### 3.2.1 Choice of Flight Computer

The choice of flight computer was made based on a trade study performed by Imken in July 2011 (Imken 2011). The results of this trade study are presented here to inform the reader on the reasons behind the selection of the current flight computer for

Bevo-2 and ARMADILLO. It was decided to use this flight computer for RACE as well after conducting the trade study.

#### *3.2.1.1 System-on-Module Chip*

Rather than building a computer from the processor level upwards, system-on-modules (SOM) were considered for the trade study. A SOM, also known as a computer-on-module, is a sub-type of an embedded computer contained on a single circuit board that can be plugged into a carrier board (MEN Mikro Elektronik GmbH 2013). SOMs come in different configurations but generally consist of a processor and standard input/output (I/O) capabilities (Critical Link 2013) which can be configured and broken out to other peripheral devices through a carrier board.

Starting with a SOM as the processor of the C&DH system instead of designing the flight computer in-house has several advantages, particularly for CubeSat missions. One of these advantages is its small size, an ideal attribute for CubeSats where size is a major constraint for all subsystems. Another advantage is that it simplifies the development of the C&DH hardware and allows for more time to be spent on developing well-written and well-tested operational FSW. A student-run lab has to deal with constraints on manpower and time. Therefore, taking the approach of using an off-the-shelf embedded computer system for the C&DH computer saves time and effort that would otherwise be needed for electronic design at the processor level. For example, SOMs include many interfaces which enable easy connection to external peripherals. This attribute saves time in designing the complex circuitry needed for proper computer interfacing (Johl and Imken 2012), and provides a level of flexibility for multiple applications. Being professionally designed, it also improves the reliability of the entire C&DH system. It reduces the risk associated with improper design which can lead to computer malfunctions in orbit and mission failure. As SOMs are mass-produced COTS hardware that is readily available at a low-cost, they are a great option for student-built satellites that have budgetary constraints. Finally, processing and computation power is not compromised, as these SOM computers are powerful enough to control the whole satellite.

#### *3.2.1.2 Trade Study*

In the performed trade study, four computers were considered (Texas Spacecraft Laboratory 2011). The selection criteria for the trade study is as follows:

- Power consumption

It was important to select a computer that had a relatively small power consumption level.

- Ease of software development

As the TSL technical staff is comprised mainly of aerospace engineering majors and no computer science majors, it was important to select a computer whose software interface was easy to comprehend and to use for the developers. Sufficient documentation and software support were also important factors in the decision-making process.

- Performance Capabilities

The chosen SOM must have a processor speed fast enough to handle the planned functionalities of the satellite. For the missions being considered for this class of satellites, the C&DH system is not a hard real-time system and therefore does not require that level of processing performance. The selected computer must provide a sufficient amount of memory to store the program files of the FSW and a partial amount of mission data in case of SD card failure. The flight computer must also have a large variety of peripheral ports for interfacing with the satellite's subsystems.

#### *3.2.1.3 Selection of LPC3250*

Based on the trade study, the selected SOM that best matched the requirements in place for the flight computer is the Phytec's phyCORE LPC3250 (Figure 11).

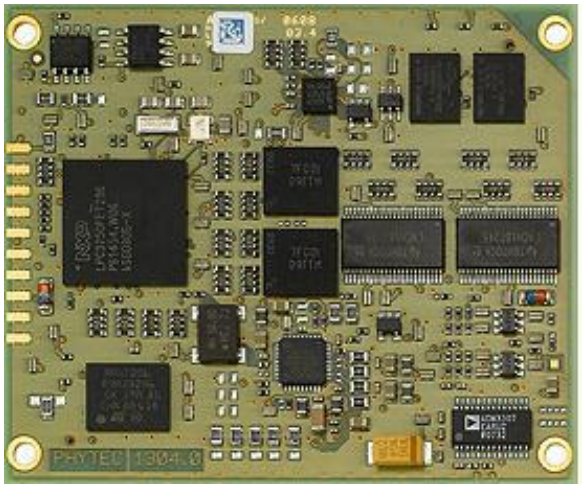

Figure 11. LPC3250 C&DH and ADC Computer

This computer includes NXP's LPC3250 microprocessor consisting of a 266 MHZ ARM926EJ-S CPU core and Vector Floating Point (VFP) coprocessor, and a large set of connections for peripherals (NXP 2011). The microprocessor is designed for low-power, high-performance applications, which is ideal for the TSL's CubeSat flight computers. Important performance characteristics of the LPC3250 SOM are listed in Table 1.

Table 1. LPC3250 SOM Performance Characteristics (Phytec 2013)

| | |
|---|---|
| CPU Frequency (Max) | 208 MHz |
| On-Chip Memory | 32 KB L1, 256 KB SRAM |
| DRAM | 64 MB |
| NAND | 64 MB |
| NOR | 2 MB |
| EEPROM | 32 kB |
| Available SD/SDIO/MMC Expansion | 2 |
| UART | 7 |
| RS-232 | 2 |
| I2C | 2 |
| SPI/SSP | 4 |
| Power Consumption (typical) | 372 mW |
| Power Supply | 3.15 V |

The Phytec LPC3250 allows for easy creation and modification of the Linux kernel through its well-supported Linux development environment, known as Linux Target Image Builder (LTIB). LTIB is a tool for integrating the build and configuration of the software packages for an embedded Linux distribution (Phytec 2011). The LPC3250 allows for the use of Linux as the running operating system on the SOM. This lends itself to a significant amount of customization in terms of the kernel and provides pre-existing software tools and libraries.

### 3.2.2 Kesler Interface Board

The design for the Kesler board is based on the interface board used for the satellite's stand-alone ADC system, developed by QVIS. The Kesler board was designed in-house by the C&DH team. The interface board has currently gone through three revisions. One significant change between v0 and v1 was switching the connection of the

camera to the Kesler from USB to micro USB. This change was made as the old configuration would have required the USB port to be in the middle of the interface board so that the cable head would not hit the inner shell of the satellite structure. Kesler v1 also features a Real Time Clock (RTC) that will be used to keep the time for the satellite and that will be updated regularly from the GPS when possible. The RTC incorporates a temperature compensated crystal oscillator (TCXO) to keep accurate timing when the GPS time is not available. Kesler v1, shown in , will be used as the flight hardware for Bevo-2.

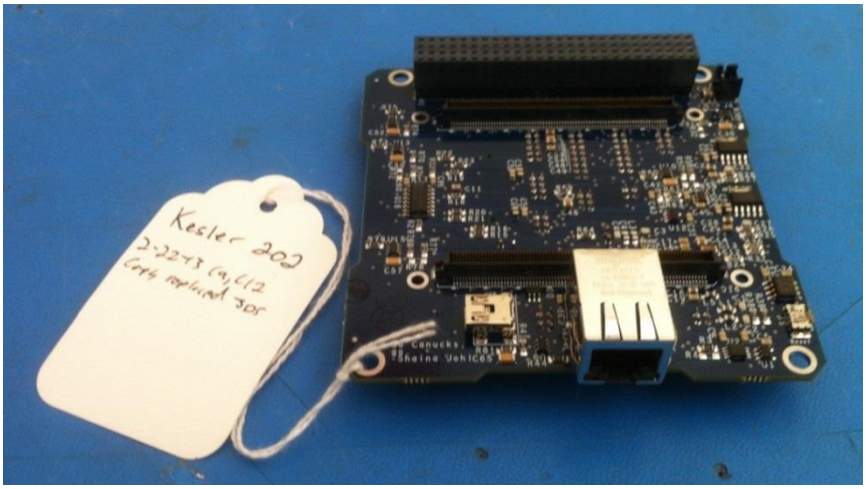


Figure 12. Kesler v1 Interface Board

As shown in the figure, the Kesler board contains a PC104 connector which is used to connect the C&DH system with the EPS and UHF/VHF COM boards in a stack to comprise the bus module. The Ethernet connector is used so that the file system can be kept on a desktop and can be accessed through Network File System (NFS) for testing rather than mounting the flight software onto the NAND flash of the LPC3250 every time recompiling is required. Using the Ethernet port to access the flight software for testing speeds up development time significantly, but it is not included on the flight version of this board.

One major improvement between the Kesler v0 and the Kesler v1 is the addition of power switches to control the power to the subsystem components. These switches are

implemented through an IC that is controlled by GPO pins on the LPC3250. The C&DH system now has the capability to power on and off the other subsystems. This feature proves beneficial when the satellite is transitioning software modes. The flight computer can then turn off any subsystems that are not required for nominal operations; for example, the payload or the camera.

Kesler v2, as shown in Figure 13, the most recent version of the board, will be used for the RACE and ARMADILLO satellites. The main reason behind modifying the design to create a third version of this board was due to the difference in the layout and connection design of the EPS system for RACE and ARMADILLO as compared to the system for Bevo-2. The EPS system used with Kesler v2 is provided by GomSpace, while the EPS system to be flown on Bevo-2 is provided by ClydeSpace. For the GomSpace EPS system, the radio connects to the stack upside down, and the Kesler v2 board's PC104 connector must be a male connector with a reversed pinout. This forces the LPC3250 to connect at an offset from the centre on the board. Other significant modifications to v2 from v1 include a backup battery supply for the RTC so that the time is not lost due to satellite resets, a connector to a separate board that houses the Ethernet port for NFS, and an additional header for power, ground, and data pins for use with mission-specific daughter boards.

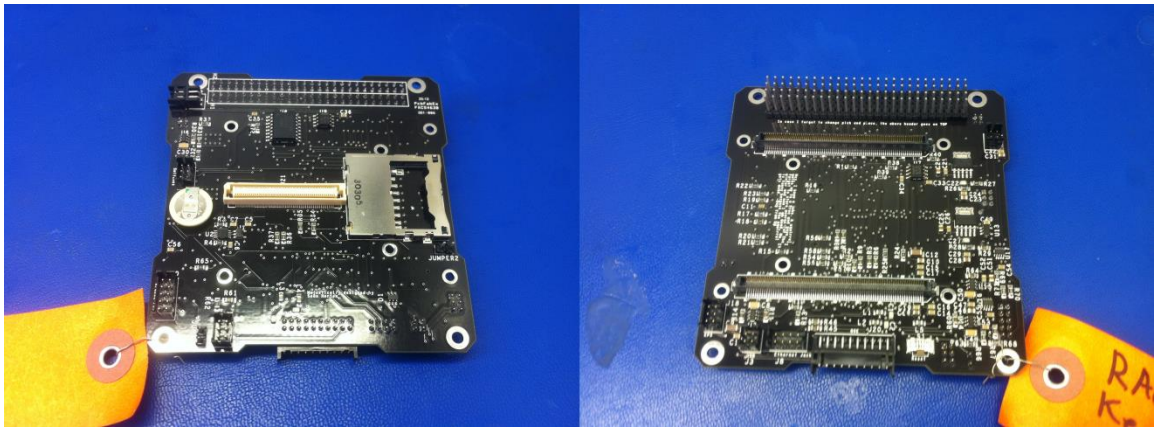

Figure 13. Kesler v2 Interface Board for RACE and ARMADILLO Missions (left: bottom of board, right: top of board)

### 3.2.3 Storage

Mounted onto the Kesler board is an SD card acting as the main storage unit for the satellite. All of the mission and health data will be stored on the 2 GB SD card. A data generation budget was created for each mission to ensure that 2 GB of storage would suffice for the data generated throughout that entire mission (Texas Spacecraft Laboratory 2011). The budget outlines the types of files that are expected to be produced by each subsystem, the rate of generation, the total size of the files for the whole mission, and the allotted storage capacity of the SD card for that type of file.

As the health data is overwritten after a pre-described amount of time and the beacons transmitted periodically to ground (containing a small sample of the spacecraft's health data) are not stored on-board, the main concern in terms of reaching the maximum limit for data storage are the payloads. For ARMADILLO, the main instrument requires 2 kB for one day for a rate of one particle strike on the detector unit per day, totaling 360 kB of data for a complete mission lifetime of 180 days. For the FOTON instrument, with the high-end expected value of 100 occultations per day, the amount of data generated for the entire mission is estimated at just over 322 MB. With this amount of mission data, the health data log files, the pre-loaded mission script files, and the images generated by the camera, the expected maximum data generated for the ARMADILLO mission is 1.31 GB, which is well below the limit of 2 GB for on-board storage. The ground station will also have the capability to remove files from the SD card during operation if it is deemed necessary.

A telemetry budget was created by the Communications team to determine the expected downlink rate. Some of the values and estimates included in the analysis were based on the results from the FASTRAC missions but were slightly improved based on the upgrade of hardware for TSLs' current missions. The expected downlink rate at a baud rate of 9600 bps for the ARMADILLO mission is approximately 234 kB per pass (Texas Spacecraft Laboratory 2011).

Sub-directories will be created on the SD card to organize the different types of data produced. If need be, the ground station will have access to commands that can

modify the state of the SD card on the satellite in-orbit, such as mounting and un-mounting, reformatting, and partitioning.

### 3.3 C&DH Hardware Architecture

The C&DH system used for Bevo-2, RACE, and ARMADILLO is a centralized architecture, with the SOM flight computer acting as the central processor for the entire satellite. A centralized architecture involves all subsystems of the satellite having a point-to-point interface with only the C&DH subsystem. Therefore, all data and commands are sent only between the C&DH system and one other subsystem. This architecture is suitable for satellite systems with a small number of distinct subsystems. Employing this architecture is reliable in the sense that if one system fails during operations, the effect of the failure is minimized as there is no direct interface with the other systems other than C&DH. Therefore, the integrity of the separate interfaces between the C&DH and the other subsystems will remain intact. Figure 14 is a block diagram showing the C&DH hardware architecture and the various interfaces between the C&DH system and the subsystem components.

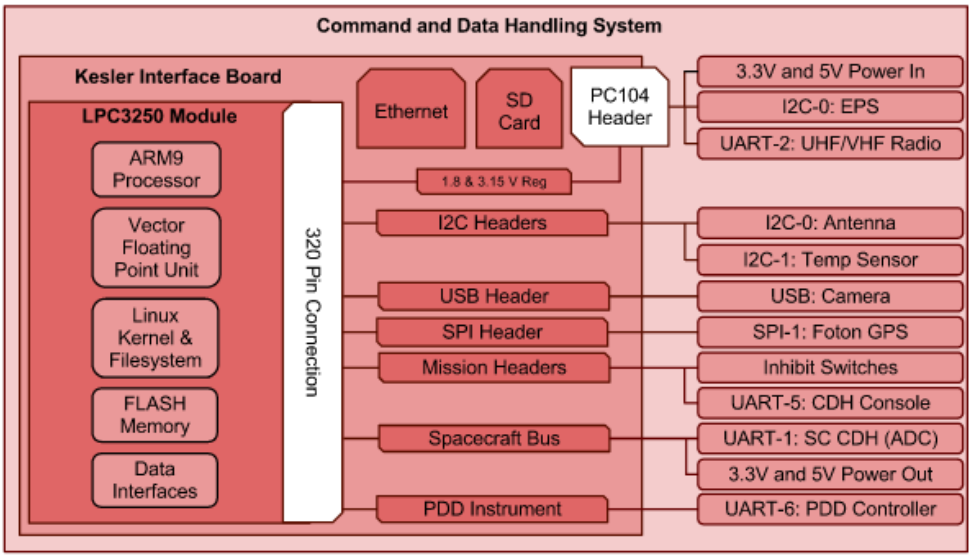


Figure 14. C&DH Main Hardware Components and Interfaces with Spacecraft Subsystems

One significant attribute of the overall satellite hardware architecture is that the ADC subsystem uses a separate computer for its attitude control-related calculations. This design choice was made as the ADC system of the spacecraft is intended to be a bolt-on, autonomous GN&C module that can be used on current and future TSL CubeSat missions, similar to the C&DH system. In addition, the algorithms for the attitude sensing and control are calculation-intensive. Therefore, being able to use a second embedded computer for the GN&C module and still being able to remain within the satellite's allowable power and mass budgets is advantageous. The GN&C embedded computer also uses the LPC3250 based on the results of a similar flight computer trade study for the ADC system. The ADC computer is attached to the Kraken interface board as shown in Figure 15.

Figure 15. ADC Computer System - Kraken Interface Board and LPC3250 Computer

### 3.4 Comparison between Past C&DH systems and Current System

The development process of the C&DH system described in this thesis did not begin from scratch. The TSL had designed, implemented, and validated C&DH systems for several past missions, some of which have been launched. Namely, the TSL has built three satellites that have flown over the years: Sara Lily and Emma as part of the FASTRAC mission, and Bevo-1. The knowledge, design work and lessons learned inherited from the documentation and personnel involved with these past missions were extremely helpful in the development of the C&DH module for the current missions. One

of the initial steps involved in designing the current system was an analysis of the past C&DH hardware architectures and FSW design for these two missions. An overview of the C&DH systems for FASTRAC and Bevo-1 are presented in this section. In addition, a discussion of what design details were reused for Bevo-2, RACE, and ARMADILLO is also outlined.

### 3.4.1 FASTRAC

The FASTRAC C&DH system encompassed a distributed architecture based on an architectural design developed by Santa Clara University (SCU). A distributed architecture involves having several processors that divide the computing responsibilities of the satellite and communicate with each other. Figure 16 is a diagram of the main components of the FASTRAC C&DH architecture.

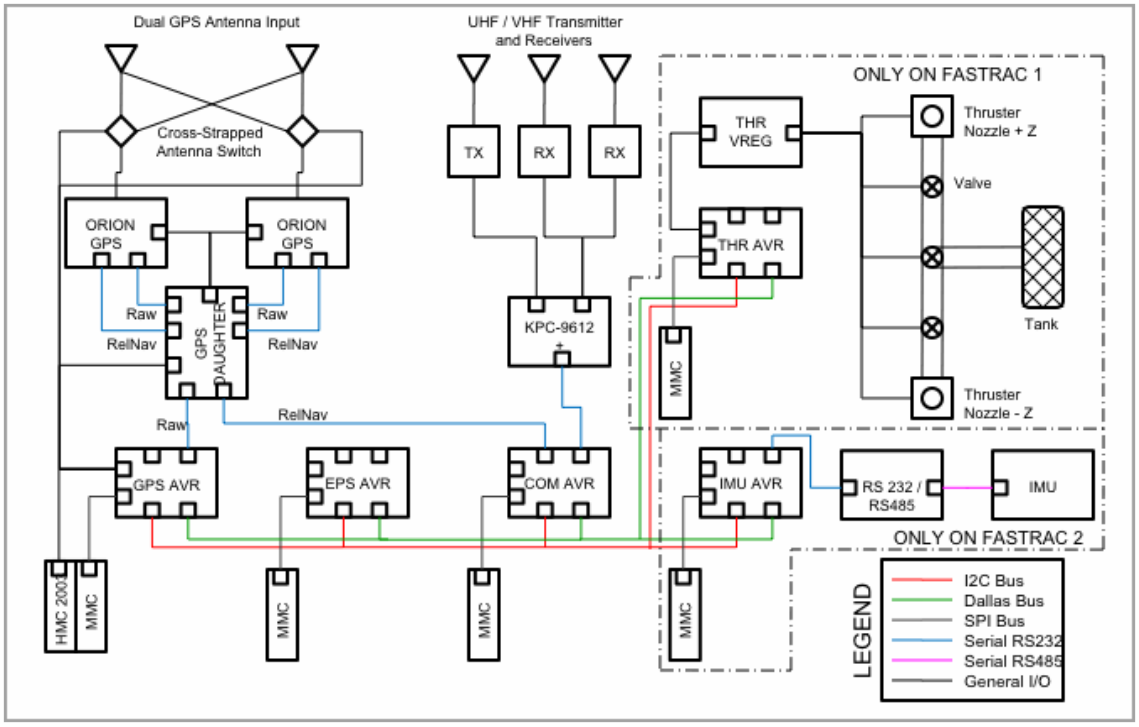


Figure 16. C&DH Architecture of FASTRAC Satellites (Smith 2008)

The FASTRAC C&DH system is comprised of four AVR-SAT microcontroller systems developed by SCU, each of which consists of one Atmel AVR Atmega128 8-bit RISC microcontroller running at 16MHz (Smith 2008). As shown on the diagram, each

AVR microcontroller system manages a separate subsystem of the satellite: communications, power, GPS, and depending on the satellite (Sara Lily or Emma), the thruster or IMU. The AVR-SAT contains 53 general-purpose I/O lines, two Universal Synchronous-Asynchronous Receiver/Transmitters (USARTs), an 8-channel, 10-bit analog-to-digital converter, a Serial Peripheral Interface (SPI), and a 2-wire (I2C) bus (Smith 2008). The four AVRs share data through an I2C bus shown by the green line in Figure 16. An SPI bus is used to connect a 128MB flash memory MultiMediaCard (MMC) to each AVR for data storage. All top-level software for the satellites was written in the C language by the FASTRAC team, and flashed onto a 4KB EEPROM on each AVR.

Based on the success of the FASTRAC mission, this distributed C&DH system architecture by SCU has proven to be effective for the FASTRAC mission. Using a system designed by a third-party and customizing it to the specific mission's needs reduced the mission development costs. This method also saved in component costs as the SCU system is comprised of COTS parts. During the design phase, the distributed architecture in particular seemed advantageous as the communications interfaces were well defined, and this simplified test procedures such as GPS simulations and crosslink tests. As the mission involved crosslinking information between Sara Lily and Emma, the crosslink design and execution was made easier as both satellites had the same distributed architecture. It also was thought that C&DH testing would be simplified as the distributed architecture clearly deconstructed the system into distinct modules whose software could be written and tested individually and in any order.

However, in reality, the FASTRAC team discovered that the distributed architecture did not prove to be the effective choice as previously determined. The main disadvantage of this architecture was in terms of internal data sharing for the satellite. The subsystems required information from each other in order to perform their respective functions. The software design was complicated by the fact that any data flow from subsystem to subsystem for even the simplest functions had to be sent and received over the common I2C bus, where each AVR-SAT was designated a master. This characteristic

introduced timing and sharing complexities, causing non-received data and lockup. Because of this subsystem interdependence, contrary to prior belief, this design did not promote quick, parallel development of the subsystems' software.

The selection of this architecture for the current 3U CubeSat missions in the TSL would have presented significant challenges. The subsystems and mission phases for Bevo-2, RACE, and ARMADILLO are more complex than those of FASTRAC, thus the data sharing between modules would have been even more difficult to manage. In addition, multiple AVRs required for the separate processors would have been difficult to fit within the volume constraints of a 3U CubeSat.

### 3.4.2 Bevo-1

The C&DH system for Bevo-1 encompassed a centralized architecture with a SOM flight computer acting as the central processor for the entire satellite, which is the same architecture as the current TSL missions. The flight computer consisted of a Bluetechnix CM-BF537 core module, shown in Figure 17.


Figure 17. CM-BF537 Core Module (Blue Technix 2012)

Small in size (3.2 cm x 3.6 cm), this 600 MHz processor has 32 MB of SDRAM, 4 MB of FLASH ROM, and has peripheral connection capabilities for SPI, I2C, UART, and SPORT. A detailed block diagram of the module is shown in Figure 18.

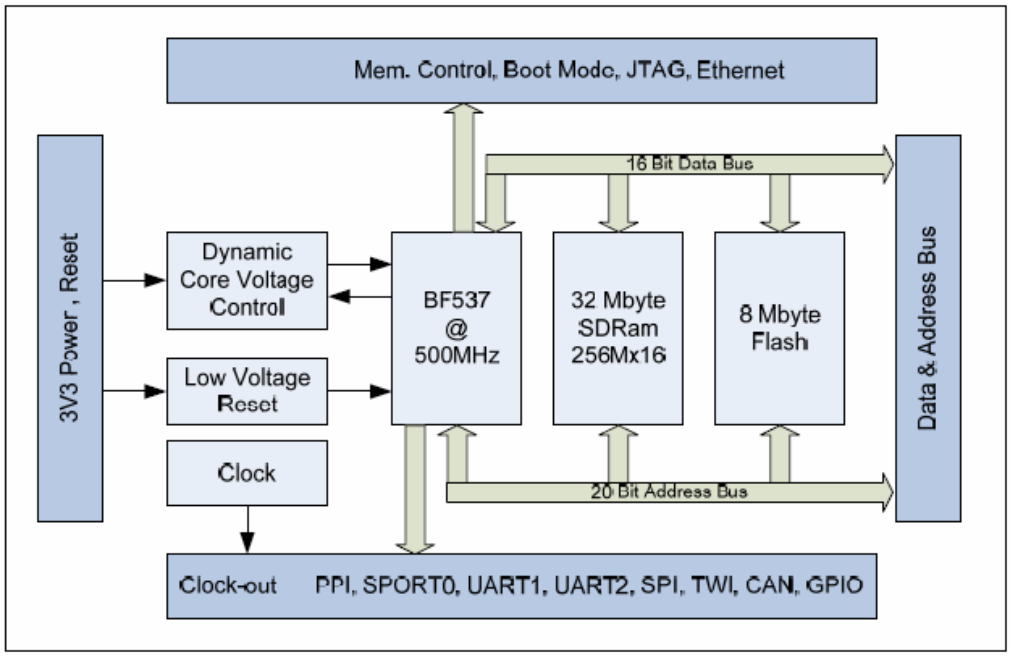


Figure 18. Block Diagram of CM-BF537 (Blue Technix 2012)

This microcontroller supports a Linux distribution called µClinux-dist, which is a software package containing a customized build system for the Linux kernel, several patches, and a large collection of userspace applications and tools (Bhatti 2008). This package is configured and built into a kernel with a root file system. The flight code, named 'Blackbird', is the TSL's first attempt at addressing the need of having reusable satellite software. The high-level software is written in C++ for a Linux runtime environment, which is loaded with µClinux (Bhatti 2008).

The design goal of Blackbird to act as reusable flight software for TSL is a success for the most part. The current C&DH architecture is based off of Bevo-1. The design of having one main flight processor that manages all the operations of the satellite is reused, as well as writing the C&DH software in C++ and running it in a Linux environment. The software architecture consisting of a state machine where the satellite can transition between several operating modes and having separate threads that can be activated or deactivated is also reused for the FSW explained in this thesis.

However, some architectural details are not able to be reused for the current missions, as the Bevo-1 C&DH design has several limitations. First of all, the

BlueTechnix processor was not re-selected as the flight computer as it is found to have a limited number of interfaces that would not suffice for the more complex Bevo-2, RACE, and ARMADILLO missions. Secondly, this microcontroller only supports the μClinux distribution, and not the full Linux kernel. There are only a limited number of differences between μClinux and Linux systems, but the major difference is that the former has no memory management. With this lack of support, no memory protection is offered, thus corruption is more likely and more difficult to diagnose. This difference, along with several other minor differences in the kernel, influenced the selection of a microcontroller using the full Linux distribution. This way, the flight computer for the current C&DH system can be reused on a variety of future missions.

Secondly, the FSW on Bevo-1 was programmed to be inherently tailored to the specific hardware of the satellite. The flight software could not have been easily separated as reusable software modules as many of the subsystem functions were interconnected. As the delivery deadline approached, there was less focus on implementing the flight software in a reusable manner, and it became specifically coded to be functional for the Bevo-1 mission (Imken 2011).

Another change in the development process of the C&DH system from Bevo-1 to the current missions pertains to the FSW implementation philosophy. One pair of team members was responsible for developing all of the software for the mission (Johl and Imken 2012). This was inefficient in terms of development time. In addition, knowledge gained from developing Bevo-1 was lost with the departure of the small software team from the TSL. The members responsible for implementing most of the FSW for Bevo-1 are no longer with the lab. Thus, the TSL had no prior experience with FSW to use for the current missions. Currently, the coding responsibilities are purposely divided between the subsystems. Furthermore, there is more of an emphasis on documenting and logging design decisions, troubleshooting methods, and software bugs. The availability of the knowledge from these past satellite development experiences provided a solid foundation for the system design of the current missions.

# Chapter 4: Software Architecture and Development Infrastructure

Developing the software for a CubeSat mission is a large and complex project. In order to complete the development in a timely manner and to minimize software failure due to rushed code development, the task of writing software is divided among each group of students working on a particular subsystem. More information on this division of software duties will be provided in this chapter. However, as the C&DH subsystem is responsible for all commands and telemetry to and from the spacecraft and the ground station, the C&DH team spends a great deal of time developing software. Hookem excludes the ADC software that will be loaded onto the ADC computer connected to the Kraken interface board. The party responsible for the compilation of Hookem is the C&DH subsystem team. Writing the code for the C&DH subsystem is a large task in itself, but the C&DH team is also responsible for maintaining and compiling all flight software to be run on the satellite's main computer. As with all large software projects, there is much planning and designing that must occur before beginning to write the code. This not only includes creating the software architecture while taking into consideration the lessons from past TSL missions, but also setting up an appropriate development infrastructure.

In this thesis, a clear distinction is made between the software architecture and the software development infrastructure. Software architecture is a common term used in software engineering. A general definition of software architecture is the structure of structures of a software system that consists of entities or components, and the relationships between them (Franchitti 2011). In this definition, a software component is an encapsulated part of the software system, acting as one of many building blocks for the structure of the system (Franchitti 2011) such as classes, objects, or modules.

In contrast, the software development infrastructure as described in this thesis includes all practices that are put in place for the C&DH team that facilitates the development of the FSW. Examples of strategies and techniques incorporated into the software development infrastructure include coding guidelines, flight software releases, and subversion control. Therefore, the software infrastructure includes practices that aid

in developing software efficiently and correctly, whereas the software architecture is the organization of the software system itself, along with its components, their relationships to themselves and to the environment (Eeles 2006).

Software infrastructure and software architecture are important definitions in the field of software engineering. The C&DH subsystem has put an emphasis on using practices and techniques learned from research and courses in software engineering when developing the software for the current missions. This is a change from the approach taken by the C&DH team from past missions in the TSL. Since a major design goal was to design the subsystem in a manner that allowed the architecture to be re-used for future missions, it was important to put into practice correct software engineering principles and processes. This would ensure the production of quality software that satisfies the requirements of the subsystem, and that can be used as a solid base from which future members of the TSL can design the C&DH software for the next generation of missions.

The flight software architecture and the flight software development infrastructure are discussed in this chapter.

## 4.1 SOFTWARE ARCHITECTURE

Well-defined and planned software architecture is an important attribute of the development for the FSW. Before discussing the details of the software architecture used for the CubeSat missions in the TSL, it is important to understand the difference between the software architecture and the software design. Software architecture, as presented earlier, is the process of structuring a software system that consists of entities or components, and the relationships between them. According to Perry and Wolf (Perry and Wolf 1992), software design involves the modularization and interfaces of the design elements, and the algorithms, procedures, and data types needed to satisfy the requirements (Eden and Kazman 2003). Therefore, software design is more concerned with the lower level detailed-design issues rather than architectural-design issues. The implementation of the software program then follows, which is writing the code based off the software architecture and design. The boundaries between the definitions of these

three concepts are often blurred, yet present. A diagram that illustrates the gradual distinction between architecture, design and implementation is shown in Figure 19.

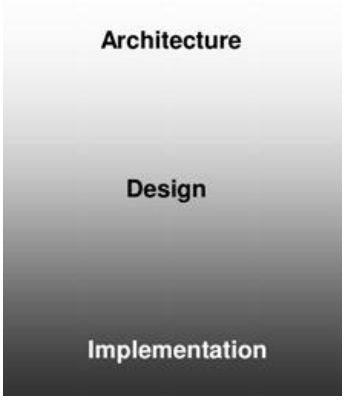


Figure 19. Common View Distinction between Software Architecture, Design and Implementation (Eden and Kazman 2003)

The remainder of this chapter presents the TSL's chosen FSW architecture while the subsequent chapter will describe the detailed design and implementation of the FSW.

### 4.1.1 Architecture Goals

The development of the software architecture begins with the problem definition that the software must address. For this project, the flight software must satisfy mission-specific requirements and the common C&DH requirements described in Chapter 2.

The software architecture must also address the requirements of the various stakeholders while handling the functional and quality requirements (Brumbaugh 2012). A stakeholder is defined as an individual, team or organization with interests in the system (Eeles 2006). The stakeholders in this case are the other members of the TSL responsible for the remaining subsystems of the satellite, the TSL as a whole, and the organization responsible for the payload. The needs of these stakeholders drive the C&DH system to have two main non-functional requirements/goals in hand: modularity and reusability. Achieving modularity in the C&DH system is important to the members working on the other CubeSat subsystems as they do not want to be concerned with the implementation of the C&DH software. Similarly, the C&DH team members do not want to be concerned with the implementation of the other subsystem's software. Reusability

is an important characteristic of the architecture to the TSL as a whole for reasons discussed previously; the more reusable the code is, the more time will be saved for software development for future missions. These non-functional requirements are significant in the software architecture development.

#### *4.1.1.1 Modularity*

The term modularity is used loosely, and has several definitions in the field of engineering. For a general definition in the software engineering sense, a software system is modular if components, or parts of the software, can easily be identified and replaced (Parallab, Bergen Center for Computational Science 2004). The modules can then interact through a defined interface, outlining the data required by and provided by each module. Focusing on modularity in the software architecture promotes software reusability, which will be discussed in the next section.

The TSL has implemented the concept of modularity in several layers of its hardware design. The hardware architecture of Bevo-2, RACE, and ARMADILLO structures are very similar. The structural layouts of Bevo-2 and ARMADILLO consist of three modules, known as the Service module, the GNC module, and the Payload module.

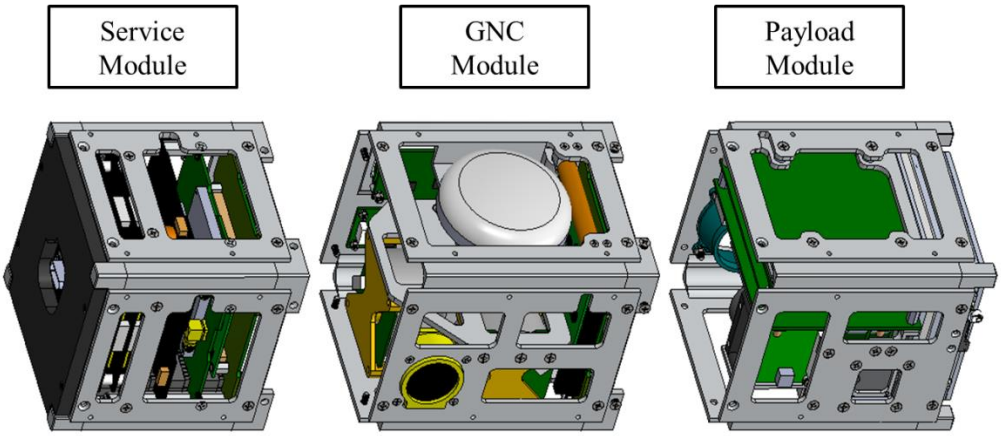


Figure 20. Modular View of ARMADILLO Satellite

Each module of the satellite serves a specific purpose. With this design, each module can be developed simultaneously with minimal dependence on the other modules. Two of these modules, the Service and the GNC, are used for each satellite with only minimal necessary changes. It was decided to mimic this concept of modularity in the software architecture as well so that subsystems as a whole, both hardware and software, could be outright replaced by an entirely different version of the subsystem. Since all control and data transfer is routed through the C&DH system, a functioning subsystem comprising of the hardware components and the software necessary to operate it accordingly can then be completely replaced without affecting the rest of the satellite.

For the flight software, the term modularity is used to describe how the software for the satellites is separated into black boxes, or modules, the contents of which are only added to or modified by the subsystem in ownership of that part of code. There is one module per subsystem, and Hookem is then composed of all the modules compiled together into one executable. Figure 21 illustrates the different modules that comprise Hookem for the current missions.

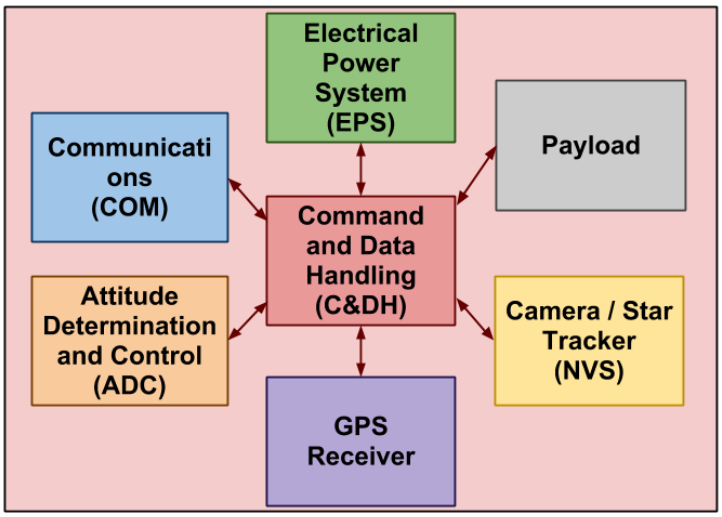


Figure 21. Software Modules within Hookem

The diagram above shows the C&DH subsystem in the center, interacting with all other subsystems of the satellite, indicated by the bidirectional arrows between modules. However, all subsystems, excluding the C&DH, do not interact with each other. For example, the PDD requires the time in the form of a timestamp which will be recorded along with the data when an impact is detected. In this event, the PDD will not request the time directly from the GPS. Rather, it will query the C&DH system for the time, which will in turn ping the GPS, and return the timestamp to the PDD instrument. This structure allows for only one software interface between each module and the C&DH.

An example of subsystem modularity can be found by observing the GPS receiver subsystem for the current missions. Bevo-2 uses NASA's single frequency DRAGON GPS receiver for the mission. The ARMADILLO mission, however, will fly the dual frequency FOTON GPS receiver which was developed by the Radio Navigation Lab at UT-Austin. Even though the FOTON will fulfill the mission's secondary objective of collecting GPS radio occultation measurements, it will also provide the same GPS receiver functionalities the DRAGON will provide for the Bevo-2 mission, such as GPS time, position and velocity measurements. From the perspective of the C&DH team, the commands sent to the GPS receiver subsystem will be functionally similar for both the DRAGON and the FOTON. The C&DH will also be able to acquire the data from the GPS and downlink it in the same fashion for data post-processing. There may be added capabilities for either receiver that the other does not exhibit, such as the FOTON being able to acquire GPS L2 frequency data in addition to L1 data. Therefore, the C&DH team may decide to modify the commands that can be sent to the FOTON, but they both would maintain the same functional software interface with the C&DH. This type of subsystem modularity is also valuable in the situation where the hardware requires an upgrade to a newer version. Inevitably, the software for that specific subsystem will need to be modified. However, no other subsystem software, including the C&DH software that performs the function call, will require any changes. An example of this has already occurred in the lab, as the UHF/VHF radio from AstroDev has been upgraded from one version (the Lithium) to a more recent version (the Helium) in the current development

cycle of the satellites. Another example is the swapping of the ClydeSpace 3U Electrical Power Supply used for Bevo-2 with the GomSpace NanoPower P-Series power supply for ARMADILLO and RACE. For this change in hardware, only the EPS low-level code needs to be modified, and the FSW is able to call the high-level EPS functions in the same manner for all satellites.

There are several more benefits associated with software modularity in addition to subsystem interchangeability. One of these benefits is the parallel development of the FSW. Complete subsystems can be developed in parallel without interdependencies (Johl and Imken 2012). This approach aids in dealing with constraints associated with a student-run lab such as lack of manpower. Subsystems can be developed at different rates, allowing re-allocation of resources such as assigning people to a different subsystem that requires more manpower for completion. Another benefit is that more students gain experience in developing skills in software engineering and software implementation. Each subsystem is responsible for the software required to meet its respective requirements.

#### *4.1.1.2 Reusability*

Software architecture helps set the foundation for the system to obtain its non-functional requirements. As the TSL is working on two missions concurrently, a non-functional requirement to consider in developing the flight software architecture is reusability. Reusability is a term that is commonly used not only in software engineering, but also systems engineering. Bevo-2, RACE, and ARMADILLO, like other aerospace projects, are complex systems that require a group of engineers to invest their time and effort into their development. However, resource constraints are always an obstacle in the development of CubeSat missions in the TSL. There are several examples of reusability implemented in different stages of the TSL CubeSat development process. The most prevalent example of reusability is the selection of hardware between the three missions. Even though the current missions have different mission objectives, all the missions have the same requirements in terms of computing needs and ground communication capabilities. Thus, it was decided in the design process and component trade studies that

the same Phytec LPC3250 flight computer and the same Helium 100 UHF/VHF radio could be used for all satellites. Therefore, two thirds of the bus is nearly identical and interchangeable between the missions, saving significant development time and costs. Another example is the application of reusability in the production of mission documents (Brumbaugh 2012). Due to the complex nature of satellite systems, it is important to keep thorough and well-organized documentation during all phases of the development cycle. Pre-existing documents, such as requirements, test plans, and interface control documents, can be used by future members of the TSL as templates. By applying reusability techniques when constructing CubeSats, the TSL is able to reduce the engineering effort required to develop a new system.

Reusability techniques play a large role in the software development of the satellites as well, and software engineers are very familiar with this term. Software reusability, as per the definition by the Institute of Electrical and Electronics Engineers (IEEE), is the degree to which an asset can be used in more than one software system, or in building other assets (IEEE Computer Society 2004). An asset can either be reusable software or software knowledge. Reusability is the quality of a piece of software that allows it to be used again by another application in the full, partial or modified version of itself (Parallab, Bergen Center for Computational Science 2004). Writing code with a design goal of reusability is generally good practice, as this practice will diminish the time and effort required to produce future code (Oualline 2003). Reusable software, including the software design, functional specifications, and code, is created with this design goal in mind. Attention is required to the choice of software architecture to allow for successful software reusability.

It was a very easy decision to develop the flight software for the current missions with software reusability in mind, as there are several advantages from this:

- Increased productivity

Reusing software allows for a quicker rate for software development. Pre-existing software components that have been fully implemented, tested, and documented can

essentially be “plugged-in” to the system, saving the developer time that would have been required to create new software.

- Shorter Development Time

The development of software for CubeSat systems is a lengthy process that takes up a significant portion of the overall development schedule. Any technique or strategy that can save time for software development while producing the same desired results is considered an asset and should be further investigated.

- Increased software quality

Any FSW that is being reused in the TSL would have had flight heritage from previous missions. This means that this piece of software has been proven to properly function. The lab puts an emphasis on documentation, which would result in the implementation, testing, and specification being easily accessible and comprehensible by members of the TSL other than the creator.

- Better leverage of engineering knowledge and skills

During development of future missions in the lab, more engineering resources can be applied to other aspects of the mission rather than to re-writing existing software. There would be no need to re-invent existing, functional code, which would result in savings on overall development time and effort.

Reusability can be further decomposed into two types: planned reuse and unplanned reuse. In unplanned reuse, also known as the opportunistic approach to software reuse, some or all components of software is reused from a software system that was not originally intended to be re-used (Jansen, et al. 2008). However, the software developer has knowledge of previously existing software and identifies it as being applicable to the current software system. In contrast, planned reuse, as stated in the term itself, involves planning for reusability from the beginning of the development phase. This latter type of reusability is favored over the former since a more planned approach

helps in identifying the impact of reusing software on the system beforehand (Fortune 2009). For example, with planned reuse, an assessment can be made on the effects of reusability on testing strategies, design standards, and on the quality and integrity of the software (Lam 1997).

Hookem is based on planned reusable techniques so that future missions in the TSL will benefit from the engineering effort and knowledge gained from these projects. With several future missions already in the conceptual phase, designing reusable flight software is vital to the continued success of the lab in producing functional CubeSats.

#### *4.1.1.3 Promotion of Reliability through Modularity*

Selecting a software architecture that is modular in nature is an effective way of building software that has reusable parts. The software is divided into well-defined modules in such a way that some of them can be reused in future applications with entirely different mission objectives. The black-box, modular approach taken in Hookem promotes the reusability and replacement of subsystems of the satellite as a whole (both in hardware and software). The reuse of these software modules in particular is desirable as software costs are usually based on the number of lines of code that must be written. Therefore, if software modules can be reused, this minimizes new code development costs and leaves only the integration and system level costs as major contributors (Larson and Wertz 2006). In conclusion, modularity and reusability are valuable software architectural goals to strive for that go hand in hand.

### 4.1.2 Architectural Patterns

A critical step in developing any large software system is defining its software architecture. It serves as the blueprint that helps the system meet its functional and non-functional requirements. Most system architectures are formed from high-level principles and patterns that are commonly used in many software systems. These principles and patterns are known as architectural patterns, or architectural styles. Architectural styles are groups of design decisions and constraints which can be applied to a system to induce chosen qualities (Fielding 2000). Each style has its own key principles, benefits, and

design rules that distinguish it from other styles. In addition, architectural styles may have qualities that are used in various aspects of applications (Microsoft 2009). For example, there are some architectural styles, such as service-oriented architecture, that describe a software communication design, whereas there are other styles, such as object-oriented or layered architecture, that describe structure designs. Therefore, software architecture usually combines several architectural styles in its design to meet its requirements.

Using architectural styles is good practice in software engineering as the user benefits in several ways. First, using architectural styles is a form of software reusability and therefore provides the same advantages as the latter concept. These architectural patterns are recurring application-independent rules and decisions. Thus, it is well known engineering knowledge that can be exploited by the software developer to facilitate the software development phase (Garlan, Allen and Ockerbloom 2009). An architectural pattern provides the user with a routine solution to certain types of common software problems. Another advantage is the high level of abstraction architectural styles provide to large, complex software systems. The software developer can look at the software problem in hand and its requirements at a high level, and then make top-level software design decisions based on selecting styles that fit the system. In other words, architectural styles provide guidance on how to design a system based on the requirements, rather than having the difficult engineering task of starting from scratch (Fielding 2000).

The architectural styles used for Hookem were selected based on the opportunity to inherit the code from Blackbird. As the C&DH hardware architecture was to remain the same for the current missions as it was for Bevo-1, it made the most sense to keep the same software architecture as well. The Blackbird FSW is fully functional and tested, and therefore provided a robust starting point for the development of Hookem. Blackbird was designed with the same architectural requirements as the current missions, including modularity and reusability. Therefore, even though Bevo-1 was relatively simpler in its mission objectives than the current missions, the C&DH team decided to follow the same architectural style for Hookem as was used for Blackbird. One minor shortcoming in the

Blackbird system is that the documentation is lacking. This may have been due to the lack of manpower for the C&DH system or a rushed delivery schedule. This is one key attribute that the current C&DH team is trying to improve, and is a motivation behind this thesis.

The software architecture for Hookem uses two architectural styles in its design, known as component-based and object-oriented. A description of these two styles and examples of how they were used in Hookem are included in the following sections.

#### *4.1.2.1 Component-Based Architectural Style*

One style that was used in the architecture for Hookem is called Component-Based Software engineering (CBSE). CBSE is a software engineering approach that concentrates on decomposing the software system into individual functional components, with well-defined communication interfaces between them. This style type may have similarities to the object-oriented architectural style, which is described in the next section, but the two styles differ in several ways. CBSE is practiced at a higher level of abstraction than the object-oriented style, and it does not contain principles regarding communication protocols or shared states (Microsoft 2009). Another difference is that in the object-oriented style, the software objects and their interactions model the real world, and can be thought of as nouns and verbs respectively (Phytec 2011). In contrast, for CBSE, this need not be the case; components do not need to follow this restriction.

The main principles for CBSE involve:

- Reusability

Most components are designed to be reused in different applications, but some components may be designed for one specific application and purpose only.

- Replaceability

Components can be substituted by other components, and are non-context specific.

- Encapsulation

Components do not reveal details of its internal functioning, variables or states. They only allow the user access to their interfaces.

- Independence

Components have minimal dependence on other components of the system. Newer versions and deployments of the components can be released without affecting the other components of the system.

The CBSE architectural style was implemented in Hookem in the form of the composition and interaction of the various subsystems of the satellite. As shown in Figure 21, the subsystems represent the components of the software system. The interfaces between components are represented by the bidirectional arrows. As mentioned earlier, all subsystems interface with only the C&DH subsystem. Each software interface between the subsystem and the C&DH consists of a set of functions that the C&DH is able to call in order to command that subsystem.

Using this architectural style helps in meeting the two non-functional requirements for the flight software. It provides modularity in the form of independent software components for each subsystem. Reusability is achieved by being able to interchange and replace the individual software modules in the current system as needed, and by having the capability of building a new system with a combination of existing modules and newly developed modules.

#### *4.1.2.2 Object-Oriented Architectural Style*

Another style that was used in the architecture for Hookem is called Object-Oriented (O-O). This is a well-known and commonly used style among software developers. The O-O architectural style involves dividing the system into object instances (Microsoft 2009). Each object contains its own relevant data and behavioral properties. Objects are independent, discrete, and loosely coupled, and communication between objects is performed through accessing properties of other objects, and by sending and receiving data (Microsoft 2009).

There are four main principles that describe the O-O architectural style:

- Abstraction

The complex software system is divided into components that are easier to comprehend. It involves reducing a part of the system to its essential characteristics as a component, and to produce general operations that support it.

- Composition

Objects can be composed of several other objects, which can be hidden or exposed to the other classes of objects.

- Encapsulation

Similar to the principle for CBSE, objects do not reveal details of their internal functioning, variables or states. They only allow other objects access to their interfaces.

- Polymorphism

The behavior of an object can be overridden by implementing new operations that are interchangeable with the pre-existing operations for that object.

The object-oriented (O-O) architectural programming style has been used extensively in the architecture of the software. The C&DH software has been written in C++, which supports object-oriented programming. Each subsystem team has written their low-level code in either C or C++, but they are required to provide a high-level C++ wrapper that includes all the functions that the C&DH is able to call. This wrapper acts as the interface between the two subsystems. Each C++ wrapper includes the commanding functions. Therefore each subsystem is represented as an object that interacts with the C&DH software. Figure 22 illustrates the interface between the C&DH software and the subsystem software.

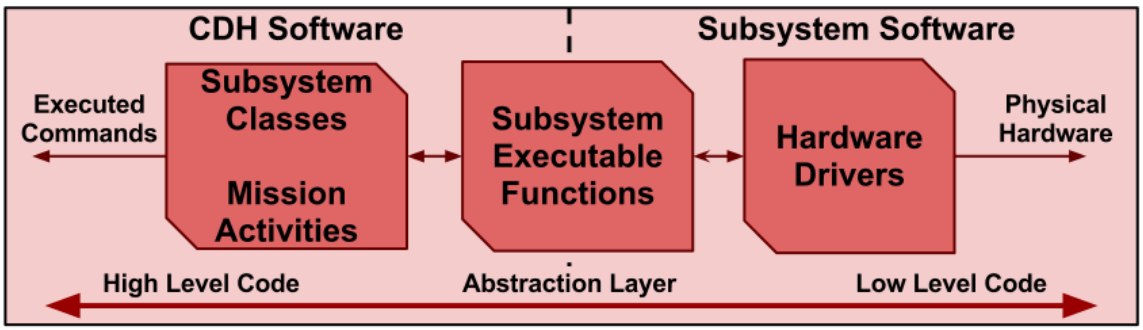


Figure 22. Interface Object Software Interaction

The block on the right side of the figure represents the low-level code for a subsystem. This is the code that directly interfaces with the physical hardware of that subsystem. The block on the left side of the figure represents the C&DH software entity of the FSW. The center block of the diagram is the subsystem component that acts as the interface between the C&DH software and the subsystem software. Complete functionality is provided through this interface, which is implemented by the respective subsystem in collaboration with the C&DH software. Not all of the functions included in this software layer may be used for a specific mission. However, it is still important that they exist, as these objects are to be used for other current missions and future missions.

An example of an application of the O-O architectural pattern used in Hookem is given as follows. The C&DH system is responsible for monitoring the EPS battery voltage and using this information to control the state of the satellite. This is done by the C&DH software querying the EPS object for the voltage by calling the high-level function that does so in the C++ interface. Encapsulated in this function, the EPS calls a C function that communicates with the EPS hardware by reading in and parsing the data from the I2C line. The high-level function then converts the binary data to a float value and returns it to the C&DH system, which then uses this information to perform an activity such as switching the satellite into Low Power mode if the voltage is below a limit, or writing the value into a beacon which will be transmitted later to the ground station.

## 4.2 SOFTWARE DEVELOPMENT INFRASTRUCTURE

The following sections describe techniques and strategies as part of the software development infrastructure for the TSL. These practices and tools were put in place to facilitate and shorten the time for the development of the flight software.

### 4.2.1 Interface Control Documents

In order to govern the software interfaces between the C&DH system and the other subsystems, Interface Control Documents (ICD) were created. The template of these documents was created by a member of the C&DH team. One ICD is created for each subsystem, and the same ICD is used for all three missions. This is made possible by having a high-level source file for each subsystem that acts as a wrapper to the subsystem's lower level functions. The sections of the ICD are completed by the lead member of that subsystem, and continually modified as further progress is made on the software. However, it is the shared responsibility of the C&DH team and that particular subsystem that the ICD is accurate, and correctly portrays the current version of that part of the FSW.

Information included in the software ICDs first involves a high-level description of the electrical interface between the C&DH and the respective subsystem. For example, for the NVS subsystem, the mvBlueFOX camera interfaces with the C&DH through the USB port on the Kesler interface board, giving the camera the capability to take pictures and to run the star tracker algorithm. A list of all the individual source and header files that comprise the device driver software (the low-level subsystem code), and the subsystem interface software (high-level C++ wrapper) is provided, along with any test files that can be run to demonstrate the functionality of the subsystem but that are not included in the final FSW. For example, the NVS subsystem has the high-level C++ wrapper named camera.cpp with camera.h as the header file, but its low-level device software includes several header files: mvDeviceManager.h, mvDriverBaseEnum.h, mvIMPACT_acquire.h, and mvPropHandlingDataTypes.h. The test file that demonstrates

how to use the high-level functions of the camera, such as the ‘power on’ and the ‘take image’ functions is named camTest.cpp.

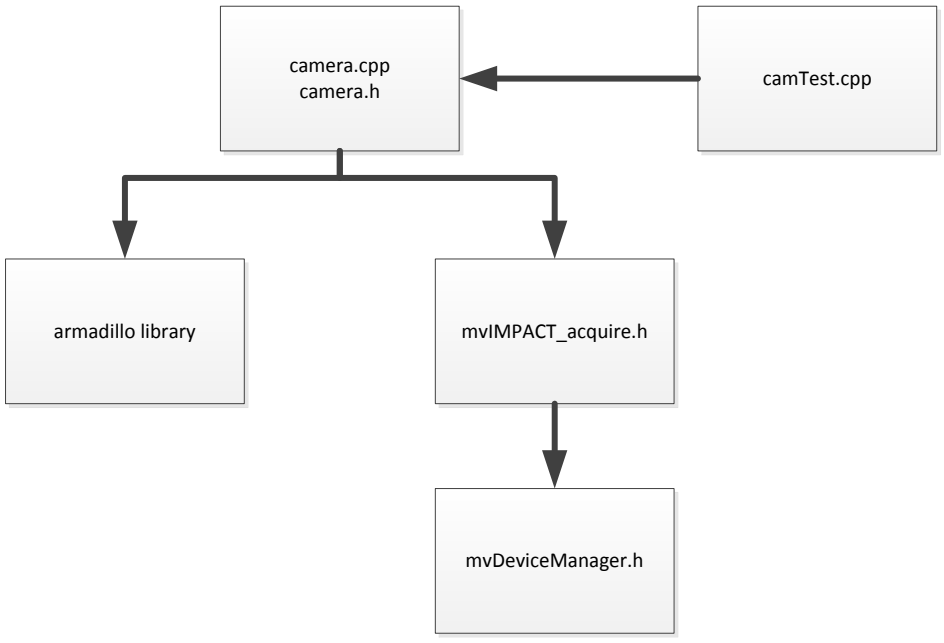


Figure 23. Software Organization Structure for the NVS Subsystem

This test file is not a part of Hookem, but it is critical to the functional testing of the subsystem. Additionally, the specifications of every high-level function that comprises the subsystem interface software, with details on the input and output variables, are given in the ICD. Finally, the document also includes a section for describing the current progress of the software, and any current issues with the high-level interface software that the team should be aware of. The ICD is updated to reflect any revisions that are made to the subsystem software that affect the interface. The ICD then can be used as a valuable reference by the project team for the summary of the subsystem software interface details.

### 4.2.2 Software Releases and Software Directories

Hookem has gone through several versions, and has had multiple software releases over the development process for the current missions. Updates to the

functionality of the FSW continuously occur throughout this process. Even though the software is not yet in a state of full functionality, releases of the software have been made for several satellite test opportunities such as the high altitude balloon launch described in section 6.8.1, and demonstrations for design reviews. The TSL uses the open-source version control systems Subversion (SVN) and Git for documentation and software control. Git acts as the file sharing repository for the entire lab's software, while SVN is used for all documentation, engineering drawings, tables, analyses and all other types of files.

The satellite software created for the current mission exists in subdirectories of one main folder named Git. Within this root folder, there are three main subdirectories, picosat_cdh, picosat_adc, and picosat_cmn (Figure 24).

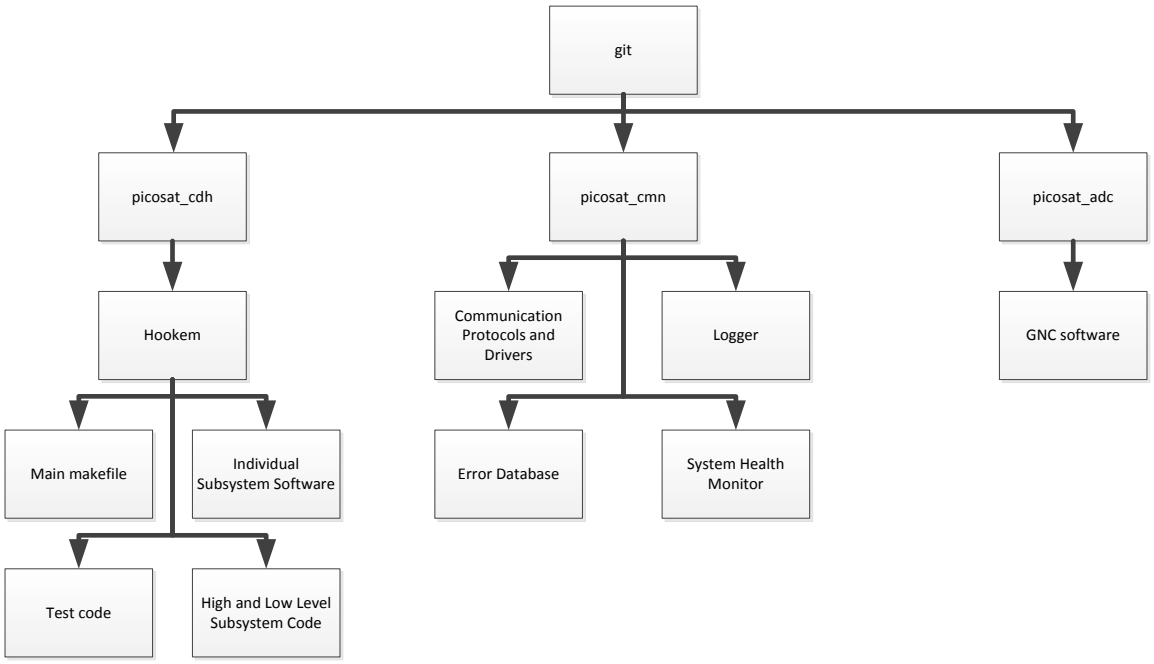


Figure 24. Diagram of TSL's Software Source Code Organization

The software written specifically for the ADC subsystem is kept in the picosat_adc folder, and executes on the LPC3250 connected to the Kraken board. The

software for all other subsystems is kept in the picosat_cdh folder, as it is compiled together and ran on the main flight C&DH computer on the Kesler board. The picosat_cmn folder is used to store all source files that are common to both the main FSW and ADC software. This includes implementations of communication protocols or standards, including NanoSatellite Protocol (NSP), UART, and I2C, as well as source code for creating and managing on-board databases used for logging received commands or satellite software errors.

Inside the picosat_cdh folder exists subdirectories for all subsystems. Within their subdirectory, there are sub-folders for each subsystem that contain their low-level subsystem software, as well as the high-level C++ interface. Each subsystem subsequently has a testing folder which contains any test source files. The picosat_cdh folder also contains a sub-directory which contains the C&DH software, as well as the file responsible for compiling all of the Hookem software into one executable. A recent change in the compilation process is that each subdirectory in picosat_cdh corresponding to a subsystem has their own make file which get called by the governing make file in order to compile the software. The software file structure through Git is important in keeping the software well organized and helps minimize file location problems arising between team members. Git also allows the software developers to revert back to previous versions of the software in case the current version no longer compiles and the issue is difficult to fix.

### 4.2.3 Development Board

Before the creation of the Kesler interface board, the flight computer was used in conjunction with a development board, the phyCORE-LPC3250 carrier board. This board, depicted in Figure 25, was provided by Phytec, along with the flight computer, as a component of a kit.

Figure 25. phyCORE-LPC3250 Carrier Board (Phytec 2013)

The development board provides all the identical electrical interfaces, and input/output connections needed for the flight computer. The Kesler board was then custom-designed to mimic the electrical interfaces of the development board. It is important when working with embedded systems to first work with a development board. Any bug in the software, or any components improperly wired to the board, could potentially damage the flight computer or other expensive connected subsystem hardware. Development boards, as opposed to the Kesler interface board, have some built-in circuit protection to limit the damage caused by these common mistakes. Additionally, bugs or failures experienced while using the development board simplifies the debugging process as all possible hardware bugs associated with the Kesler board are ruled out.

### 4.2.4 Coding Standard

Common coding style is a useful practice to apply to large software systems involving many code developers, such as the FSW for CubeSat missions. Proper coding techniques include writing functional specifications for each method or function in the source file, commenting lines of code, and proper indentation, making the structure of the program easier to read. Maintaining a uniform and well-structured naming convention for

variables, methods, and classes included in the C&DH components of the software is also important. Ideally, all the code that comprises the FSW should cohere as if one person had written all the code. Well-styled code will also reduce the time and effort needed from other TSL members to understand the code, or for future software developers trying to understand and reuse the code for their needs. The C&DH team implemented these coding style practices while developing their parts of the code. A guide was created in order to document these practices, and is followed by new TSL members to maintain the correct code structure even with TSL member turnover. Future missions in the TSL will implement these coding styles throughout the FSW development and test activities.

# Chapter 5: Software Architecture and Development Infrastructure

The previous chapter discussed the tools and strategies put in place in the TSL to simplify the process of writing the FSW. It also described several software architectural styles that were used to guide the FSW development. However, the software architecture is only a high-level abstraction of the software. There is still much work that must be done to transition from first having the system architecture selected, to generating a design for the FSW on a lower level, and then to finally commencing implementation. The C&DH team spent several months generating diagrams, including flow, class, and sequence diagrams before writing any code. These diagrams help to create a model of the software system that is an abstract representation of the system (Gomaa 2011).

Modeling the FSW is good software engineering practice as it helps the developer better understand the system before delving into the details. Taking the time up front in creating diagrams to model the system saves time in the long run, as major software flaws or gaps are less likely to occur during implementation. Nonetheless, it is inevitable that modifications to the software design must occur while writing the code. However, being able to go back through the diagrams, identify the errors, and find ways to rectify them on paper saves time in comparison to changing the code directly and discovering that the changes do not work. Diagrams aids in the visualization of breaking down the larger software problem into implementable modules and determining their functional characteristics and relationships.

In the early design stage, the C&DH team members originally created simple flow charts on Google Docs. It was decided that the next versions of these figures should be more detailed. Therefore, they were re-made in a program dedicated to generating graphical software models. There are currently several graphical modeling languages available for software-intensive systems. However, Unified Modeling Language (UML) is the industry standard graphical language and notation for object-oriented software applications. UML helps the user generate models used to describe the requirements, analysis and design of an object-oriented software system (Gomaa 2011). UML models were used in the design process of the C&DH part of the FSW, known as the Mission

Manager: the software written by the C&DH team that controls the satellite's state and operations.

This chapter is dedicated to presenting, with the help of the created UML drawings, information on the design and implementation of the C&DH FSW. First, the high-level description of the Mission Manager structure will be presented, introducing important concepts used in the FSW such as the mode manager, satellite operational modes, and activities. Then the static model of the flight software will be presented, outlining the system's classes, attributes, operations, and relationships. Finally, the implementation of the FSW's main functionalities are described, including command processing, telemetry management, beaconing, error and fault detection, satellite system recovery, ground pass prediction, and file management.

## 5.1 C&DH Flight Software Design

The C&DH part of the FSW is responsible for any decision-making and command execution that the satellite performs. In other words, the C&DH software determines how the satellite interacts with its environment through controlling its state and actions. If the C&DH software fails in orbit due to a software bug, then the satellite will have little to no functionality, as all other subsystem software is called by the C&DH software on the flight computer. Therefore, it is vital to mission success that the C&DH software design process is thoroughly executed, reviewed and tested to ensure that software bugs are minimized. It also helps to improve reliability if parts of the software have flight heritage. As mentioned in Chapter 4, the software architecture for Hookem was influenced by the flight software for Bevo-1. The following sections explain the design of the Hookem's Mission Manager code by describing its main components.

### 5.1.1 Definitions

First, defining the terms for the different constituents of the C&DH flight software that will be used in the upcoming sections will help in the reader's understanding.

- Mode: A mode is a particular state of the C&DH software system. For each mode, there are certain activities that are allowed to run.
- Transition: A transition is a change from one mode to another that occurs according to specific rules.
- Activity: An activity is a process that occurs in the context of one or more modes; an activity may be continuous (Looped Activity) or may run finitely (Activity).

### 5.1.2 Mode Manager

The C&DH system is the only subsystem that can change the state of the satellite. Therefore, the C&DH flight software was designed as a state machine. The state machine is implemented as a class called *ModeManager*. The *ModeManager* governs the switching of states between satellite software states, called *Modes* (detailed in the following section). The switching between *Modes* is caused by a change in another entity of the software, the transition variable. The transition variable is implemented as an instance of the *Transition* class. The value of the transition variable is altered when certain conditions are met, causing the *ModeManager* to de-activate the current *Mode* that the satellite software is running in, and activate the new *Mode* based on the transition variable's value.

### 5.1.3 Modes

The *ModeManager* places the satellite into a certain state, which is designated as a *Mode* in the C&DH flight software. There are four *Modes* defined for the FSW that the satellite can transition into: Startup (SU), Automatic Command Execution (ACE), Low Power (LP) and Fail Safe (FS). The *Mode* class acts as the base class for four sub-classes, one for each specific mode mentioned. The state chart show in Figure 26 illustrates the different software *Modes,* and the conditions to be met that cause transitions between the *Modes*. The *Modes* are represented as the oval boxes in the figure.

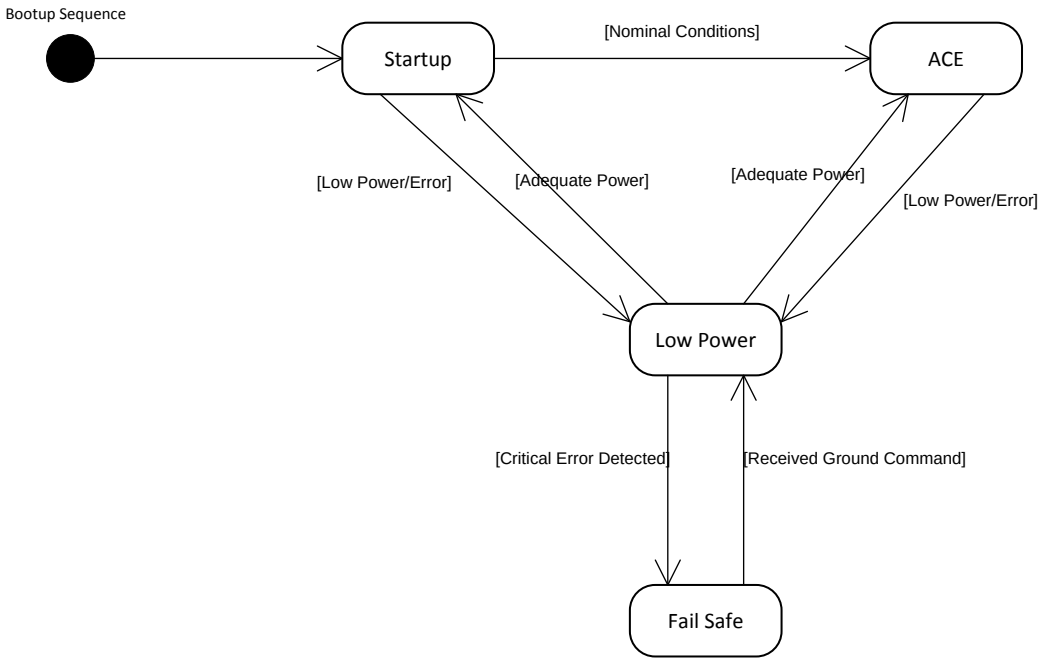


Figure 26. State Diagram for Mode Manager of Hookem Software

The initialization of the state machine occurs once the flight computer has completed its boot sequence. The *ModeManager* then places the satellite into the Startup mode, where it can then transition between the other modes based on the transition variable. The transition variable is represented by the arrows (excluding the arrow from the initialization circle to the Startup mode as this transition is performed outside of the state machine) in Figure 26. The modes depicted in the figure are described in further detail in the subsequent sections. However, before explaining the various FSW modes, the Concept of Operations document is presented.

#### *5.1.3.1 Concept of Operations*

One of the responsibilities of the C&DH team is to create and continuously modify the Concept of Operations (ConOps) documents for all three current missions. It is important for the software developers to understand what events need to take place, and in what order, to execute a successful mission. The ConOps document acts as a detailed summary of the mission in chronological order. In this document, each step of

each phase in the mission is defined. The ConOps documents for the current missions were generated early in the satellite development stage. However, as part of the flight software design process, the original ConOps was modified to incorporate more information related to the FSW and how each step in the mission phase will be accomplished. The modified document included descriptions of the software modes, and classifies the events that occur in each mode. For each step, the document contains details related to the subsystems involved, the power mode the satellite is in, the flight software mode the satellite is in, whether the step is operating autonomously or ground commanded, the criteria for the step to be considered complete, and the verification methods for this criterion. For each mission phase, the expected generated data, and the data sent to the ground is summarized. The mission timeline is continuously updated and refined as more knowledge is gained on the details of the overall mission, and the functionalities and operation of the subsystems. Flow charts for each of the four modes that provide a graphical representation of the events and/or activities that are executed in that mode are listed in the Concept of Operations document.

#### *5.1.3.2 Startup*

The first mode that is entered automatically after the FSW is initiated is the Startup mode. It will also be automatically entered if the satellite reboots for any reason. However, some actions that occur when the satellite is in this mode will not re-occur in for this case. For example, the antennas will be deployed once the satellite enters the Startup mode for the first time, but will not re-deploy if there is a system reboot. Therefore, this mode is slightly different than the other three modes in that it is mostly composed of actions that are a part of a one-time initialization process that occurs when the satellite powers on for the first time. These actions will be skipped if it is determined in the software that the satellite had been previously in operation before the reboot.

The Startup mode is similar for all three missions, with some slight differences. For each mission, a flow chart was created for the Startup mode outlining the essential actions that must be performed in this mode. The flow chart generated for the Bevo-2 mission is shown in Figure 27. The flow chart for the Bevo-2 Startup mode has been

shown instead of that for the ARMADILLO and RACE Startup modes as it is the most complex version of the mode.

For consistency, all flow charts shown in this thesis are for the modes for Bevo-2, and there are only small variations to those generated for the ARMADILLO and RACE missions.

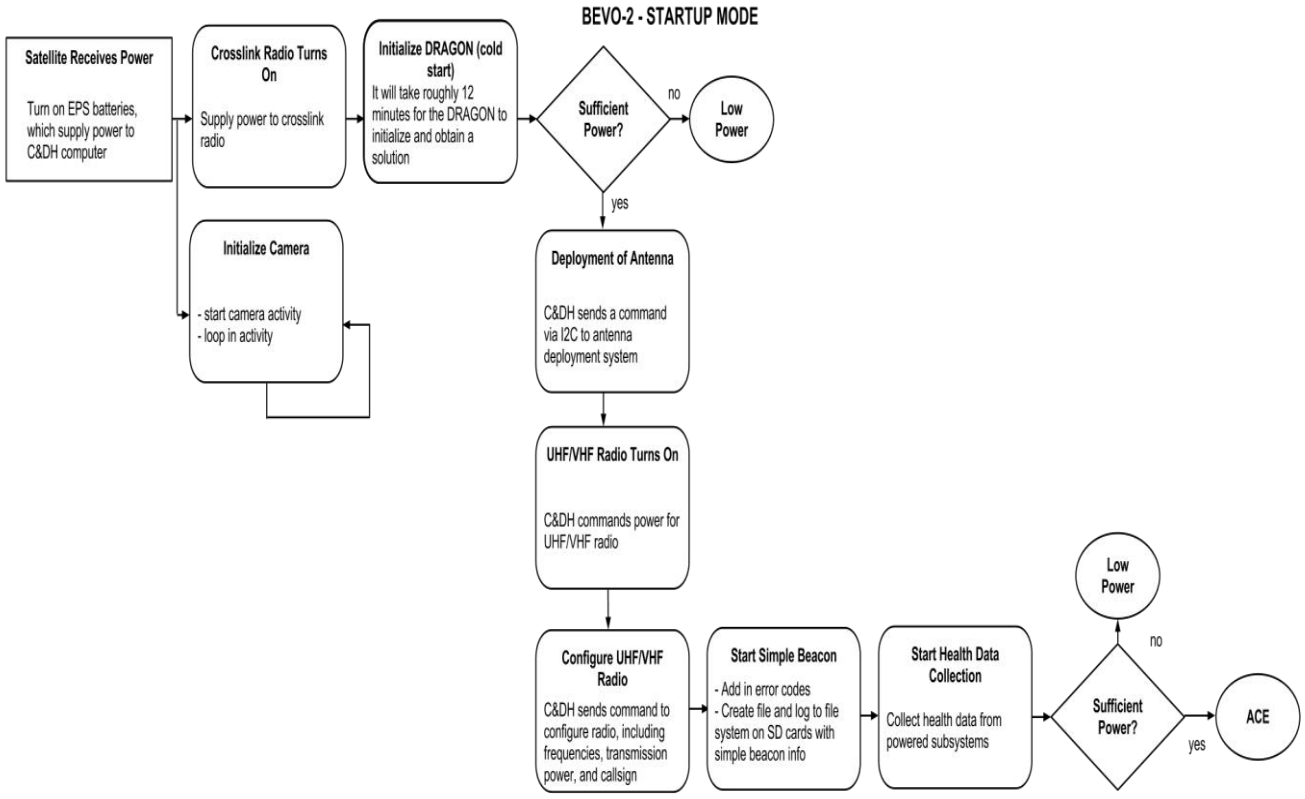


Figure 27. Flow Chart of Hookem's Startup Mode

The satellite receives power from the EPS batteries upon being launched from the ejector. This will allow the Hookem executable to start, and the satellite software to enter the Startup mode. The first actions taken in this mode are to turn on the Crosslink radio, initialize the camera to start taking pictures of AggieSat4, and to power on the DRAGON GPS receiver.

Bevo-2 is launched into orbit from AggieSat4, and these two satellites will separate from each other over time. Therefore, any proximity operations such as crosslinking GPS data between the two satellites, and taking pictures of the other spacecraft, must occur as soon as possible. The crosslink radio needs to be turned on as soon as possible as it is used to crosslink data between the Bevo-2 satellite and the AggieSat4 satellite. A mission objective is to acquire images of AggieSat4 as it moves away from Bevo-2. Therefore, the C&DH software must power on and initialize the camera immediately. The software will then run a looped activity that continuously takes images at a set interval until halted.

Before the satellite can begin transmitting or receiving data, the radio antennas must first be deployed. Attempting any transmission over the radio without first deploying the antenna could damage the hardware. The C&DH system will send a command over the I2C line to deploy the four UHF/VHF antennas. A file is generated to signify a success if the antennas' receive the signal to initiate deployment. The UHF/VHF radio will then be powered on, and be configured with the correct settings, such as the correct uplink and downlink frequencies, baud rates, call signs, and transmission power level.

Once the initial sequence of events has completed and a timeout period before spacecraft radio transmission has been observed, the C&DH can then start the beacon activity. This constitutes transmitting a short message formed by minimal health data over the radio. The health data activity is also commenced at this time. This activity queries the various subsystems that are currently powered on and collects the health data produced by their hardware components. The health data is collected into multiple circular buffers that are large enough to hold data generated during a pre-determined time interval, in the range of 1-2 hours. The contents of the buffers are dumped to a file and transmitted to ground upon receipt of a ground command to do so.

Throughout the sequence of events in the Startup mode, the battery voltage of the EPS system is monitored. If the power level falls below a pre-defined limit, the satellite will transition into Low Power mode automatically. Transitioning back into the Startup

mode occurs once the voltage returns above the limit. If the battery voltage is maintained at an allowable level, then the satellite will transition to ACE mode after all the actions have been accomplished. All transitions between modes will be logged on-board, and can be requested by the ground station if desired.

#### *5.1.3.3 Automatic Command Execution (ACE)*

The ACE mode is the nominal mode of the satellite. This mode involves executing the mission scripts that contain the commands necessary for completion of the mission phases, and executing any commands that are uplinked by the ground station. The flow chart illustrating the main actions performed in ACE mode is shown in Figure 28.

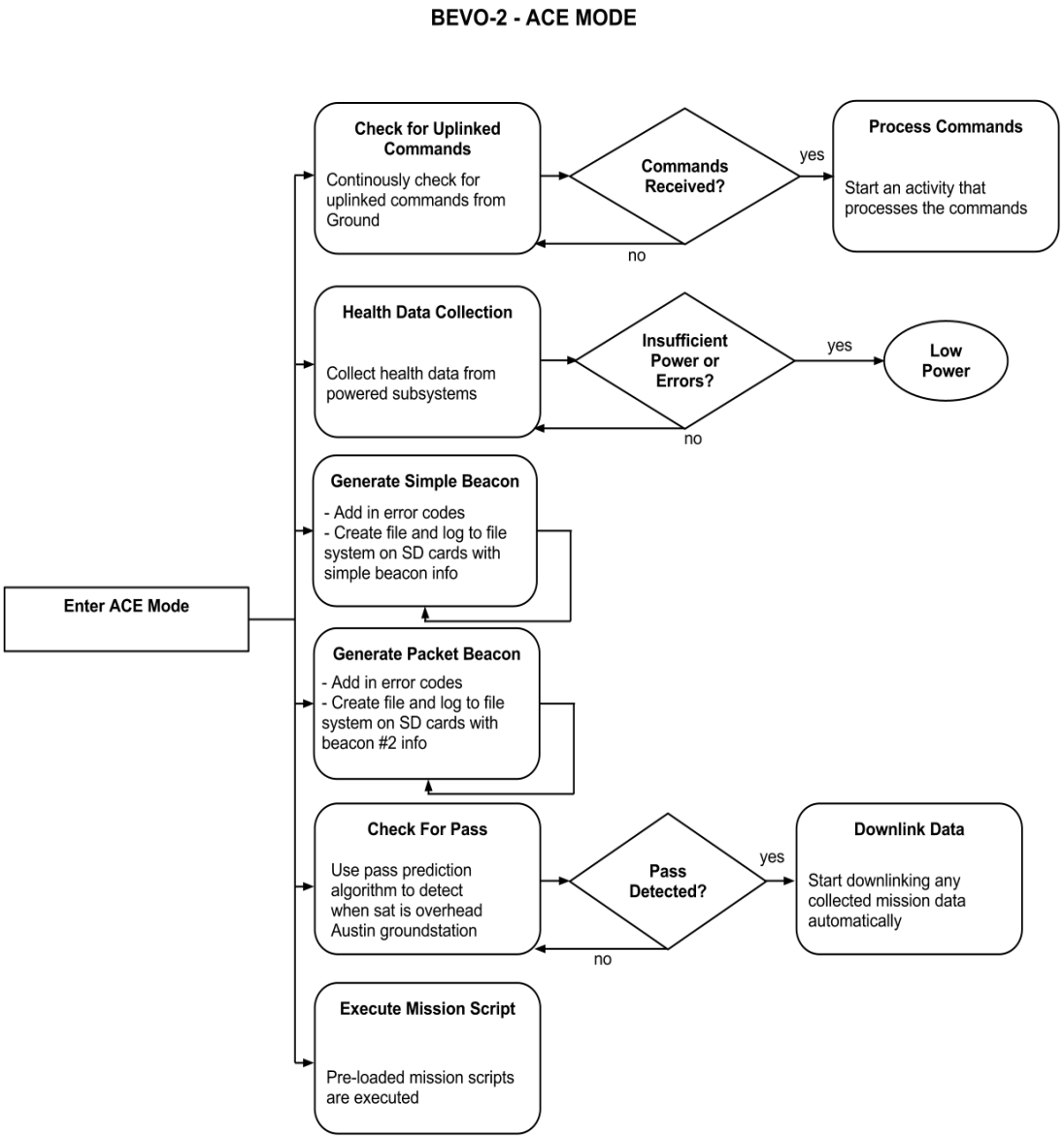


Figure 28. Flow Chart of Hookem's ACE Mode

The ACE mode, unlike the Startup mode, does not have a built-in sequence of actions to be performed (such as the initialization sequence described for the Startup mode). This mode consists entirely of activities that are activated or deactivated based on ground commands, or autonomously based on acquired mission or health data. The two

activities initialized in Startup mode, the beacon and the collection of health data, continue to run in ACE mode. There is one looped activity whose purpose is to listen for any uplink to the satellite. If the radio receives data, then another activity will be initialized to process the commands. Commands, or mission scripts, may either be uplinked from the ground or stored on the SD card before launch. Another action that gets performed in ACE mode is the execution of mission scripts. ACE mode executes commands by transferring them from the current mission script to a command buffer once the uplinked script is validated by the command file validator software component of the FSW (see section 5.3.9).

A transition from ACE mode to Low Power mode occurs if the satellite's battery voltage falls below the pre-set voltage limit, or if an error is detected that disrupts the execution of commands from the ground station or from the current mission script. If the *ModeManager* initiates a transition to the LP mode, it will know which command in the mission script it was processing prior to interruption. Upon return to ACE mode, the software will continue with the interrupted command, unless otherwise directed from a ground command.

#### *5.1.3.4 Low Power*

The LP mode is one of two safe modes for the satellite. LP mode is designed to be a catch-all for error scenarios aboard the satellite, including the spacecraft having insufficient power. Low Power mode allows the satellite to enter a mode where only necessary activities are performed and basic commands can be executed. Only essential subsystems will remain on when the satellite enters this mode. Health data for all powered subsystems continues to be gathered, and the beacon continues to be transmitted. The satellite will listen for any commands that are uplinked from the ground station. However, not all commands received from the ground will be executed by the spacecraft due to the limited power available. Therefore, the list of possible executable commands is restricted compared to that available for execution by ground command in ACE mode.

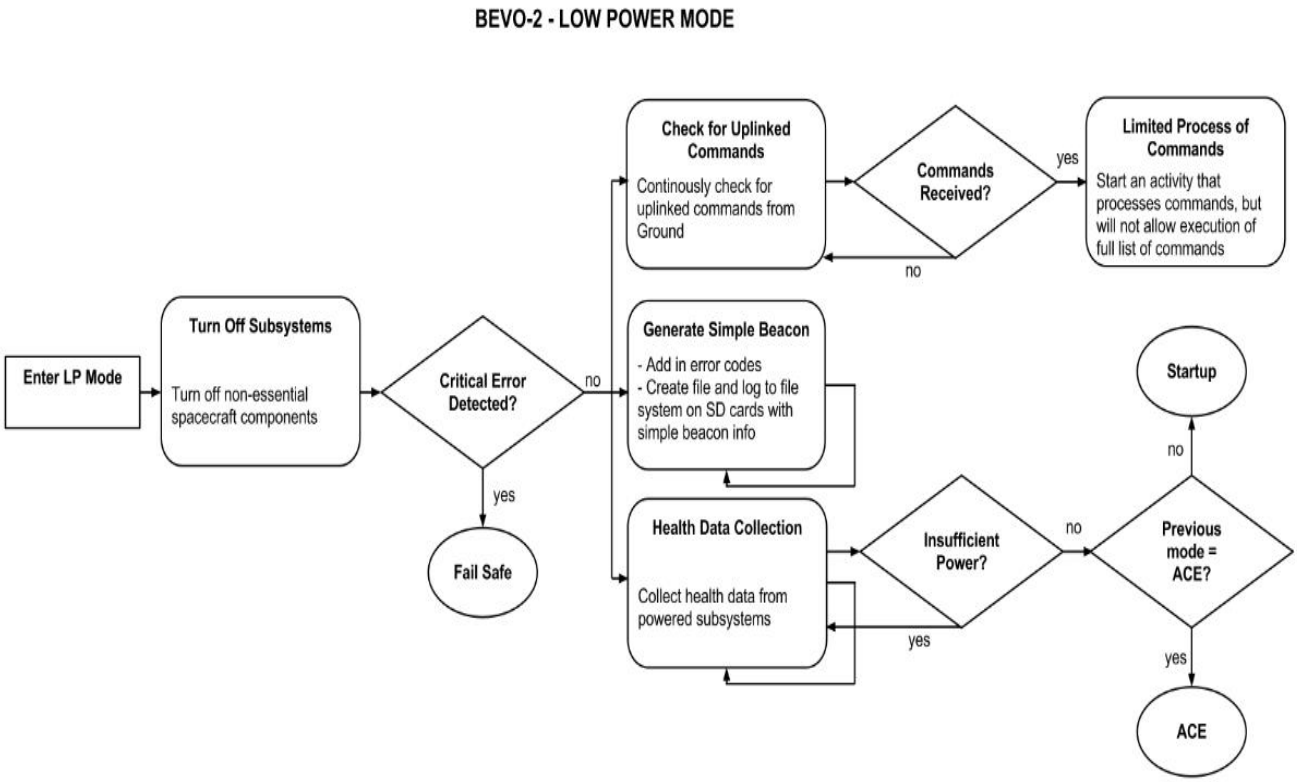


Figure 29. Flow Chart of Hookem's Low Power Mode

The Low Power mode can be entered from all of the other flight software modes. A transition from Startup mode or ACE mode to LP mode can occur when the satellite automatically determines that it does not have enough power. If the satellite is in ACE mode and the EPS voltage is below 6.5 V (initial approximation, this value may change before flight), the satellite will enter Low Power Mode. The voltage level that will cause a transition from Startup mode to LP mode will be slightly larger than that for transitioning from ACE, as it is undesirable for the satellite to start its initialization sequence if it is there will not be enough power to complete the process. The satellite will transition back into its previous mode when the EPS batteries have charged up to the appropriate level for nominal satellite operations. LP mode is designed to be power positive, meaning that any activities performed will use less power than what is being

generated by the solar panels. The current estimate for the appropriate voltage level to cause a transition back into ACE mode is 7.2 V. This is an approximation for the minimal voltage level required to perform scientific operations (operating PDD or FOTON). Again, the voltage level to transition back to Startup from LP will be slightly larger. A transition into LP mode can also occur if any error is autonomously detected by the satellite, if the satellite has no ground communication for a pre-determined length of time while in ACE, or if the spacecraft receives a command by the ground station.

The C&DH will also check for errors that are classified as critical. If the error detected that caused a transition to LP is critical, then the satellite will enter the FS mode.

#### *5.1.3.5 Fail Safe*

The fourth mode for the flight software is Fail Safe mode. Fail Safe mode only runs the minimum necessary activities for spacecraft survival. The flow chart illustrating the operations performed in Fail Safe mode is shown in Figure 30.

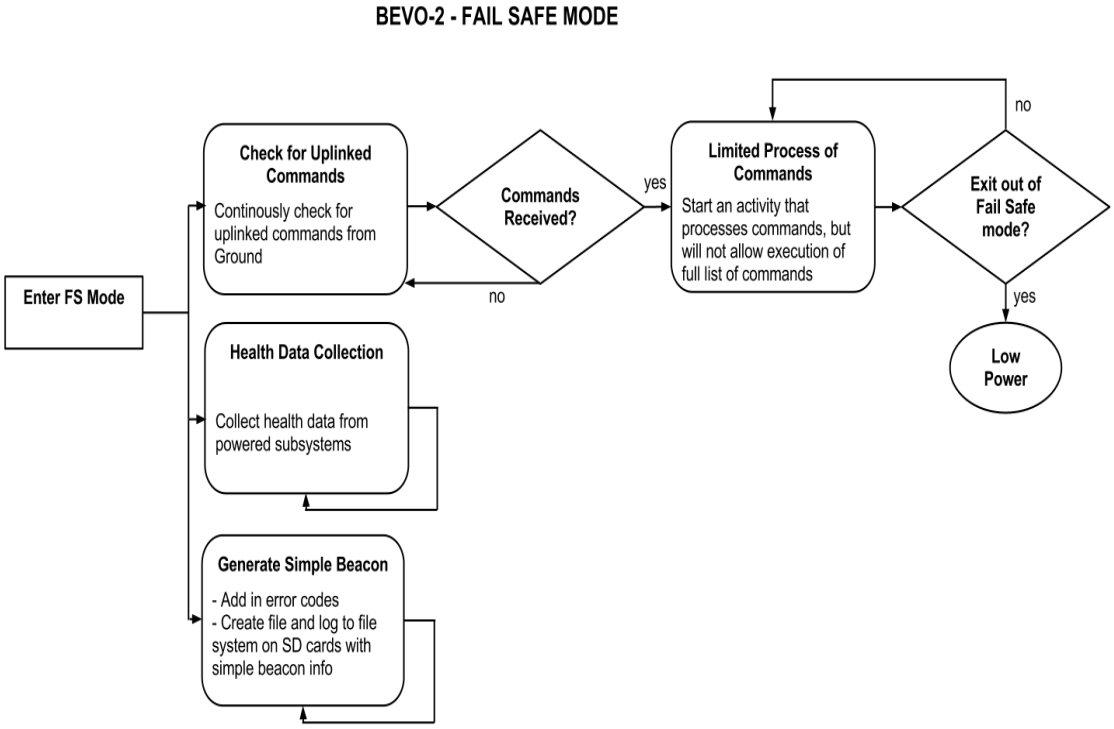


Figure 30. Flow Chart of Hookem's Fail Safe Mode

Fail Safe mode is the only mode that the satellite must transition out of via a ground command. It will always transition out of Fail Safe into Low Power mode, and then can return to either Startup mode or ACE mode. The reason for this approach is to have one software mode which in the case of a critical error, all non-critical components will be powered off. The satellite will remain this way until the mission operators decide the errors have been resolved and it is safe to proceed with the mission. This mode essentially removes all autonomous capabilities of the satellite, with the exception of rebooting in the case of certain detected critical errors, thus preventing any undesired actions performed by the satellite. The mission operators also have the capability to command the satellite to transition into Fail Safe mode. FS mode is designed to be power positive in an average sense, since non-essential satellite subsystems are turned off and are not operating. The spacecraft can only transition out of FS mode through a ground command.

### 5.1.4 Activities

Activities are repeating actions that occur in the FSW. They are implemented as software threads within modes to perform specific tasks. The C&DH FSW has a class named *Activity* which has a child class named *LoopedActivity*. Both of these classes have one main method that includes the tasks for that activity. The *Activity* class will run through the tasks in this method once, whereas the *LoopedActivity* class's method will be repeated at a specified interval based on one of its private variables. Activities can either be started and completed within one mode, or can be transitioned between modes without stopping.

The *Activity* class is essentially a wrapper class for multi-threading. Activities can either be started and completed within one mode, or can be transitioned between modes without stopping. Most of the actions depicted in the flowcharts for the ACE, Low Power and Fail Safe modes are implemented as *Activities*. This cannot be said for the Startup mode, as most of the tasks occurring in this mode need only to be executed once upon launch.

Hookem uses the POSIX threads Pthreads library for software multithreading. Pthreads is a standardized implementation of the C language threads programming interface as specified by the IEEE POSIX 1003 standard that emulates parallelism into the software (Barney 2013). A software program has potential parallelism when procedures can be executed in different orders without changing the result (Buttlar, Farrell and Nichols 1996). Multithreading exploits potential parallelism in the program, where the software developer defines the tasks, or threads, that can run concurrently. A thread is defined as an independent stream of instructions which gets scheduled to run by the operating system (Barney 2013). In other words, a set of lines of code that runs independently of the main program is a thread. The available Pthread library is included through a header and is incorporated into the FSW.

In the flight software, multithreading allows multiple *Activity* objects to run simultaneously in a mode by implementing them as threads. Multithreading permits the C&DH system to execute and monitor multiple subsystems simultaneously, such as

maintaining the watchdog, checking the voltage, and parsing health data from all the subsystems at the same time. Threading allows for activities involving different subsystems to make progress over the same time duration. It does so by enabling the processor and operating system to schedule threads so that every activity is completed. These events can all be executed in an independent manner. However, they can still interact with and are controlled through the FSW modes and the transition variable. All of these actions are performed within the single FSW executable.

Implementing multithreading in the C&DH software does have a downside, however. For a software developer who has never encountered this concept before, it takes time and effort to understand how to properly use it in the code. Multithreading was also used in for Bevo-1, and so a functioning software example used previously in the TSL existed prior to development for Bevo-2, RACE, and ARMADILLO. However, further reading into how to properly implement the standard was necessary. There are also common problems that occur in software programs through oversights or miscomprehension involving threading. A typical issue is mismanaging shared resources and data. This occurs when two threads try to access and modify the same memory space at the same time, which can cause a thread to hang. For example the activity for creating the beacon and the activity for collecting health data may both try to change the value of the transition variable simultaneously. In order to avoid this conflict, tools for proper thread synchronization must be applied, such as mutexes and conditions. A mutex variable acts as a mutually exclusive lock, allowing only the locking thread to access the data. For example, the activity creating the beacon will lock the EPS object when querying it for the battery voltage. If the activity for collecting health data tries to access the EPS during this time, it will only be able to query it after the beacon activity unlocks the object. A C++ wrapper for mutexes was created in the FSW to simplify its usage. A condition variable can be used by a calling thread to wait until a condition is met before executing its tasks. Meanwhile, other threads may use those resources the calling thread will use, and may signal the condition variable that the calling thread is waiting on.

## 5.2 CLASS DIAGRAM

Class diagrams are used to depict a static structural model of the software program. A class diagram for the FSW was formed as part of Hookem's design process, and can be found in the Appendix. The diagram illustrates the classes that compose Hookem for Bevo-2, their attributes, operations and relationships (The University of British Columbia 2003).

The class at the top of the diagram is the *ModeManager* class. As mentioned earlier, the *Mode* class is a parent class to four child classes representing the four operational modes of the FSW. There exists an *Activity* class that acts as the parent class to *LoopedActivity*. The *LoopedActivity* class allows for a procedure to occur repeatedly after a sleep in the thread for the amount of seconds corresponding to its timeout attribute. The high level C++ classes for the subsystems are also shown in the class diagrams, labeled *HeliumDriver*, *GPS*, *Xlink*, *EPSFunctions*, and *ADC*, but with no class details. This encapsulation is done to symbolize the existence of these classes in Hookem, but also showing that they act as black boxes to the Mission Manager software.

## 5.3 Main Functionalities of Flight Software

In the following section, some of the main functionalities that define the C&DH FSW are described. These functionalities involve the high level responsibilities of the C&DH subsystem in order to maintain the operations of the satellite. Explanations are given for the C&DH software main attributes which include ground satellite command processing and communication methods, mission scripts, file management, beaconing, error handling and recording, command logging, automatic downlinking, telemetry management, startup sequence, and system recovery.

### 5.3.1 Ground to Satellite Command Processing

One of the primary roles for the C&DH system is to handle and process commands received from the ground station. The C&DH system must be able to interpret uplinked commands, and perform the corresponding actions in a timely manner. For the satellites built in the TSL, the COM system is responsible for reading and writing any data sent to or from the satellite from the ground station. However, there is no command processing or interpretation performed by the COM system. Any data received by the radio will be written as is into a designated file stored on-board. It is then the C&DH system that opens this file and parses the data to acquire the desired commands. The command and data flow to the required subsystems for command execution and response is controlled by C&DH.

A large portion of the design of the FSW involved defining the ground station to satellite command processing architecture. This task was done in collaboration with the communications (COM) system, as the C&DH system relies on the COM system to receive and transmit data that is produced by the satellite with minimal loss, which allows for efficient and correct command processing.

There are several activities that each play a role in executing commands that are uplinked from the ground station, namely: *CheckUplinkActivity*, *CommSchedulerActivity*, *MissionScriptActivity*, *InterruptBufferActivity*, and *OpsCommands*. Each of these activities is a class in the FSW that has one object created upon execution, and that does

not get destroyed during runtime. Figure 31 illustrates a sequence diagram showing the interaction of these activities involved in the satellite's execution of uplinked commands. In the figure, the boxes are instances of the classes involved in the execution of uplinked commands. The lifelines of the instances are drawn as dashed lines, and the messages between classes or messages to one class itself are represented by the arrows.

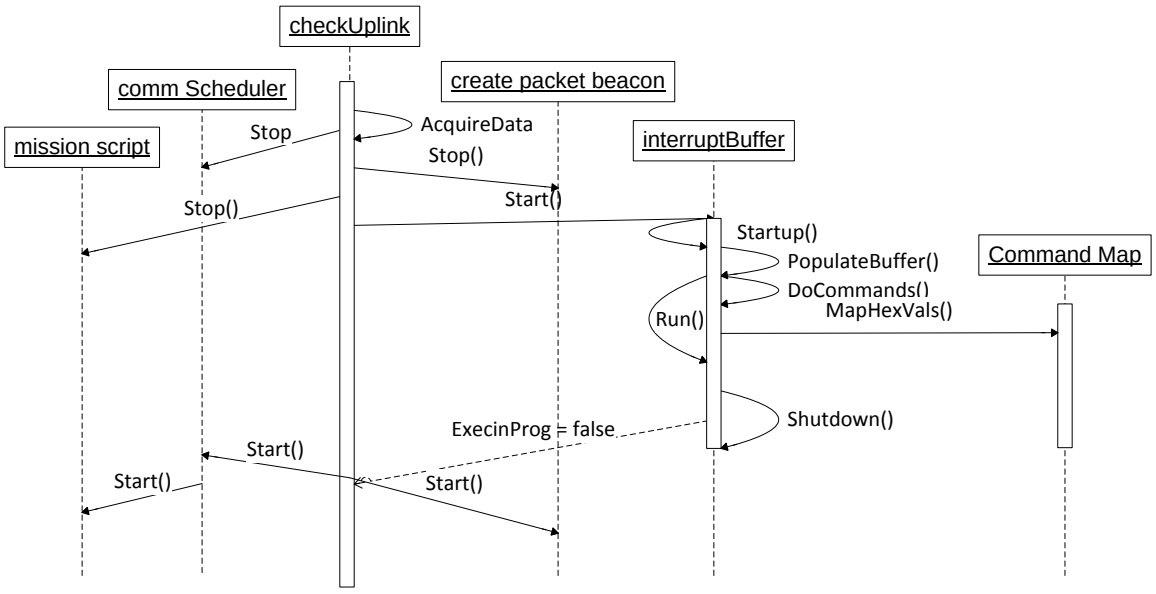


Figure 31. Sequence Diagram for On-Board Command Processing

The *CheckUplinkActivity* (CUA) is the primary activity involved in querying the UHF/VHF radio to determine if any new input has been received from ground. Currently, this activity runs as a *LoopedActivity*, continuously calling a high-level method of the UHF/VHF radio software that will indicate if there is a new file sent by the ground station that contains commands. If it is determined that new data has been uplinked from the ground station, the activity will first halt the beacon, the mission script scheduler (the *CommScheduler* object in Figure 31), and the mission script executor (the mission script object in Figure 31). The CUA will then make a call to start the *InterruptBufferActivity* (IBA) thread. The CUA activity will re-activate the beacon and mission script-related

activities only after it receives a signal that the IBA thread has stopped running, or if no new commands have been received by the satellite for longer than a time period that is hard coded into the software.

The IBA thread is responsible for copying commands from a file into a temporary buffer, and for calling the thread that maps the commands into the corresponding satellite actions, known as *OpsCommands* (represented by the *CommandMap* object in the figure). This activity can be activated in two different manners: by the CUA in the manner mentioned previously to handle incoming commands from the ground, or by the *CommandScheduler* activity to execute time-stamped mission scripts. The first method is described here, whereas the second method is described in the Mission Scripts section 5.3.2. Similar to both methods of activation is the copying of the commands from the file into a temporary buffer.

Any file uplinked from ground will have a header line at the beginning of the file which indicates whether the commands are to be executed immediately (known as a command file), to be treated as the contents of the new replacement mission script, or to be treated as contents of a time-stamped mission script for execution at a future time. When the IBA thread is started by the CUA object, the commands from the uplinked file are copied into the IBA object's temporary buffer, and the header line is checked to determine the proceeding command processing steps. If the header indicates that the commands are to be executed immediately, the activity will then make one call for each command to the method of the *CommandMap* object until all commands have been executed. If the header indicates that the file is a new mission script, the IBA object will place the commands into a mission file for use either by the Mission Script Activity (mission script object in Figure 31) when it is reactivated. If the uplinked file contains the header indicating a time-stamped mission script, the commands are stored and will be used in the future by the same IBA object.

#### *5.3.1.1 Command Messages and Command Logic*

One major difference between the ground to satellite command processing architecture of Bevo-1 and that for the current missions is the communications protocol. Namely, the format of the messages sent to and from the satellite and the ground station is different. Bevo-1, for the majority, used ASCII text for its on-board logs, telemetry and command files. However, the current missions' scripts, telemetry, and almost all on-board logs are communicated in binary format.

For Blackbird, there was a set of predefined nouns and verbs that formed the commands that Bevo-1 was able to interpret. The nouns and verbs were hardcoded into the software. The ground station would uplink command files with each command implemented as a C++ String object. Each string contained a noun and a verb, and could contain parameters depending on the command, for example "DRAGON TURNON" or "RUNDGN SETDURATION 10". The noun in the command corresponded to a particular device that could be controlled in software, such as "CLOCK", "GPS", or "RADIO", or an activity in the software that could be started or stopped, such as "MONPWR" corresponding to the monitor power activity. The verb would then indicate what action to perform for that noun. Each noun could have several verbs, and through using a specific combination, would map to a different satellite operation pertaining to the corresponding device or activity.

This type of String command processing was abandoned for the current missions in favour of binary commands. Instead of using "nouns" and "verbs", C&DH op-codes are used to distinguish between commands that the C&DH can interpret. Each op-code, implemented as a distinct byte, can be related as a specific combination of a noun and a verb in the form of command processing used in Bevo-1. Therefore, each subsystem has many op-codes that correspond to its related operations. Similar to Blackbird, Hookem's op-codes are also hard coded into the software in a class called OpsCommands. This command processor class can also expect a certain number of parameters to be included with the op-code to form one command. Each command is SLIP framed before being transmitted (see section 5.3.1.3).

The main advantage of implementing binary commands rather than C++ String commands is the savings in resources required for data transfer between the satellite and ground station. The size of uplink and downlink files is reduced, thereby minimizing the time required to communicate data with the ground station. This is advantageous as there is a limited time period for each pass when communication with the ground station is possible.

#### *5.3.1.2 Command Management*

The C&DH system is responsible for populating the list of allowable op-code commands that the satellite can process. All the commands and their corresponding op-code byte values are kept in a spreadsheet available to the subsystem lead engineers in the TSL. The list of commands are segmented into categories including system level, C&DH high-level, subsystem level, and mission specific commands.

The system level commands include those which are the required for basic satellite operability from the ground station, such as the indicator op-code distinguishing between commands for immediate execution and mission scripts, resetting the C&DH computer, transitioning out of Fail Safe mode, and requesting an acknowledgement from the satellite. C&DH high-level commands are the commands used in the FSW for activity configuration, for example starting and stopping activities, and for data related actions, such as requesting certain files to be downlinked. The rest of the commands map to the high-level functions included in each subsystem's software interface with the C&DH as described in the respective ICDs. By allowing access to only the subsystem high-level functionality, most of the subsystem capabilities can be utilized by the ground station operators. However, by not including access to every possible satellite subsystem function, a level of abstraction between the satellite software and the ground station is established that decreases the complexity of the ground station to satellite command processing.

#### *5.3.1.3 SLIP encapsulation*

As the C&DH system is transmitting and receiving binary data, it was evident that the subsystem required a communication protocol for interactions with the ground station. A version of the Serial Line Internet Protocol (SLIP) was implemented by the C&DH subsystem in order to encapsulate all data that was sent to and from the ground station. SLIP is a practical standard mainly used for point-to-point serial connections running TCP/IP (The University of British Columbia 2003). It works by framing the packets on a serial line through a definition of a sequence of characters. It is important to note that SLIP is the protocol used internally between the C&DH and the ground station software, and is implemented to encapsulate mission data in the high-level C&DH software before the COM software is called to actually transmit or receive. The SLIP encapsulation is entirely external to any data or packet manipulation the COM executes during data transmission or receipt.

SLIP was the protocol of choice for the C&DH system for several reasons. First, the data shared between the C&DH subsystem and the ADC subsystem will follow the NanoSatellite Protocol. This protocol was developed by the Space Flight Laboratory at the University of Toronto, and is based on the Simple Serial Protocol (SSP) (Sinclair Interplanetary 2008). The ADC subsystem uses NSP as the reaction wheels are from Sinclair Interplanetary and are configured for NSP communications. The NSP messages from the ADC computer to the ADC devices are encapsulated into packets using SLIP framing. It was through discussions with the ADC team that the C&DH system decided on using SLIP framing for the satellite flight computer to ground station communication. Another reason for using SLIP encapsulation is that it is easy to implement as it has a very low level of complexity. There is no error detection or correction, compression mechanisms, or packet type identification. However, the C&DH system did not need the protocol to have error detection properties, as the COM system handles packet and data authenticity in the form of checksums. Therefore, minimal effort by the C&DH team was required to implement SLIP framing for its needs.

### 5.3.2 Mission Scripts

The list of tasks to be performed by the satellite in ACE mode for completion of the mission objectives is known as a mission script. A mission script is formed by a set of commands that together will provide the satellite with the instructions necessary to complete a mission phase. There will be one mission script for each mission phase that will be loaded onto the satellite before launch. Therefore, the satellite will already have the sequential list of commands to execute each mission phase before being in communication with the ground station. A diagram depicting the pre-loaded mission scripts for the ARMADILLO mission is shown in Figure 32.

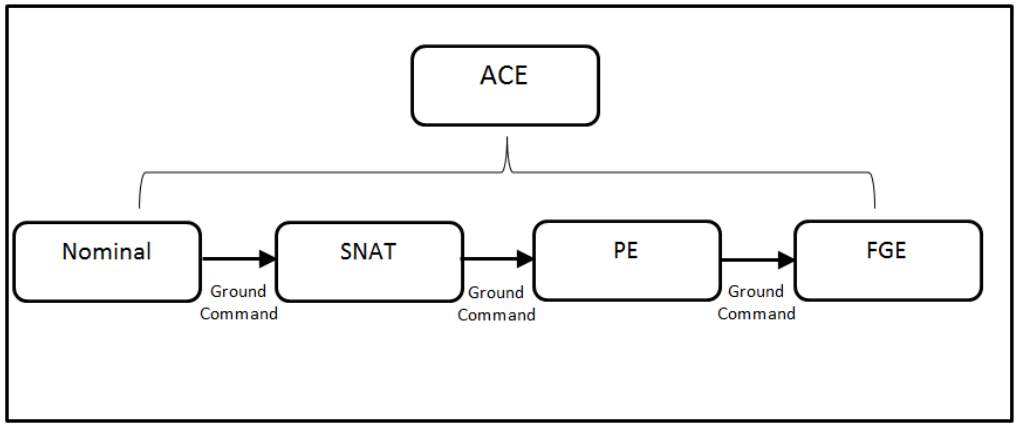


Figure 32. Sequence for Execution of Pre-Loaded Mission Scripts for ARMADILLO Mission

For the ARMADILLO mission, there will be four mission scripts pre-loaded onto the satellite, Nominal, Sensors and Actuators Test (SNAT), PDD Experiment (PE), and FOTON GPS Experiment (FGE), all of which are executed in ACE mode. As shown in the diagram, after completing the tasks in the current mission script, the satellite can only move onto the next script, or repeat a previously executed script, after receiving a ground command enabling it to do so. This is done in order to allow the operators of the ground station the time to ensure that all the necessary data has been collected, downlinked and received for that mission phase before moving on to another script.

In addition to the pre-loaded mission scripts, the satellite will have the capability to receive and execute uplinked mission scripts while in orbit. The uplinked mission scripts can be one of two types, a mission script to replace that which is currently getting executed, or a time-stamped mission script to be executed at a later time. The satellite can distinguish between these two types of scripts based on the header line at the beginning of the uplinked file. The purpose of being able to uplink mission scripts is the flexibility of replacing the pre-loaded scripts or adding entirely new mission scripts if this decision is made based on the mission data or any possible errors encountered during operations.

The sequence diagram from Figure 33 shows how the different classes involved with replacing the current mission script with a newly uplinked script interact to perform this task. The mission script scheduler (depicted in the figure as the *commScheduler* object) is responsible for the management of the mission scripts. Once this object receives the signal from the *check Uplink* object to start after the communication with the ground station during a pass is concluded, it will set the *missionscript* activity's file to the new mission script, and will start the activity for execution.

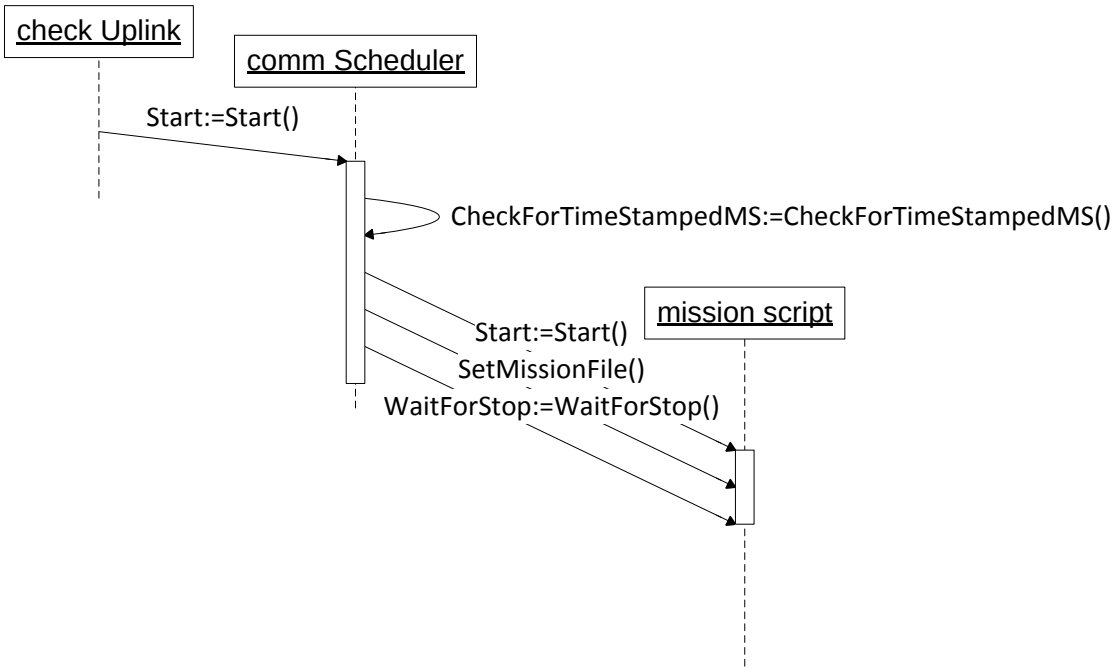


Figure 33. Sequence Diagram for Execution of the Replacement of the Current Mission Script

Time-stamped mission scripts provide a level of autonomy to the FSW, and therefore the spacecraft. The satellite is responsible for determining when it is time to execute the mission script after receiving it. Therefore, rather than the satellite only executing commands when it is in the range of the ground station, future commands can be uploaded at the convenience of the ground station operators, and they can be executed at times without the requirement of ground communication. The sequence diagram in Figure 34 shows the process for uplinking and executing a time-stamped mission script. If the c*ommScheduler* object determines that the time-stamped mission script time matches the satellite time, it will shut down the *mission script* activity executing the current mission script, and start up the IBA. The *interruptBuffer* activity will then copy the commands from the time-stamped mission script into a buffer for execution. Once

these commands have been processed, the *commScheduler* object is free to restart the mission script activity and return back to nominal operations.

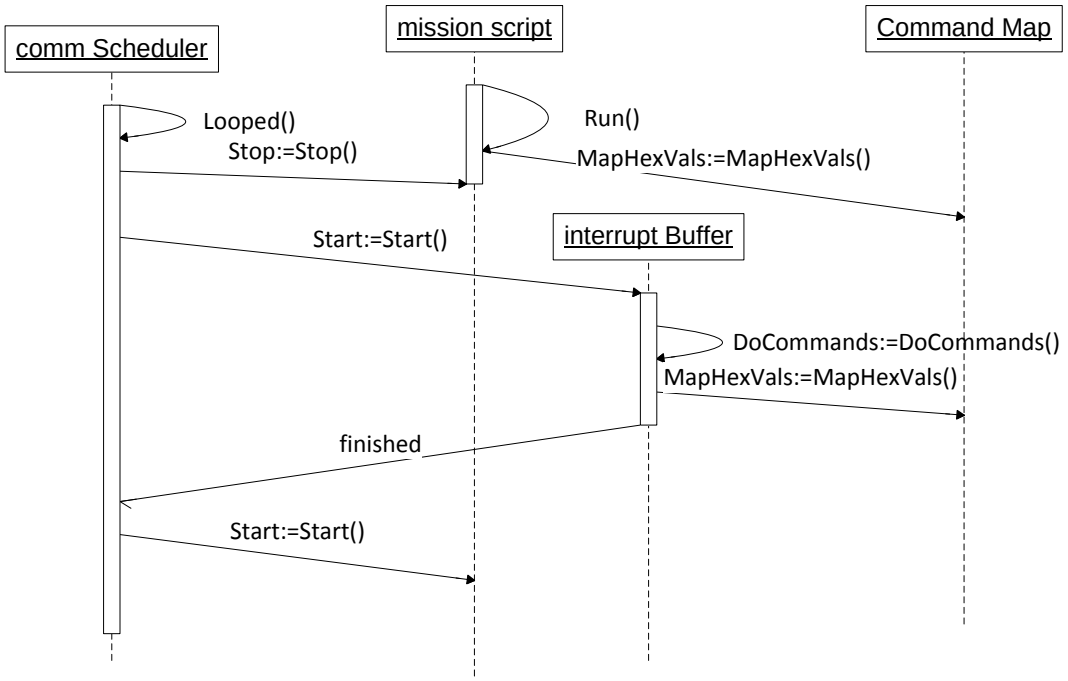


Figure 34. Sequence Diagram for Execution of a Time-stamped Mission Script

### 5.3.3 Pass Prediction-Based Automatic Downlinking

A distinguishing characteristic of the FSW is the automatic downlinking of data based on on-board pass prediction. An activity running in Hookem, the *CheckForPassActivity* class will be responsible for predicting the time until the satellite's next pass over the UT-Austin ground station. After the predicted amount of time has passed, *CheckForPassActivity* will start the activity responsible for downlinking the files listed in the file request to ground. Most of the code for this class, including that which gets executed in a loop for this thread and that which involves predicting when a pass will occur, has been inherited from Blackbird. The Blackbird software would automatically downlink DRAGON data when a pass was predicted on-board. However, Hookem will

downlink any data files included in the file request that has not yet been downlinked, regardless of which subsystem generates them.

A state diagram depicting the main events that occur in the looped method of the *CheckForPassActivity* class is shown in Figure 35.

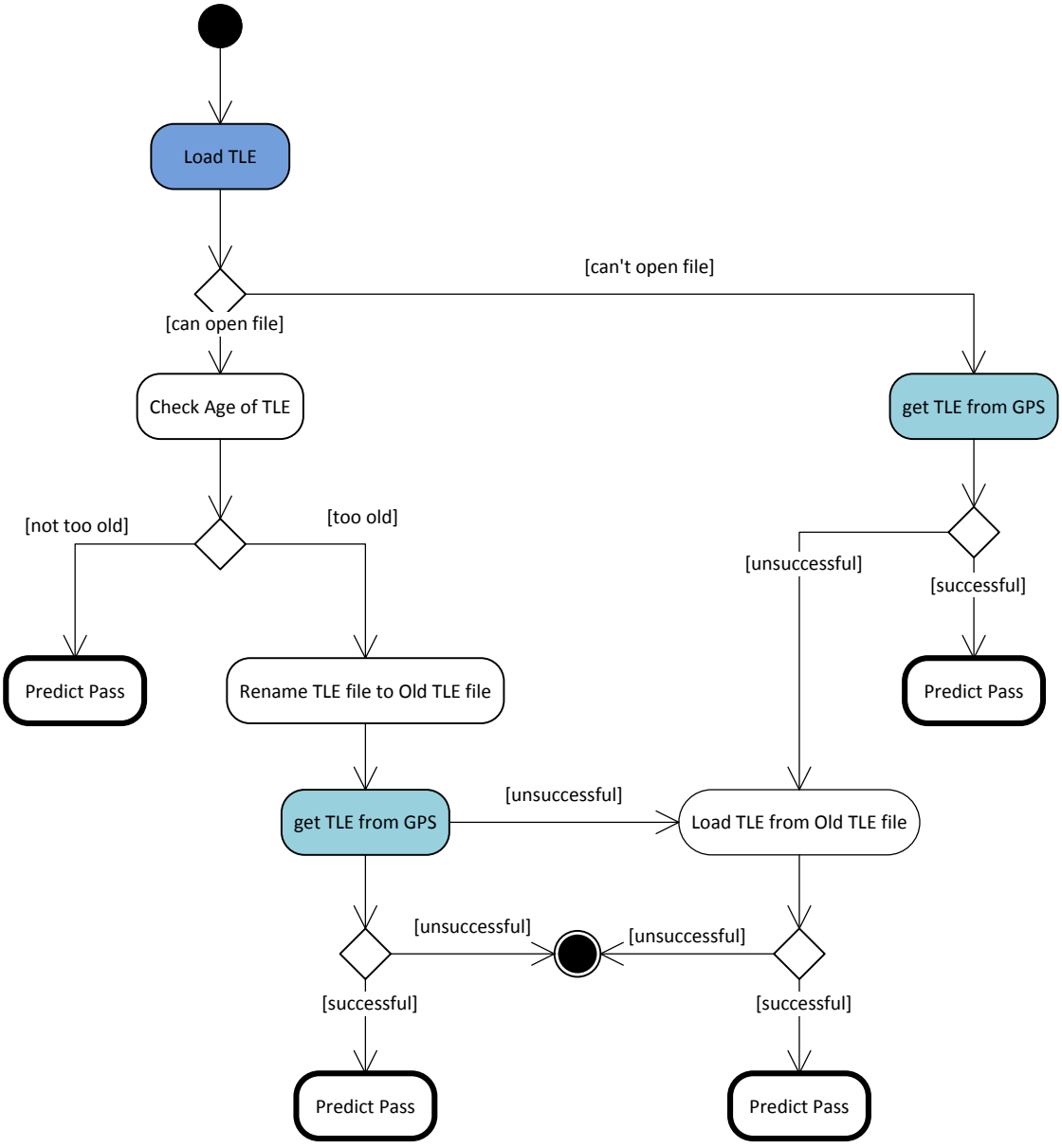


Figure 35. Flow Chart for Hookem's Pass Prediction Feature

One of the tasks of the GPS is to update a file stored on-board with the Two-Line Element (TLE) set when the C&DH requests it. There are two files kept on-board for storing TLEs generated by the GPS, one file to keep the latest TLE, and another that stores the most recent outdated TLE. In the *CheckForPassActivity*, the C&DH will retrieve both lines of the latest TLE. If the retrieval is successful, and the TLE has been updated within the past day, then a method will be called to predict the next pass using this information. Whereas if the TLE is outdated by more than 1 day, then it will be renamed as the outdated TLE and the C&DH will command the GPS to obtain the required data and form a new TLE. This newly created TLE can then be used to predict the next pass. The C&DH will also command the GPS to obtain a new TLE in the case that the system was unsuccessful in retrieving the latest TLE. If no solution is acquired by the GPS, the pass prediction method will be called using the TLE contained in the outdated TLE file.

The *CheckForPassActivity* class uses one method for predicting when the pass commences and ends. The method makes use of a package of C++ files developed by Henry that is an implementation of North American Aerospace Defense (NORAD) Command's Simplified General Perturbations-4 (SGP-4) orbital models for near-Earth objects, commonly used in satellite tracking software (Henry 2013). The package contains supporting classes that provide the capability of propagating a satellite's orbit and calculating predictions of orbital parameters such as azimuth, elevation, range, and range rate using TLE data gathered by the GPS.

The pass prediction method implemented in the FSW first creates an object representing UT-Austin's ground station with its known location as its attribute, and an object representing the satellite itself with the TLE data. The times until the next Acquisition of Signal (AOS) and until the next Loss of Signal (LOS) are calculated using the attributes of the ground station and the satellite and the current satellite time. The algorithms that calculate AOS and LOS are adopted from PREDICT, an open-source software that provides real-time satellite tracking and satellite orbital predictions (Magliacane 2013). The algorithms determine the time until the satellite rises a certain

level above and below the horizon of the ground station. The activity will then sleep until the time of AOS is reached, at which point it will start the downlinking thread of Hookem. However, any files automatically downlinked off the file request list will not be marked as "successfully downlinked" until the satellite receives a confirmation from ground that they have been received without loss of data. The pass prediction activity can be de-activated through a ground command if it is decided by the ground station to turn off this feature.

As this automatic downlinking capability adds a level of complexity to the FSW, a significant amount of testing must be performed on the software responsible for on-board pass prediction and for the proper interaction between the commencement of file transmission through pass prediction and commencement through ground command request. At the time of publication, both the *CheckForPassActivity* and the *CommWithGroundActivity* classes, the latter being the thread responsible for downlinking the files on the file request, have been unit tested. However, scenario tests must be conducted with test cases for various potential situations that can be encountered during flight involving the pass prediction capability. These tests are a part of the future work that will be conducted on the C&DH system before completion.

### 5.3.4 File management

The Kernel and the root file system (Rootfs) are stored on the NAND flash memory of the flight computer. The Rootfs is the Linux root file directory for the FSW. The Kernel and the Rootfs, including Hookem, are stored and booted from the NAND flash, as opposed to the NOR flash or the SD card. The NOR flash is not used as the FSW file in its current state is 4 MB in size and would not fit on the NOR flash. The FSW is not stored on the SD card in case there are any problems encountered related to mounting functions, or corrupted SD cards, and therefore any problems with the SD card will not result in complete mission failure. A discussion on how the backup copies of the FSW will be used for software verification and validation at runtime is discussed in section 5.3.8.

All mission data generated by the payloads for each mission, as well as all health data, will be stored on the external SD card. Separate directories will be created on the SD card to maintain an organized file system to facilitate the transmission and storage of generated mission data. The naming convention for generated files will be kept uniform across subsystems to minimize differences in the manner of requesting files from the ground. To prevent the SD card from being a single-point failure for the data storage system, a spacecraft command will be able to change the future data storage location from the SD card to the NAND flash on the LPC3250.

### 5.3.5 Beaconing

One of the lessons learned from the FASTRAC mission was to transmit two beacons rather than the one beacon. The FASTRAC satellites transmitted a beacon of 126 bytes in size via a UHF/VHF radio every two minutes while waiting for a connect request from the ground station (Greenbaum 2006). However, the FASTRAC team suggested that for future missions, it would be helpful to have two beacons, a continuous wave (CW) beacon and a packet beacon. The CW beacon includes very basic satellite health data whose primary purpose is to provide the ground station with a quick indication that the satellite is operational. The packet beacon includes more in depth satellite health data and can be downlinked by amateur radio users outside of the main ground station reception area.

Currently, the design is for the current missions to have two separate beacons. The first beacon type is designated as the simple CW beacon, and will be transmitted using Morse code at a rate of roughly one beacon per minute. However, producing the Morse code beacon is a time consuming software project that has not yet been started by the Communications team. Therefore, it is unknown whether this capability will be implemented in time for the launch of Bevo-2 and RACE, but should be ready for the ARMADILLO mission. The packet beacon will be transmitted at a slower rate than the simple beacon, at approximately once every two to three minutes. It will be transmitted using the radio in the same manner as the other health data files and mission telemetry

data. The file is broken down into smaller packets which get sent concurrently. The file can be re-requested if not correctly transmitted. Data included in the beacon are the satellite on-board time, the latest satellite position and velocity from the GPS, the latest attitude data from the ADC system, and an indicator to bring to attention to the ground station if any new errors have been generated by the FSW since the previous beacon.

### 5.3.6 Telemetry management

Telemetry collected on-board is managed through a health data looped activity. This activity is responsible for querying all subsystems for their health data after a specific time interval. The health data is collected into circular buffers grouped together into a C++ structure, with all buffers using the same beginning and ending pointers. The length of the circular buffers and the time between loops of the activity was chosen so that the satellite can store roughly three months of health data. The health data will periodically be written into a file stored on-board in case of a failure causing the satellite flight computer to reset and lose the data.

The activity is implemented such that the health data from all subsystems is collected into the buffers at the same rate. A timestamp is also stored into a buffer in the structure to accompany the health data.

A command can be sent from the ground station requesting the downlink of the health data. The activity then puts a pause to its data collection, and dumps the contents of its circular buffers into a file, with each line as one set of health data for a certain timestamp. One addition to this activity will be to handle requests for sub-sets of this data from ground, rather than always dumping the entire content to a file for transmission. This file is then transmitted to ground via the UHF/VHF radio, and data collection continues in the looped activity. The satellite will still be able to receive and execute commands while this data is being downlinked.

Health data files will also be generated and stored on board from the contents of the circular buffer in this activity. When the circular buffers reach capacity, its contents will be dumped into a health file for on-board storage automatically. The satellite will

continuously maintain and store two files worth of health data (approximately six months of health data) independent of the file generated upon a health data downlink request. This extra storage of health data will ensure that if an error occurred on-board at a time earlier than three months before the ground station requested a health data downlink, then the data collected at that time is not completely lost by the circular implementation of the health buffers as it is stored on-board. The ground station will then have the capability of requesting health data further back than then the three months that are stored in the circular buffers if this is needed.

### 5.3.7 Command logging

The C&DH part of the FSW is responsible for maintaining an on-board log. The log is implemented in the FSW using the SQLite 3 software library, which is used for implementation of databases in many embedded devices with constrained memory. Therefore, the log is kept in non-volatile memory, and its data can be accessed even after a possible satellite reset without loss. Both the main flight computer and the ADC computer will use a log database to record important events occurring on the satellite that can be used to help determine errors experienced during testing or flight.

Particularly for the C&DH software, the log will be used to track all commands that get uplinked from the ground station, and the operational mode that the satellite has entered. Each command that the satellite can interpret and has attempted to execute will be logged with a timestamp generated based on the satellite's time. The log entry for a command will also indicate if the command execution was a success or a failure based on the return information from the component involved to the calling C&DH thread, *OpsCommands*. Log entries are also formed when the satellite transitions from its current software mode to another. An entry created for this type of event also includes the transition variable's value, which reveals more information as to the cause of the transition between modes. Ranges of entries in the log database can be requested by the ground station via ground command. The ground station can then use this information to

help debug satellite operational errors, and can determine which commands were not received, or incorrectly executed by the satellite.

### 5.3.8 Satellite Software Redundancy

One major concern of having the FSW encompassed into one program is possible corruption of the executable file. Single-event effects can be caused by ionizing radiation damage to the flight computer and other electronic components of the satellite due to the space environment, and must be considered when developing the C&DH subsystem. The events include single-event upsets (SEU), single-event latchups (SEL), and single-event burnouts (SEB) (Larson and Wertz 2006). SEUs are the least damaging of the three categories, but involve soft errors such as a bit flip in memory cells or registers. An SEL can be a soft or hard error, and can potentially damage the affected device due to a hang up and a resulting excessive current draw that is not dissipated. SEBs are types of events that cause the device to fail permanently.

The radiation the satellite experiences in orbit is a reason behind having a run-time process for validating the FSW executable on the C&DH system before it is executed after launch. Redundancy of the flight software has been used to mitigate the effects of program corruption. Two redundant copies in addition to the primary copy of the FSW executable are stored on-board--one copy on the NAND flash memory and the other on the SD card. Upon boot up of the flight computer, a script will be executed that will check the integrity of the primary FSW executable against the two backup copies. The integrity checking process is outlined in the diagram shown in Figure 36.

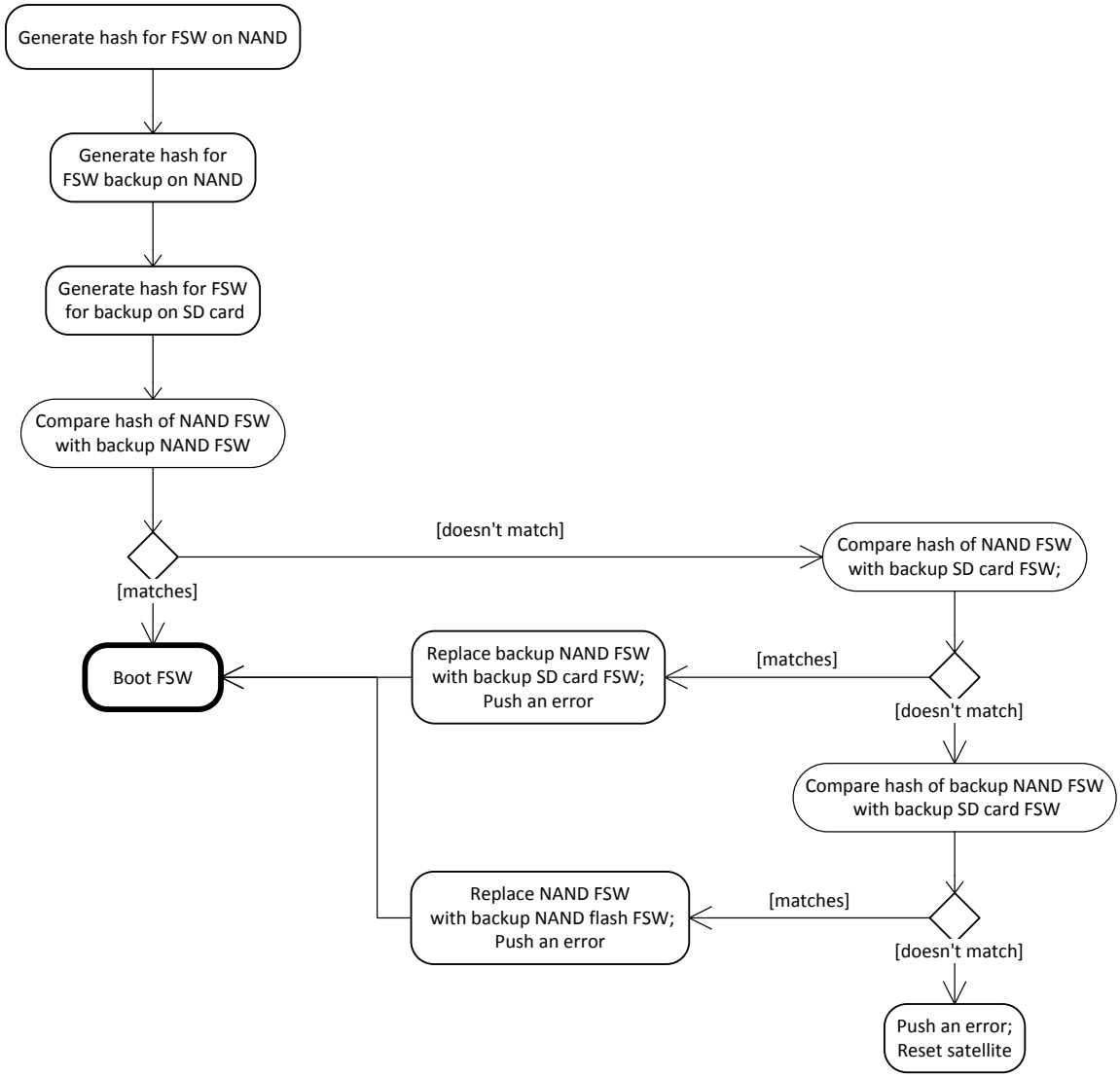


Figure 36. Flow Chart of Process for FSW Integrity Checker

The validation of the integrity of the FSW is performed through a cryptographic hash function that generates a hash string for each of the three executables. A cryptographic hash function is an authentication algorithm that is used to check the integrity of information by generating a message code, or hash, of fixed length using the information as the input to the function. A hash function is a one-way function, meaning

it is difficult to decipher and invert the hash to retrieve the original transmitted information. A hash function can also be collision-free, where it is infeasible to have two sets of information that generate equal hashes. Thus, hash functions are widely used in software integrity applications as they generate unique digital signatures for software programs (Khan, et al. 2010). Any bit flip caused by radiation will result in a different hash for that software. The hash function's method of generating the hash itself can vary. Examples of commonly used hash functions are MD2, MD5, and SHA-1. MD5 was ruled out for the integrity software algorithm as, even though it is a strengthened version of MD4, the hash function was found to not be collision-free, and therefore should be avoided for digital signature applications such as this one (Wang and Hongbo 2005). Therefore, out of the three available hash methods, SHA-1 was chosen to be the most suitable. The integrity checker software, along with other components of the FSW, was developed using the QT framework, as it provides a large selection of helpful object classes through its Application Programming Interface (API) that streamline the C&DH software development process.

Upon bootup, the integrity checking software generates hashes for all three copies of the flight software executable. All three hashes are compared, and the copies with the matching hashes are assumed correct, and executed. A copy with a mismatching hash from the other copies is considered incorrect. If the incorrect copy is the primary FSW, it is replaced by a backup. The worst-case scenario is that all three copies of the software do not match, as pictured in the bottom right state of the diagram. If this is the case, the primary copy of the FSW will be executed. Upon a successful startup of Hookem, an error message will be recorded, and this problem will have to be further analyzed on ground.

### 5.3.9 Command File Validation

A command file validator is integrated into the ground station to satellite communication process in order to ensure the authenticity of uplinked files. All files that are uplinked to the satellite are authenticated on-board before the FSW makes any

attempt to interpret the contents. Therefore, the *CheckUplink* Activity thread will not attempt to open or process a file unless the command file validator checks to make sure it has come from an acceptable source, either from the UT-Austin ground station, or one of their partnering organizations.

The command file validation process, similar to the FSW integrity checker, is performed by generating hash strings using a cryptographic hash function. Upon creation of a command file or mission script by the ground station, the hash string is created and appended to the file. The string is generated from a secret key that is hard-coded into the FSW and the ground station software. When the satellite receives the uplinked file, it will then run the same hash function with the same key as on ground to generate the hash string. The string generated on-board is compared to that appended to the end of the uplinked file. If both strings are identical, the flag will be set to indicate that there is a newly uplinked file that the satellite can interpret and process. If the strings do not match, then the file is ignored and deleted, and an indication of receiving a non-authorized file is logged.

### 5.3.10 Watchdog

The LPC3250 includes a processor independent watchdog with disable, normal, and extended modes. The watchdog provides the ability to recover the satellite in the case of a processor lockup by resetting the computer. The watchdog program is responsible for kicking the watchdog to prevent a reset. In other words, the program will periodically reset the watchdog timer by changing the input state at a regular interval faster than the timeout period.

### 5.3.11 Error database

The satellite is expected to experience errors during on-orbit operations. During software development the C&DH team is responsible for maintaining the master document that lists all pre-defined errors that can occur in the FSW. These errors will be automatically logged on-board if they occur during flight. Examples of pre-defined errors

include the spacecraft not being able to open a file for reading or writing, or the spacecraft receiving a parameter associated with a command that is out of bounds. Each subsystem lead is responsible for listing any FSW errors pertaining to their system in the master document, and ensuring that the error will be logged in subsystem code. Included in the master error database document for each pre-defined error is a unique ID #, the name of the program files where this error could be created and the position of occurrence, the type of error (for example a null file pointer or an I2C write error), and the actions the satellite should take, if any, to resolve the error. The responsible party for inputting the error into the document can also include information on how the ground station operator would resolve the error. Some critical errors may severely affect the satellite's ability to complete the mission requirements, and may require a reset of the satellite in order to be resolved, which is indicated in the document as well.

Two on-board error databases were created to keep track of any such errors that the satellite experiences throughout the mission. The databases are implemented in the FSW using the SQLite 3 software library. The first database placed on-board will be used for logging errors that are referenced in a file containing all the possible error IDs, and any subsequent autonomous actions that the satellite should take. The second error database is the log that is populated as errors occur. This second log can be requested by the ground station to see if any FSW errors have occurred so that appropriate ground actions for resolution can be taken. Previous log entries can be deleted from the spacecraft storage when the information is received on ground.

# Chapter 6: C&DH System Testing

In order to verify the functionality of the C&DH system, it must pass extensive testing before it can be used for TSL's upcoming satellite missions. As much effort and focus should be put into the testing process of the FSW as is put into the design and implementation phases. The TSL has implemented the practice of performing tests on the satellite software and hardware throughout the development process. Testing commenced early in the FSW development cycle on individual software classes for a specific hardware component, and has continued in the assembly and integration phases by running vibration and thermal tests and simulating Day-in-the-Life scenarios with the full satellite.

This chapter describes the previous and current tests that have been executed to validate the C&DH system, as well as the FSW for the integrated satellite. In addition, previous test flight opportunities and demonstrations that were used as milestones in the FSW development schedule are presented in the chapter.

## 6.1 Flight Software Testing

It is vital to the success of the mission that all software to be flown on the satellite is put through rigorous testing to reveal software bugs and memory leaks. Software testing is a process of running a program with the objective of finding errors (Myers 2004). In other words, testing a program will increase the confidence that it will function as intended by finding imperfections and cases which refute this idea. The acceptance of this definition then leads to the following goals for software testing performed in the TSL: repeatability, systematic testing, and documentation. The tester should be able to repeat the encountered software defect, and show it to other TSL members if need be. This is an important step in resolving the defect, so that once the fix is implemented, it is known how to attempt to repeat the defect for the purpose of ensuring that the resolution was indeed successful. Systematic testing is used in this context to describe the action of choosing particular test strategies and cases so that the tests cover the full range of the

program's behaviour and usage (The University of British Columbia 2003). All formal testing should be well documented so that the lab has a record of which tests have been executed on what components of the FSW, and the results of these tests.

Working on a satellite mission, the software developers must perform as much testing as possible within the strict time constraints of the schedule. Even with allotting a significant amount of time in the schedule for software testing, it is impractical for all the possible points of failure to be detected and corrected. The schedule allows for an execution of a limited number of test cases. Therefore, it is important to put thought into the test cases so that they cover a wide range of possible scenarios and events that could cause failure.

### 6.2 C&DH Software Unit Tests

The first type of testing that was performed on the C&DH software was unit testing. Unit testing involves testing each separate unit of a software program on its own to ensure that it meets its specification (The University of British Columbia 2003). For the C&DH software, written in C++, each class was tested as a separate unit. Black box and glass box testing are common test techniques employed for testing engineering software products, and these tests were performed on each class.

Black box testing is a technique where the focus is on the specifications and requirements of the software (Homes 2013). The test on a particular software unit is considered complete when the all the requirements have been verified. The name comes from the notion that the test cases are generated without viewing the actual code, thereby treating the unit as a black box, but only from considering the specifications. The goal of this technique is to verify that the software unit will interface correctly with the rest of the program by testing the unit with different inputs and observing the resulting outputs. One advantage of black box testing is that the tester need not be familiar with the implementation details of the software component but must rely on the functional specifications. Therefore, the tester will not make any assumptions about the

functionality of the software when running various test cases. Another advantage to this form of testing is that it requires that the specifications be well documented.

Glass box testing, as the name suggests, considers the internal structure of the code being tested, and identifies defects with the internal functionality of the code. This technique focuses more on the implementation of the unit and examines the logical paths through the software (Iskrenovic-Momcilovic and Micic 2007). The list of test cases should be path complete, meaning that all possible paths through the code have been tried with at least one test case. It may be impractical to test all paths through a software unit, such as with the case where a loop is run through N times, where N is a very large number. In these cases, the tester must use their judgment, such as in the aforementioned scenario of having test cases where the loop is run through 0, 1, 2, N-1, and N times. The tester should keep in mind that all possible paths to exit the loop should be tested.

Each class created for the C&DH software was put through unit testing, both black box and glass box tests. Once the implementation of a class was completed, a document was generated to record the results of both tests. Included in this document is the description of the class, the test procedure, and the list of methods to be tested. The test procedure section includes any necessary hardware and the set up instructions. For each method, the functional specifications are listed in the document for reference. For both the black box and glass box tests, the documentation includes each test case performed, the expected and actual results, and a pass/fail check box. Also included in the document is a section for analyzing any failed cases and any major changes to the code needed to resolve the issue. An example of the information given for one method, *DownlinkFile()* from the *CommWithGroundActivity* class, is given below.

**int DownlinkFile()**

Downlinks the first file in the FRL that hasn't been downlinked yet.

**Function Specifications**

//MODIFIES: the FRL file (changes the 0 to a 1 beside the file name in the FRL if that file is downlinked)
//EFFECTS: downlinks the next file in the frl that hasn't been sent to ground yet and returns 1 if successful. returns 0 otherwise.

**Test Details**

| TEST CASE | EXPECTED RESULT | ACTUAL RESULT | PASS / FAIL |
|---|---|---|---|
| Function call when no files are in the FRL | Does nothing to FRL; return 1 | 1; Does nothing to FRL | P |
| Function call with all files downlinked already | Do nothing to FRL; return 1 | 1; FRL doesn't get modified | P |
| Function call with all files yet to be downlinked | Change the first file indicator to 1; return 1 | 1; changes the first file indicator in FRL to 1 | P |
| Function call with no FRL existing | Creates an error; returns 0 | Error message; returns 0 | P |

Figure 37. Information Included for the *DownlinkFile* Method of the *CommWithGroundActivity* in its Unit Testing Documentation

For the majority of the classes, the correct functionality was confirmed through the performance of these tests. In some cases, when coming up with the test cases, the C&DH team could recognize an error in the method before running the tests. In the case of the black box testing, if the tester was the person who wrote the method, they would recognize that a particular scenario or a set of inputs was not considered during implementation, and that the method would not respond correctly. Therefore, the tester could then update the code in order to handle this case correctly, and the test case would be considered as passed. It might have been beneficial to ensure that the tester was not the same person as the developer in order to get a better idea on how many of the test cases would have failed without implementing fixes concurrently. However, the goal of the tests is to identify failure points for each method, and this was accomplished.

The performance of this type of testing on the C&DH software provided valuable information pertaining to its development. First of all, the generation of the test cases forces the developers to put thought into as many possible cases where each method

could fail and produce undesired outputs. Most of these cases would not have been considered if not for these tests. In particular for the C&DH software, due to the FSW's centralized architecture, there are many classes and methods whose purpose is to interpret files and instructions sent from other subsystems or the ground station, and to relay this information to another component of the satellite. Therefore, there are many test cases that involved characterizing how the software could handle scenarios that are special, but not rare, such as improperly formatted commands, or empty or missing files. Performing these tests helped identify and prepare for these cases.

Another useful outcome from these tests is that they provided the software developers with an opportunity to put more thought into the purpose and functional requirement for each class involved in the C&DH software. Since these unit tests emphasize the functional specifications, a consequence was that each class' role and how it fit into the C&DH software design was analyzed in scrutiny.

### 6.3 KESLER INTERFACE BOARD TESTS

In order to validate the Kesler interface board for flight, a test plan was generated and executed for each board used in the TSL. The test plan outlined several testing procedures to be completed for each new interface board used for either software development or for flight. The test procedures to be performed for testing the Kesler board include checking for short circuits, current draws of the voltage regulators, header connectivity, proper power switch functionality, watchdog capabilities, and communication interface functionality. Similar to all hardware test procedures that are executed in the TSL, the Kesler board hardware test is administered by the technician who enters all of the results into the test plan with his/her initials. Once completed, the test plan is examined by a team member acting as the Quality Assurance agent. The tests were completed on the Kesler boards once they were acquired from the manufacturers. The most recent Kesler v2 boards for the RACE mission were tested this summer with no anomalous results.

## 6.4 GROUND STATION GRAPHICAL USER INTERFACE

To aid in the testing phase of the development process for the TSL, a ground station Graphical User Interface (GS GUI) is currently being developed by the C&DH team to act as the ground station's interface with the satellite during integrated system testing, as well as flight operations. Work on the GS GUI commenced in 2012, and new functionality has continued to be added to the software over time. Versions of the GS GUI have been used for past satellite software and hardware demonstrations, as well as a tool for performing FSW testing including the Command Execution Tests (CET) and Day-in-the-Life tests.

### 6.4.1 GUI Features

The objective of the GUI is to minimize the effort required for the ground station operator to interact with the satellite during the testing phase and for flight. The GUI meets this objective by allowing the user to generate mission scripts and command files by inputting their selections through buttons or drop-down lists. Screen shots of the current versions of the two windows comprising the GS GUI, the command window and the telemetry window, are shown in Figure 38 and Figure 39.

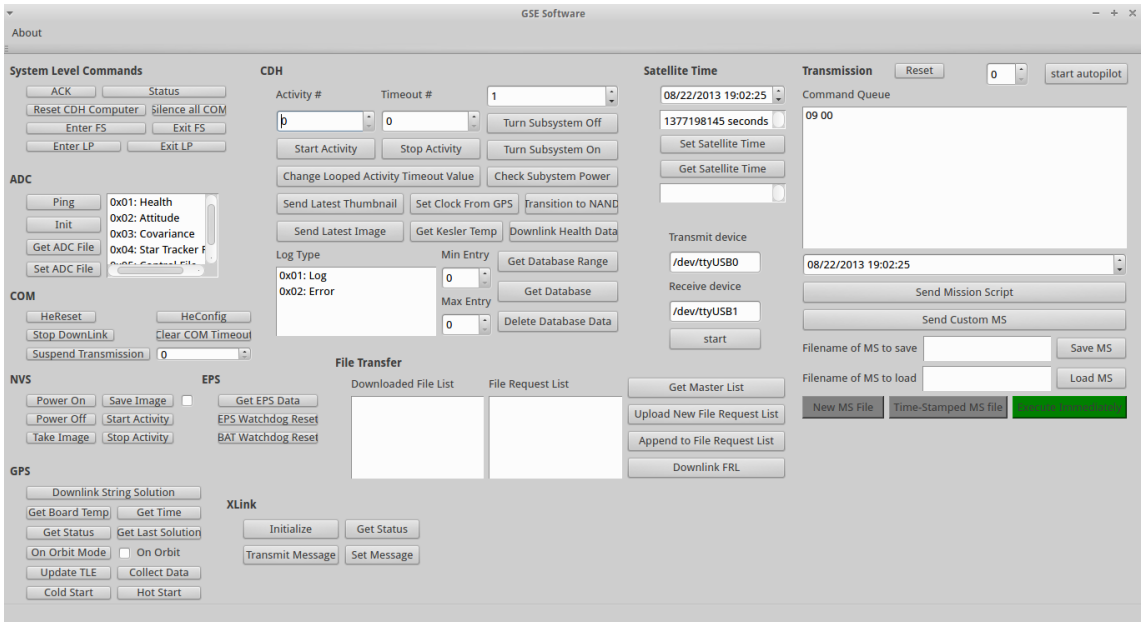


Figure 38. The Command Window of the GS GUI

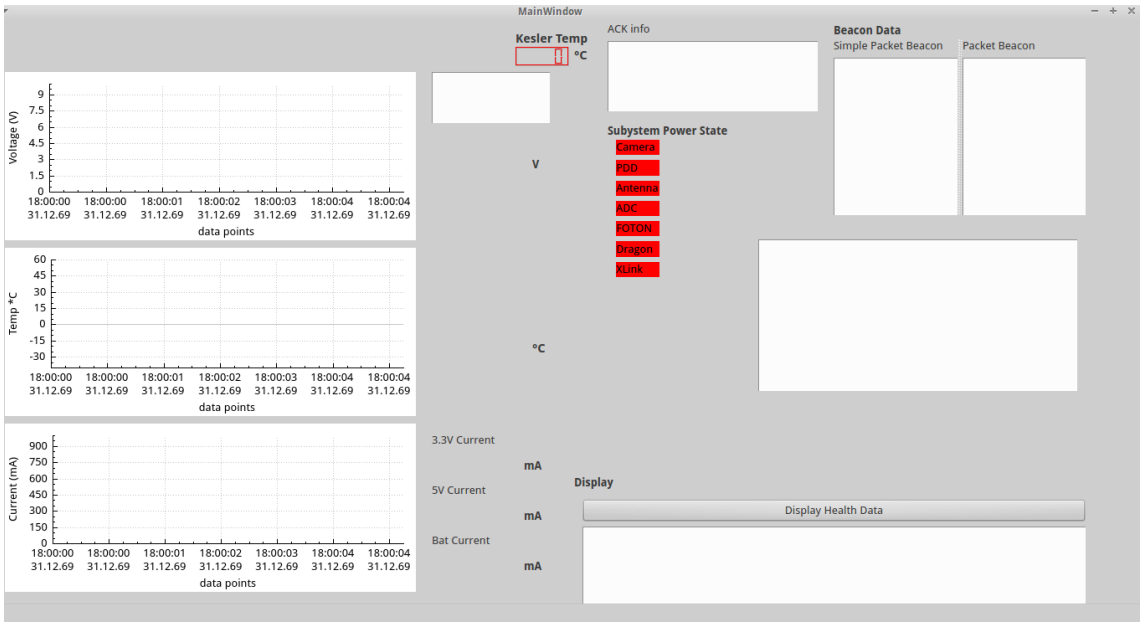


Figure 39. The Telemetry Window of the GS GUI

The command window is where the user can create a mission script or a command file, and select the commands that will populate the file. The file to be uplinked is generated automatically by the ground station software based on the user's selections. The command queue box located on the right side of the window shows the data that will be written to the command file based on the user's command selection. This window acts as an aid for checking for the correct op-codes to be sent to the satellite. The data is shown before slip encapsulation and before the hash string for command file validation is appended.

Upon transmission, the GUI generates two files to be saved on the host computer. The first file generated contains the raw data that is transmitted to the satellite in the exact format that is uplinked. The second file is a human readable file which contains the names of the commands that are included in the uplink file. This file can be quickly scanned by the user to view which commands were included in a past command file or mission script. Allowing the GS GUI to populate the file with the op-codes and restricting the user to the buttons and menus on the command window reduces the risk of transmitting an incorrect list of commands to the satellite as the formatting of the command file or mission script is performed by the GUI.

The telemetry window displays some of the data that is downlinked by the satellite. The GUI is used for receiving satellite beacons, health, telemetry, and mission data downlinked in response to a sent command. The data received from the satellite is saved onto the ground station computer and timestamped with the ground time. The telemetry window then parses the data received from the satellite, and displays either certain fields of the parsed downlinked data or a generated message indicating that mission data has been received and saved on ground. Data that is currently presented in the window includes health data such as battery voltage, current, Kesler board temperature, satellite acknowledgement messages, the on/off status of subsystems, and requested downlinked camera images displayed in a pop-up window.

Another useful feature of the command file generation capability of the ground station software is the capability to communicate with the flat sat either through radio transmission or through a UART interface. The flat sat comprises similar hardware and electrical connections as those included in the flight version of the satellite, but it is laid out flat on a surface for easier accessibility while testing. All formal tests are first conducted on the flat sat before they are attempted on the flight spacecraft. The current configuration of the TSL's flat sat, consisting of the Bevo-2 versions of hardware, is shown in Figure 40.

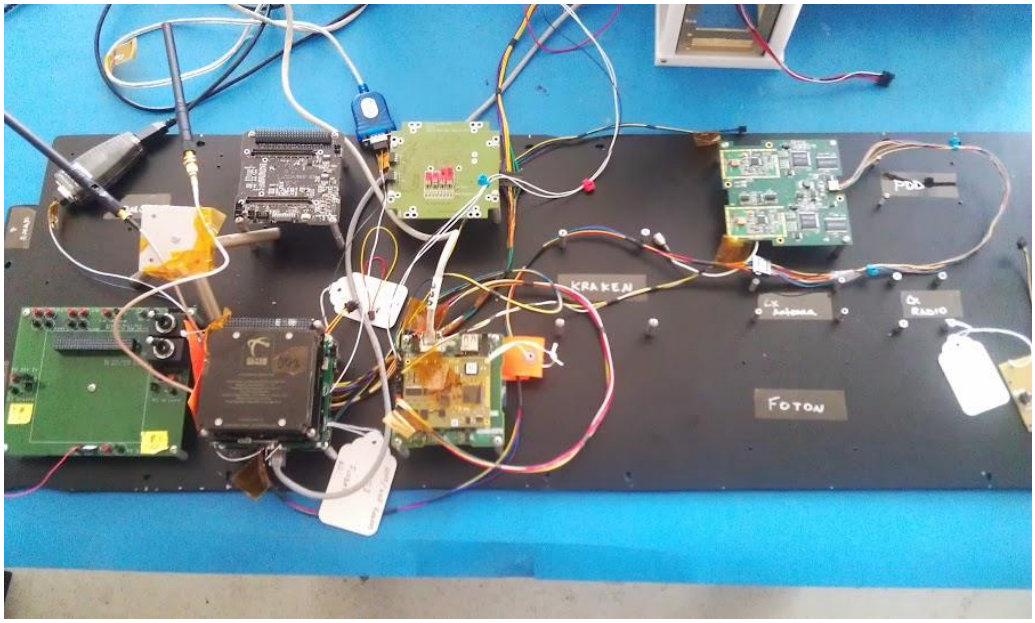


Figure 40. Flat Sat used for C&DH and Full FSW Testing

The GUI allows the user to specify the method of communication. Only one variable needs to be changed in the FSW in order to switch between selections. This feature has been useful during software testing with the flat sat in the times where radio transmission is not possible. This could be due to the radio software being in an inoperable state, or not having access to the communication team's hardware to be used on the flat sat. As this feature was not thought of until after later in the development of the C&DH system, the UART connection to the flat sat has been through the interface designated for the PDD

instrument used on ARMADILLO. Therefore, when flat sat testing involving the PDD begins for this mission, connecting the ground station GUI through UART instead of using radio transmission will not be possible. This capability will be examined for addition to future C&DH system designs.

### 6.4.2 Current Progress

The future work of the C&DH team on the GS GUI will consist mainly of completing the implementation of the telemetry window capabilities. This includes finalizing the actions and the method of display of the GS GUI in response to downlinked data from each satellite command sent through the command window. For mission data, a capability that needs to be completed is an indicator to the user that there is more recent downlinked data. Another modification to occur is to ensure that the software is operable, or can be operable with minimal changes, with Ground Station Equipment (GSE) other than what is currently available in the UT-Austin ground station. The purpose of this feature is two-fold: the GSE GUI is meant to be portable as it is to be used for satellite testing no matter the location, and the UT-Austin ground station equipment is soon to be upgraded before launch of the current TSL satellites. These additions, along with other minor modifications, will help make the GSE GUI a very valuable testing tool for the current missions, and a strong stepping stone for testing tools for future TSL missions.

## 6.5 GSE HARDWARE

The GSE hardware consists of the Ground Support (GS) laptop, the GSE interface box, and the umbilical wire harness that connects to the satellite from the GSE box. The GS laptop runs the GSE GUI and other scripts used in the integration and testing phase. If a direct connection to the satellite is required (and not through interaction with the GS GUI) during testing, the GSE interface box acts as the medium between the GS laptop and the satellite. The GSE box was designed by members of the TSL for use with the current missions. Its functionality includes providing inhibit switches, displaying the raw battery voltage through the use of a voltmeter, and enabling charging of the EPS system

by providing 5V DC power. Figure 41 shows the connections between the GSE equipment and the satellite.

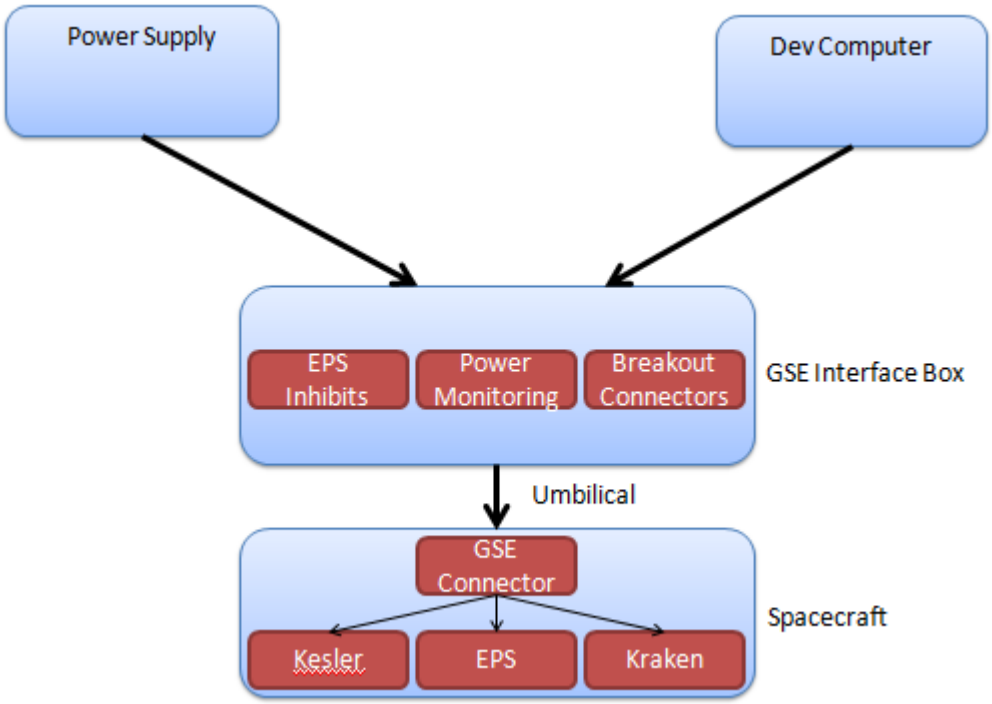


Figure 41. Hardware Block Diagram of GSE Setup (Texas Spacecraft Laboratory 2012)

The block labeled *DevComputer* represents the GS laptop for a direct connection, where the user can see the console output of the LPC flight computer and type commands via command line directly to the LPC on the spacecraft. The GSE box also provides a connection to the Kraken interface board so that a second development computer can be used to directly connect to the ADC computer.

### 6.6 Past Flight Software Demonstrations and Testing Opportunities

There have been several opportunities over the past year to demonstrate the capabilities of the flight software and its use with the Engineering Development Unit (EDU) hardware. These demonstrations acted as major milestones in the overall satellite development schedule, and helped advance the progression of the FSW. For each

demonstration, the TSL wanted the FSW to meet a certain level of functionality. Therefore in preparation for each event, a significant collective effort was made by the TSL members to meet these goals. Thus, major improvements in the capabilities and testing of the FSW were achieved. Past demonstrations were held at the ARMADILLO Proto-Qualification Review (PQR) in August 2012, at the Small Satellite Conference in Logan, Utah, and at the Flight Competition Review (FCR) in January 2013. Another major milestone in the FSW development was in preparation for the Student Hands-On Training (SHOT) II workshop in Summer 2012.

In late June 2012, four members of the TSL participated in SHOT II in Boulder, Colorado. SHOT II is a three-day workshop hosted by the Colorado Space Grant Consortium for teams from the universities entered in the current UNP Nanosat Competition. Occurring after the first year in the two year lifetime of the competition, the workshop gives each team the opportunity to test a component or multiple components of their satellite in a flight-like environment by launching it on a high altitude balloon.

The TSL's main objective from this testing opportunity was the verification of the C&DH system's interface with other subsystems. The payload contained similar or identical hardware as is expected to fly on the ARMADILLO satellite. Therefore, the mission was designed to verify that the SW would successfully boot and initialize, turn on the various subsystems, and record and save data to the SD card for downlink during flight and for post-flight analysis. UT-Austin's payload consisted of the C&DH system, including the LPC3250 connected to a Kesler v.0 interface board, the Lithium 1 radio, a Honeywell HMR2300 magnetometer which is a component of the ADC system, the ClydeSpace EPS system, the NVS system, and the FOTON GPS receiver. The Lithium 1 radio is another UHF/VHF radio fabricated by AstroDev which is the half-duplex version of the Helium radio to be used on TSL's current satellite missions. Another hardware component flown on the SHOT II payload that is not a part of ARMADILLO is a Honeywell pressure/temperature sensor to monitor the temperature and to relate this to the data generated from the FOTON, magnetometer or the EPS system's voltage and current readings.

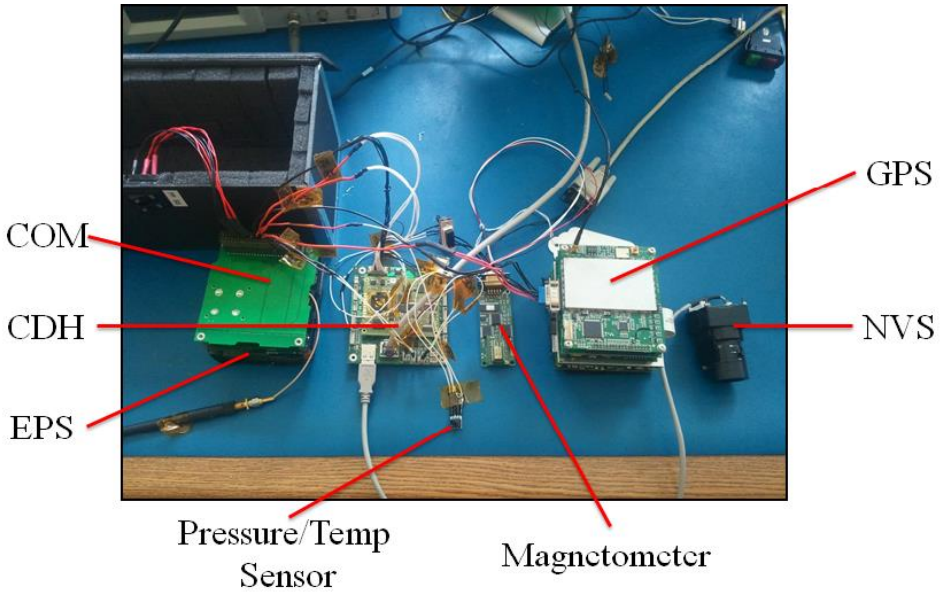


Figure 42. Components of the SHOT II Payload for the TSL

The flight experiment consisted of recording magnetometer and pressure measurements at a rate of 10 Hz, capturing and saving images with the NVS camera every 30 seconds, beaconing a simple text message every minute, and collecting GPS readings from the FOTON.

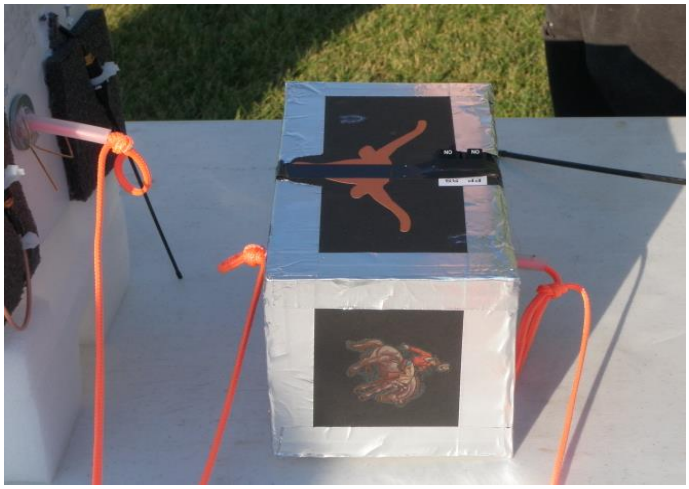

Figure 43. SHOT II Integrated Payload for TSL Attached to the Other Payloads before Launch

The results from flight showed partial mission success. The FSW was initialized correctly upon launch, and log files containing the generated data began being produced 90 seconds after bootup. Log files were successfully produced and saved onto the SD card for all subsystems. Magnetometer and pressure data was successfully generated and stored on board. However, the files created by the FOTON did not contain any data, indicating that the instrument was not able to track any GPS satellites. From post-flight analysis, it was determined that there was no visible damage or bad connections to the antennas, and that all the hardware was still functional. The conclusion by the TSL SHOT team was that the FOTON board, along with the other subsystems of the payload, was attempting to pull an excess amount of current which caused the EPS to be current limited, and thus to not generate enough power for the FOTON to operate successfully. For the SHOT flight, the FOTON was attached to an interface board that needed to communicate with Kesler v.0 board. This extra interface board will not be flown on ARMADILLO. Therefore, the power budgets generated for ARMADILLO, and not updated for the SHOT II mission, indicated that there was sufficient power for the payload. However, these budgets should not have been applied for this experiment.

Another failure from this mission was that the camera images produced during flight were completely black, with the conclusion being that the settings were not adjusted accordingly in order to take pictures of the Earth.

Participating in the SHOT II workshop supplied the TSL with numerous lessons that will all aid in the continuing development of current and future satellite missions. It also demonstrated the importance of proper testing of all software and hardware components not only as a separate unit, but together as an integrated spacecraft.

### 6.7 FUNCTIONAL TESTS

The first tests to be run on the fully integrated satellite were the functional tests. These tests however, have an emphasis on verifying the satellite's hardware rather than the FSW. The purpose of the functional tests is to ensure that every hardware component of the satellite, with the exception of the ISIS deployable antennas, is operable and ready for flight. These tests are designed so that they can be executed before launch to ensure no component is in a state of failure at this time. Each subsystem was asked to create a functional software test script that will run through the commands necessary to verify that its hardware is operating correctly. These scripts were then compiled into an executable that allows the user to select which subsystem script to commence for testing. The executable is run from the satellite's flight computer, thus the GS GUI is not used for these tests. The testing procedures outlined for the functional tests dictate the order in which the subsystem scripts should be executed. Figure 44 shows the performance of the functional tests on Bevo-2. As shown in the figure, the user is directly connected to the satellite from the GSE laptop via the GSE interface box (located in the bottom of the figure).

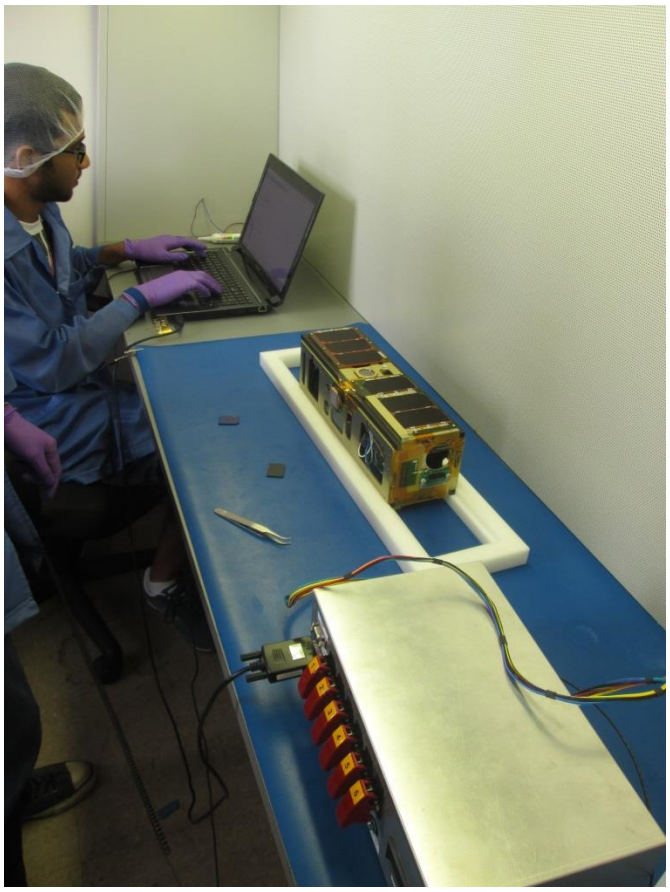

Figure 44. Functional Test in Progress on Flight Version of Bevo-2 Satellite

The functional tests were run first on the flat sat to observe the satellite actions before running the tests on the Bevo-2 flight satellite. Functional testing for Bevo-2 commenced earlier in June 2013, and has been executed for all subsystems twice on the flat sat, and once on the flight unit. Several of the individual subsystems tests did not pass when the tests were conducted on the flat sat, namely the scripts for the NVS, GPS and COM subsystems. However, it was determined that these failures were due to minor bugs in the functional test scripts themselves. For the functional tests on the flight unit, the only subsystem that failed was the ADC system. Troubleshooting to find the cause of this failure led to the discovery of a hardware defect in the EPS flight unit which is currently being resolved. Even though the mission development schedule experienced a setback

due to this failure, the importance of running the functional tests was demonstrated through this result.

### 6.8 Command Execution Tests

The next major step to be performed in the FSW testing process is the Command Execution Test (CET). The purpose of the Command Execution Test is to run through every command that can be uplinked to the spacecraft (Air Force Research Laboratory 2013). This is to help prevent the ground station from sending a command that can place the satellite into an unknown state and jeopardize its integrity. The results generated in this test can then be used to compare and recognize the satellite actions that are taken during flight.

The test is designed to be executed with the FSW running on the integrated satellite and with the hardware reacting to the uplinked commands. Therefore, the CET will not only test for any bugs or defects in the FSW, but it will test the various software and electrical interactions between all the components of the integrated satellite. The CET differs from functional tests run on the full satellite in that the functional tests only execute one script at a time, and each script is written specifically to test only the functionality of one subsystem. The CET is the first test involving both hardware and software from multiple satellite subsystems. Similarly to the functional tests, the CET is executed first on the TSL's flat sat as a dry run, and every command the spacecraft can interpret must pass before the CET is attempted on the flight unit.

The CET will be the first test with the fully integrated satellite that involves utilizing the end-to-end operation of the TSL's communication system. All commands sent to the satellite will be transmitted using UT-Austin's GSE and received using the satellite's Helium radio. Therefore, the CET has added importance in that it will also test the integration of the communication system's software with the C&DH software. The interface between the COM and the C&DH software is critical, as a failure in receiving or transmitting data for the satellite will lead directly to mission failure as no data will be collected. The test is designed to involve commanding the satellite in as similar a process

to flight-like conditions as possible, thereby maximizing the detection of possible errors and defects that can be encountered while in orbit. To date, most of the preparation time of the FSW for the CET has been on integrating the COM code with the FSW.

### 6.8.1 Test Description

The CET is performed to verify the satellite's behaviour for both cases of the spacecraft receiving a command in the proper format with the correct parameters, and of receiving the command with incorrect parameters. If a command is accompanied by incorrect parameters, an error message should be generated and pushed to the error database, and the command should not be executed. It is also important to try the same command but when the satellite is in different states. For example for Bevo-2, a command exists to take a picture using the star tracker camera. This command should be sent as part of the CET for the cases of when the camera is on and the satellite is in ACE mode, when the camera is off and the satellite is in ACE mode, and when the camera is off and the satellite is in Fail Safe mode. All three cases should result in the satellite taking different actions. A picture is only taken for the first case, whereas an error is generated and pushed to the error database for the other two cases. In the second case, the satellite should push an error indicating that the camera is currently off. Whereas in the third case, the error will indicate that this command cannot be executed while in Fail Safe mode.

### 6.8.2 Test Procedure

The C&DH team has recorded in the document listing the op-codes for each command the following information to be used during the test: a brief description of the command, the parameters needed to be accompanied with the command and their accepted range if any, the expected actions taken by the satellite, the data to be downlinked, the possible error IDs that can be generated, and the high level subsystem functions that are called to accomplish the command. The expected actions taken by the satellite include any data generated and stored on-board, and any change of the satellite's

software or hardware state. While running the CET, the tester is expected to document the observed actions of the satellite in response to a command along with any data downlinked, and compare them to the expected results in order to validate the command.

# Chapter 7: Future Work and Conclusion

This thesis presents the work that has been performed to date in the development of the C&DH system to be used on the Bevo-2, ARMADILLO and RACE missions. With the delivery dates for Bevo-2 and RACE quickly approaching in the spring of 2014, there will be no decrease in effort in order to finish the FSW implementation and testing to prepare the system for flight operation. At the time of publication of this thesis, the focus of the C&DH team members of the TSL is on preparing for the CET, and finishing the software necessary to run scenario and Day-in-the life testing.

## 7.1 Scenario and Day-in-the-Life Testing

Upon completion of the CET, the next tests to perform on the integrated satellite are scenario tests and Day-in-the-Life tests (DITL). These tests are designed to simulate in-flight events and activities the satellite will experience. Similar to the CET, these tests will only pass data in and out of the FSW by means that will be used during flight operations. The purpose of these tests is to verify the functionality of the fully integrated satellite while it is performing various sequences of flight operations.

In terms of the C&DH FSW, examples of particular scenarios that should be tested are the proper interaction of downlinking data based on the on-board pass prediction and based on ground request, the uplink of a new mission script or command file to execute when the current mission is partially completed, and the generation and storage of mission data when reaching SD card storage limit.

The DITL tests are the last type of tests to be run on the satellite before it is delivered for launch vehicle integration. Each mission phase for the particular satellite under test must be fully executed and in the correct order. As the lifetime of the current TSL satellites range from 6 months to 2 years, the DITL tests will be an accelerated simulation of the mission. The boot-up sequence and startup procedure that will occur once the EPS system's inhibits are disabled are included. The test should also force the satellite to enter and operate in all of its FSW modes, including Low Power mode and

Fail Safe mode. Therefore, the test should simulate situations where the satellite does not have enough power to accomplish the commands included in the current mission script or command file, or where the satellite experiences a critical failure when attempting to execute a command.

Every possible scenario that may be encountered while in orbit cannot be simulated in ground testing. Therefore, the comprehensive suite of tests that are performed (functional tests, command execution tests, and day in the life tests) are designed to encompass a wide range of possible events that may occur. Judicious selection of test cases allows proper behavior of the FSW to be demonstrated in multiple situations. The goal of the comprehensive satellite testing is to demonstrate correct behavior of the integrated satellite to the extent possible, and to detect and identify any software bugs or anomalies that occur so they may be documented and corrected prior to flight.

## 7.2 CONCLUSION

Since their beginnings in the late 1990's, CubeSats have been the satellite form factor of choice for an increasing number of scientists and researchers in both the educational and professional industries due to their low cost and advancing performance. As the CubeSat community continues to grow, so must the technologies and capabilities that can be flown on these types of satellites, including the C&DH system. The TSL at UT-Austin has experienced this expansion in the quantity of CubeSat missions the lab is currently involved in and in the increasing operational sophistication of these missions. The presented C&DH system was developed to meet the multiple requirements and capabilities of the current and future missions of the UT-Austin TSL. Goals that were considered in the design of the C&DH subsystem are modularity and reusability. Designing the system around a COTS system on module as the flight computer running a Linux environment, and implementing the FSW in C++ using O-O techniques, allows for a software architecture using a component-based architectural style. Structuring the FSW in a modular manner where each subsystem is treated as a component that interacts with

the central flight computer provides system advantages such as subsystem upgradability and interchangeability. As with any new system, thorough and well-planned testing is an integral step in the development process, requiring just as much effort as the implementation. The tests included in validating the C&DH system, such as the Command Execution test and the Day-in-the-Life test, are described in this research. Leading up to satellite delivery of the Bevo-2 and RACE spacecraft in 2014, the emphasis for the C&DH team will be on completing this testing in order to validate the FSW before flight unit delivery.

# Appendix

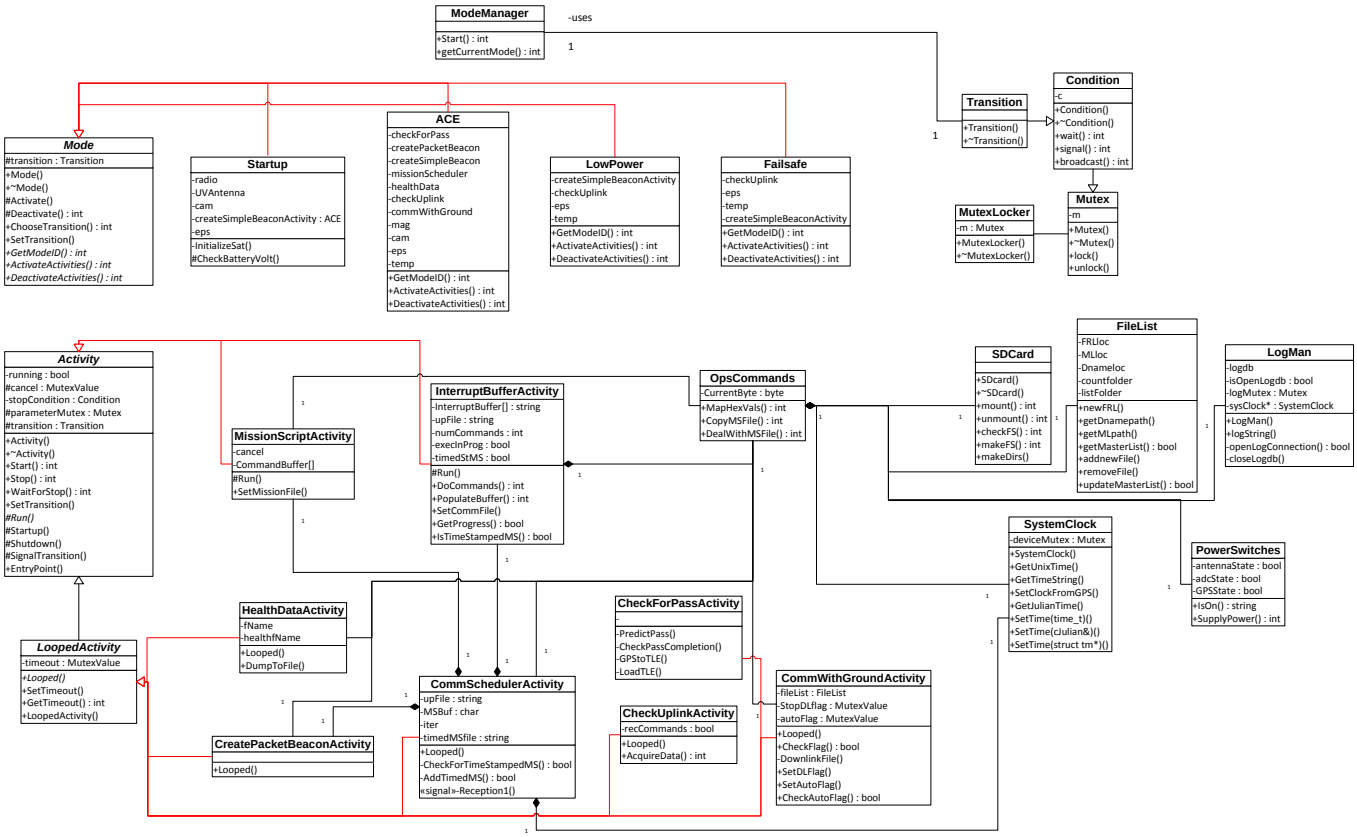


Figure 45. Class Diagram of C&DH FSW

# REFERENCES


AggieSat Lab. *AggieSat Lab at Texas A&M University - "Design, Build and Fly".* 2010. http://aggiesatweb.tamu.edu/index.php/projects/lab_projects/aggisat_2 (accessed 03 15, 2013).

Air Force Research Laboratory. "Nanosat-8 User's Guide." User's Guide, 2013.

Barney, Blaise. *POSX Threads Programming.* 01 11, 2013. https://computing.llnl.gov/tutorials/pthreads/ (accessed 03 21, 2013).

Bhatti, Jahshan. "The University of Texas Experience with uClinux, the TCM-BF537, and C++." Presentation, Austin, 2008.

Blue Technix. "Hardware User Manual CM-BF537." Document, 2012.

Brumbaugh, Katharine. "The Metrics of Spacecraft Design Reusability and Cost Analysis as Applied to CubeSats." Masters Thesis, Austin, 2012.

Buttlar, Dick, Jacqueline Farrell, and Bradford Nichols. *PThreads Programming A POSIX Standard for Better Multiprocessing.* O'Reilly Media, 1996.

California Polytechnic State University. "CubeSat Design Specification." Specification, 2009.

Critical Link. 2013. http://www.mitydsp.com/system-on-module (accessed 01 20, 2013).

Department of Aerospace and Engineering Mechanics at the University of Texas at Austin. *Lightsey Research Group - "Bevo-2".* 2013. http://lightsey.ae.utexas.edu/research/bevo-2/ (accessed 01 06, 2013).

Eden, Amnon H., and Rick Kazman. "Architecture, Design, Implementation." *25th International Conference on Software Engineering.* Portland, 2003. 163-182.

Eeles, Peter. *What is a Software Architecture.* February 15, 2006. http://www.ibm.com/developerworks/rational/library/feb06/eeles/ (accessed 03 01, 2013).

Fielding, Roy Thomas. "Architectural Styles and the Design of Network-based Software." Doctoral Dissertation, Irvine, 2000.

Fortune, Jared. "Estimating Systems Engineering Reuse with the Constructive Systems Engineering Cost Model (COSYSMO 2.0)." PhD Dissertation, 2009.

Foust, Jeff. "Emerging Opportunities for Low-Cost Small Satellites in Civil and Commercial Space." *24th Annual AIAA/USU Confernece oon Small Satellites.* Logan, 2010.

Franchitti, Jean-Claude. "Application Servers G22.3033-011;Session 2 – Sub-Topic 2, Enterprise Architecture Frameworks (EAFs) & Pattern Driven EAFs." Presentation, New York, 2011.

Garlan, David, Robert Allen, and John Ockerbloom. "Architectural Mismatch: Why Reuse Is Still So Hard." *IEEE Software*, 2009: 66-69.

Gomaa, Hassan. *Software Modeling and Design: UML, Use Cases, Patterns and Software Architectures.* Campbridge University Press, 2011.

Greenbaum, Jamin Stevens. "Flight Unit Fabrication of a University NanoSatellite:." Masters Thesis, Austin, 2006.

Henry, Michael F. *The OrbitTools Libraries NORAD SGP4/SDP4 Implementations in C++ and C#.* 2013. http://www.zeptomoby.com/satellites/ (accessed 03 25, 2013).

Homes, Bernard. *Fundamentals of Software Testing.* Wiley, 2013.

IEEE Computer Society. "IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes." Standard, New York, 2004.

Imken, Travis. "Design and Development of a Modular and Reusable CubeSat Bus." Honours Undergraduate Thesis, Austin, 2011.

Iskrenovic-Momcilovic, Olivera, and Aca Micic. "Mechatronic Software Testing." *Telsiks.* 2007. 486-489.

Jansen, SLinger, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir. "Pragmatic and Opportunistic Reuse in Innovative Start-up Companies." *IEEE Software*, 2008: 42-49.

Jet Propulsion Laboratory. "CHARM Project Overview." Presentation, Austin, 2013. *Phaeton Early Career Hire Development Program - Radiometer Atmospheric CubeSat Experiment (RACE).* 2013. http://phaeton.jpl.nasa.gov/external/projects/race.cfm (accessed 07 2013).

Johl, Shaina, and Travis Imken. "Design of the Command and Data Handling System in the UT Austin Satellite Design Laboratory." Austin, 04 2012.

Khan, Arindam, Mukesh Sharma, G. Ganesh, S.D. Dhodapkar, B.B. Biswas, and R.K. Patil. "A Cryptographic Primitive Based Authentication Scheme for Run-Time Software of Embedded Systems." *2nd International Conference on Reliability, Safety and Hazard (ICRESH).* Mumbai, 2010. 500-504.

Kjellberg, Henri. "Design of a CubeSat Guidance, Navigation, and Control Module." Masters Thesis, Austin, 2011.

Lam, W. "Achieving Requirements Reuse: A Domain-specific Approach from Avionics." *Journal of Systems and Software*, 1997: 197-209.

Larson, Wiley J., and James R. Wertz. *Space Mission Analysis and Design.* Torrance, California: Microcosm Inc., 2006.

Lightsey, Glenn. "Getting There: An Update on UT-Austin's Texas Spacecraft Lab." Presentation, Austin, 2013.

Magliacane, John A. *Predict.* 2013. http://www.qsl.net/kd2bd/predict.html (accessed 03 25, 2013).

MEN Mikro Elektronik GmbH. *About Computer on Modules.* 2013. http://www.men.de/products/computer-on-modules,som,About-Computer-On-Modules.html (accessed 01 21, 2013).

Microsoft. *Chapter 2: Architectural Patterns and Styles.* 2009. http://msdn.microsoft.com/en-us/library/ee658125.aspx (accessed 03 01, 2013).

Munoz, Sebastian, Richard W. Hornbuckle, and Glenn E. Lightsey. "FASTRAC Early Flight Results." *Journal of Small satellites*, 2012: 49-61.

Myers, Glenford J. *The Art of Software Testing.* New Jersey: John Wiley & Sons, Inc., 2004.

National Aeronautics and Space Administration. *1.* May 21, 2013. http://www.nasa.gov/directorates/spacetech/small_spacecraft/smallsat_overview.html#.Uh0lBxtQE1Y (accessed 01 02, 2013).
NXP. "LPC3220/30/40/50 Product Data Sheet." October 20, 2011.
Oualline, Steve. *Practical C++ Programming.* Sebastopol: O'Reilly Media, 2003.
Parallab, Bergen Center for Computational Science. "Software Reusability and Efficiency." Sectoral Report, Bergen, 2004.
Phytec. *phyCORE®-LPC3250.* 2013. http://phytec.com/products/system-on-modules/phycore/lpc3250/ (accessed 01 22, 2013).
Phytec. "Quickstart Instructions phyCORE-LPC3250 Rapid Development Kit for Linux." Document, Austin, 2011.
Sinclair Interplanetary. "NSP Packet Protocol for Sinclair Interplanetary Hardware." Document, 2008.
Smith, Aaron. "The FASTRAC Satellites: Software Implementation and Testing." *22nd Annual USU/AIAA Small Satellite Conference.* Logan, 2008.
Texas Spacecraft Laboratory. "ARMADILLO Computer Trade Study." Internal Document, Austin, 2011.
Texas Spacecraft Laboratory. "Bevo-2 GSE Description and Operation." Internal Document, Austin, 2012.
Texas Spacecraft Laboratory. "Bevo-2 LONESTAR Program CDR Presentation." Presentation, Houston, 2011.
Texas Spacecraft Laboratory. "SYS193-Data_Budget." Internal Document, Austin, 2011.
Texas Spacecraft Laboratory. "SYS193-Telemetry_Budget." Internal Document, Austin, 2011.
The University of British Columbia. "EECE 310: Software Engineering Testing/Validation." Presentation, Vancouver, 2003.
Toorian, Armen, Ken Diaz, and Simon Lee. "The CubeSat Approach to Space Access." *IEEE Aerospace Conference.* Big Sky, 2008. 1-14.
Wang, Xiaoyun, and Yu Hongbo. *How to break MD5 and other hash functions.* Springer-Verlag, 2005.

## Vita

Shaina Johl is a graduate student currently pursuing her Ph.D. in Aerospace Engineering at UT-Austin with Dr. Glenn Lightsey as her advisor. She graduated with a Bachelor of Applied Science in Engineering Physics from the University of British Columbia in 2011. In August 2011, she entered the Graduate School at UT-Austin. She is currently acting as the C&DH subsystem lead in the TSL at UT-Austin. Shaina has had internships at companies around the world including Dolby Canada, Mettler Toledo, and MacDonald Dettwiler and Associates (MDA). At MDA, she worked in the Space Missions group as a member of a team of engineers acting as the prime contractor for Sapphire, a satellite provided to the Canadian Department of National Defence to track resident space objects.

Permanent email: shaina.johl@gmail.com
This thesis was typed by the author.