SOFTWARE REQUIREMENTS SPECIFICATION FOR LUNAR ICECUBE

A Thesis

Presented to

the Faculty of the College of Science

Morehead State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Michael R. Glaser-Garbrick

April 21, 2017

Accepted by the faculty of the College of Science, Morehead State University, in partial fulfillment of the requirements for the Master of Science degree.

_____
Jeffrey A. Kruth
Director of Thesis

Master's Committee: _____, Chair
Dr. Charles D. Conner

_____
Dr. Benjamin K. Malphrus

_____
Kevin Z. Brown

_____
Date

SOFTWARE REQUIREMENTS SPECIFICATION
FOR LUNAR ICECUBE

Michael R. Glaser-Garbrick
Morehead State University, 2017

Director of Thesis: _____
Jeffrey A. Kruth

Lunar IceCube is a 6U satellite that will orbit the moon to measure water volatiles as a function of position, altitude, and time, and measure in its various phases. Lunar IceCube, is a collaboration between Morehead State University, Vermont Technical University, Busek, and NASA. The Software Requirements Specification will serve as contract between the overall team and the developers of the flight software. It will provide a system's overview of the software that will be developed for Lunar IceCube, in that it will detail all of the interconnects and protocols for each subsystem's that Lunar IceCube will utilize. The flight software will be written in SPARK to the fullest extent, due to SPARK's unique ability to make software free of any errors. The LIC flight software does make use of a general purpose, reusable application framework called CubedOS. This framework imposes some structuring requirements on the architecture and design of the flight software, but it does not impose any high level requirements. It will also

detail the tools that we will be using for Lunar IceCube, such as why we will be utilizing VxWorks.

Accepted by: _____, Chair
Dr. Charles D. Conner

_____
Dr. Benjamin K. Malphrus

_____
Kevin Z. Brown

To mom, dad, Danny, and Shayna.

My love for you has no bounds.

Contents

1 Introduction

This document describes the requirements for the Lunar IceCube (LIC) flight software. Lunar IceCube is a joint project between Morehead State University (MSU), Vermont Technical College (VTC), Busek Inc., and NASA. The project entails the construction of a 6U CubeSat that will orbit the moon and gather information about volatiles there, such as water in its various phases. This software requirements specification focuses on only the flight software for the LIC mission. This is the portion of the project for which Morehead State University is responsible for. We do not consider here the software requirements of the various subsystems on the spacecraft, since those subsystems are produced by agents. In some cases the subsystems are off-the-shelf components produced by vendors who are not directly involved in the LIC project. Also we do not directly consider the requirements of the ground software or ground operations, except to the extent that they impact the flight software. This document describes, essentially, version 1.0 of the software. LIC is a specialized application with an effective user community of one user (the LIC team). However, it is possible that some components of this flight software will be reusable to other missions, although support for such reusability is not a specific requirement. The LIC flight software does make use of a general purpose, reusable application framework called CubedOS. This framework imposes some structuring requirements on the architecture and design of the flight software, but it does not impose any high level requirements.

2 Best Practice

Since there is a collaboration between Vermont Technical College and Morehead State University to review Lunar IceCube's efficiency, reliability, and accuracy each member will be held responsible for their contribution of the project. Regular monthly meetings will be required to ensure progress is being made on the development of Lunar IceCube's software. The responsibilities of each developer are as follows:

- Develop the requirement specification and cost estimation for the project
- Develop the design plan and test plan for testing the tool
- Implement and test the application and deliver the application along with the necessary documentation
- Give a presentation to the team on completion of the analysis, design and testing phases.
- Planning, coordinating, testing and assessing all aspects of quality issues.

The responsibilities of the other team members are to:

- Review the work performed by the developer
- Provide feedback and advice

Quality must be defined and measured if improvement is to be achieved, however the term quality is ambiguous, and that it is commonly misunderstood. Quality is vastly misunderstood term because it is a part of the average person's vocabulary which bleeds into the

workplace, and it is a multidimensional idea. An idea is that quality isnt able to be measured. It can be discussed, but it can't be measured. According to Han "Terms such as good quality, bad quality, and quality of life exemplify how people talk about something vague, which they don't intend to define."(Martin, 2008) This view reflects the fact that people perceive and interpret quality in different ways. The implication is that quality cannot be controlled and managed, nor can it be quantified. This view is in vivid contrast to the professional view held in the discipline of quality engineering that quality can, and should be, operationally defined, measured, monitored, managed, and improved. However this is just one view of quality, or what can be called customer satisfaction. The other view is that the producer adheres to the software requirements to achieve quality (Martin, 2008). This allows us to define a metric and development model to determine the overall quality of software(Martin, 2008).

3 Requirements

3.1 Intended Audience

This document is intended to serve as a contract between the general LIC team and the flight software developers at VTC and Morehead State. Its audience consists, therefor, of three groups:

1. The engineers coordinating the construction of the spacecraft
2. The mission operations team and science team who have an interest in the behavior of the spacecraft
3. The faculty and students at VTC and Morehead State University both of which are writing the software for Lunar IceCube.

3.2 Scope

The scope of this document shall provide a system's level overview of the entirety of Lunar IceCube's flight software. This document shall not describe the functions or routine's for Lunar IceCube. That shall be described in a Software Definition Document.

3.3 Mission Outline

Lunar IceCube is a joint project between Morehead State University, Vermont Technical College (VTC), Busek Inc., and NASA. The project entails the construction of a 6U CubeSat that will orbit the moon and gather information about water volatiles and correlate it to a function of temperature, altitude and location.

3.4 Overall Description

The LIC flight software is responsible for coordinating the activity of the spacecraft to ensure the mission's scientific goals are realized. Specifically the software is responsible for the following items.

1. Post-deployment initialization activities such as: stabilizing the spacecraft, deploying and orienting the solar array, and establishing communications with Earth via the Deep Space Network (DSN).

2. Gathering, and potentially buffering, telemetry data from the spacecraft's subsystems (including image and other science data from the BIRCHES instrument), and transmitting that telemetry to the ground at an appropriate time.

3. Receiving scripts of timestamped commands from the ground and executing those commands at the specified time.

4. Detecting various fault conditions and responding to those faults in an appropriate way.

5. Accepting uploaded patches (or entire executables) from the ground and applying them to update the software as required.

The LIC flight software is not intended to autonomously control the spacecraft except during deployment and certain fault handling situations. Nominally the software is only intended to gather telemetry and playback commands transmitted from the ground. Additionally this software requirements specification does not cover any aspect of the ground software itself except to the extent that the flight software must exchange data with the ground. The

flight software will gather telemetry and send commands to the various subsystems of the spacecraft. From a flight software perspective the subsystems of interest are:

1. BIT-3 ion drive, more specifically its gimbal that will be used for pointing the drive relative to the spacecraft orientation, and the amount thrust that will be generated.

2. The pumpkin deployable solar array which will have gimbals for pointing the solar array toward the sun.

3. BIRCHES instrument which will be used for gathering the information about the water volatiles on the moon.

4. Iris radio which we will interface using Deep Space Network (DSN) protocols to communicate to the ground station.

5. Attitude determination and control system (ADACS) based on Blue Canyon's XACT. Note that XACT contains both a star tracker for determining attitude and the momentum wheels for controlling attitude.

The flight software shall use the VxWorks operating system, version 6.8. It shall run on SpaceMicro's Proton 400k flight computer using a dual core Freescale 2020 processor. The environment is assumed to be relatively harsh where subsystem failures are potentially common. Furthermore we assume that a subsystem that has failed may return to working order at an arbitrary time even without the flight software taking any recovery steps. System resets are assume to occur at any time and may be frequent. Certain elements of the system state shall be stored non-volatilely so they will persist across resets. Furthermore the integrity of non-volatile storage shall not be assumed. All items stored in non-volatile storage shall

have associated CRC checksums that are updated on writes and verified on reads. To the greatest extent possible, the flight software shall be written in SPARK 2014.

3.4.1 User Classes and Characteristics

The main users of this product are the mission ground controllers who will be interacting with the system via commands transmitted over the Deep Space Network. Indirect users of the software include the science team who will be analyzing the data gathered by the spacecraft.

3.4.2 Operating Environment

- **Environment.Platform** The flight software shall run on SpaceMicro's Proton 400k flight computer using a dual core Freescale 2020 processor.
- **Environment.HardFail** The environment is assumed to be relatively harsh where sub-system failures are potentially common. In particular, we assume that a subsystem may fail and then return to working order at an arbitrary time, even without the flight software taking any recovery steps.
- **Environment.AsyncReset** Because the Proton 400k flight computer is radiation hardened, it shall be assumed that asynchronous resets of the processor will not occur.
- **Environment.FileIntegrity** The VxWorks operating system detects failed I/O operations. In addition other integrity checks are in place elsewhere in the system. Thus the flight software shall not be required to add additional integrity checks on non-volatile storage, but it shall be required to check all file I/O operations for failure at run time.

- **Environment.StoredCRC** The CRC values on packet data from the subsystem shall be stored in non-volatile storage and downlinked so the ground system can verify the integrity of the data.

- **Environment.Reset** There shall be several restart modes: soft restart where most of the system state is carried over, hard restart where system state is (largely) reinitialized, and reset which simulates a cold reboot. The circumstances when these reset modes are used is TBD.

3.4.3 Design and Implementation Constraints

Tools.SPARK To the greatest extent possible, the flight software shall be written in SPARK 2014. Concurrency is permitted as allowed by SPARK

3.5 Interface Requirements

3.5.1 User Interfaces

The main user interface to the flight software is the testing/debugging interface. This interface is primarly used by software developers and testers.

3.5.2 Hardware Interfaces

Access to hardware shall be primarly via the VxWorks operating system and the Ada language runtime system (where applicable). See the subsytem ICDs for specifics about the low level software interface to the subsystems.

3.5.2.1 Proton 400K The Proton 400K will serve as the Communication and Data Handling system for Lunar IceCube. It will provide all the electrical interfaces for each individual subsystem.

3.5.2.2 IRIS Radio Transponder The IRIS radio is the primary communication method with the satellite. It provides the route for all information flow to the DSN ground network. The IRIS radio software interface is described in the ICD.

3.5.2.2.1 SPI Interface The IRIS Radio SPI driver shall be:

- It shall operate at 1 MHz clock
- It shall operate full duplex
- It shall be controlled by a chip select line

3.5.2.3 Lithium-1 The Lithium-1 is the secondary communication method with the satellite. In the event that we are not be able to talk to Lunar IceCube, it shall provide route for communication with the satellite. The Lithium-1 radio software interface is described in the ICD.

3.5.2.3.1 RS-422 Interface The Lithium-1 RS-422 driver shall be:

- It shall operate at 115200 bits per second
- It shall be 8 bits, no parity, 1 stop bit

3.5.2.4 BIRCHES

The BIRCHES infrared spectrometer shall be the payload of Lunar IceCube. It will measure water volatiles on the moon as a function of temperature, altitude, and location.

3.5.2.4.1 SPI Driver

The BIRCHES Spectrometer SPI driver shall be:

- It shall operate at 1 MHz clock
- It shall operate full duplex
- It shall use a chip select line

3.5.2.5 BIT-3

The BIT-3 shall serve as the propulsion unit on Lunar IceCube. It utilizes iodine crystals as its source of fuel.

3.5.2.5.1 RS-422 Driver

The BIT-3 RS-422 driver shall be:

- It shall operate at 115200 bits per second
- It shall be 8 bits, no parity, 1 stop bit

3.5.2.6 Electrical Power System

The Electrical Power System shall handle all power distribution. It shall interface with the solar panels and their gimbals, and all electrical power specification for the individual subsystems.

3.5.2.6.1 GPIO

To control the gimbal, we will be using GPIOs for controlling the motors.

3.5.2.6.2 I2C The EPS I2C driver shall be:

- It shall operate as a slave device
- It shall operate at 400 kHz
- It shall be identified by a 7 bit address

3.5.2.7 XACT The XACT shall act as the Attitude Determination and Control System for Lunar IceCube.

3.5.2.7.1 RS-422 Driver The XACT RS-422 driver shall be:

- It shall operate at 115200 bits per second
- It shall be 8 bits, no parity, 1 stop bit

3.5.3 Software Interfaces

Software interfaces are specified in this section. The following requirements apply:

- **Interface.OS** The flight software shall run on VxWorks version 6.8
- **Interface.CommandDictionary** The flight software shall support incoming commands specified in the spacecraft telemetry dictionary.
- **Interface.TelemetryDictionary** The flight software shall produce all messages specified in the spacecraft telemetry dictionary.

3.5.3.1 Delay Tolerant Networking

According to RFC 4838 Delay-tolerant and disruption-tolerant networks, and is an evolution of the architecture originally designed for the Interplanetary Internet, a communication system envisioned to provide Internet-like services across interplanetary distances in support of deep space exploration. DTN will be utilized by the IRIS radio, and use aspects from DTN known as Bundle Protocol (BP) and Licklider Transmission Protocol (LTP) as its main source of communication.

3.5.3.1.1 CCSDS File Delivery Protocol

The science data gathered by the spacecraft will be transferred to the ground over the Deep Space Network using the Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP). This is a general purpose file transfer protocol designed for use with very high latency, unreliable space links. A full CFDP implementation has many features that are not relevant to the Lunar IceCube mission. However, there is a partial implementation of CFDP, entirely in SPARK, to support our mission needs. We intend to make our implementation available to other groups as open source software in the hope that it may promote the use of SPARK in other space flight software systems. IceCube shall contain a CFDP implementation in SPARK compliant with CCSDS specification 727.0-B-4. CFDP facilitates file transfers of arbitrary size over a connection from two or more parties, directly or via proxies. The service is specifically intended for use over the DSN. Features implemented to date include:

- **CFDP.SPARK** The CFDP implementation shall be in SPARK.
- **CFDP.EncodingDecodingFull** support for CFDP header encoding/decoding.
- **CFDP.UnacknowledgedTransfer** facilitates unacknowledged file transfer between

two or more entities.

- **CFDP.AckNak**
- **CFDP.Ground**

3.6 System Features

In this section we provide requirements for the various subsystems in the spacecraft as outlined in Overall Description.

3.6.1 Deployment Activities

When Lunar IceCube is deployed from SLS there are a number of special activities that must be executed before normal mission operations can begin. Lunar IceCube is not intended to be highly autonomous. Most of its activity is directed from the ground in the form of sequences of commands sent to the command scheduler. However, deployment is different since no commands can be received until the spacecraft is able to communicate.

3.6.1.1 Functional Requirements The sequence of events during deployment, from a flight software perspective, are as follows (starting from when power is turned on):

Set RTC To Zero;	Initialize the real-time clock.
Delay 45 seconds;	To clear the SLS deployer.
Deploy Solar Array ;	Should already be done. This is a backup.
Initialize Subsystems ;	Put subsystems into appropriate initial states .
Turn On XACT;	Starts detumbling process automatically .
Wait For Battery;	Wait until "suffcient" charge is reached (or time out).
Point Antenna To Earth;	
Turn On Beacon;	

We now receive commands from the ground.

All future activity is driven by commands.

3.6.2 State Manager

The state manager maintains the state machine of the spacecraft. It plays the role of the "main" module. When the software starts it enters an initial state. Transitions between states, due to external inputs, or due to the passage of time, stimulate the spacecraft's activities.

3.6.2.1 Functional Requirements

- **States.Initial** Upon start-up the system shall consider that it might be coming out of various reset types.

3.6.3 Storage Manager

The storage manager is the interface to the non-volatile file system. This file system is used to store telemetry and science data.

3.6.3.1 Functional Requirements

- **Storage.Interface** The interface to the non-volatile file storage shall be the usual Ada library file handling subprograms.

3.6.4 Scheduler

The scheduler holds a database of commands that have been uplinked along with timestamps indicating when each command is to be executed. Here a "command" includes a unique identifier (so the command can be the subject of later editing or removal), information about which subsystem the command is for (the APID of the subsystem), the command itself, and any arguments to the command. The scheduler includes operations for inserting a command into its database and an internal timing loop that issues and removes commands when appropriate. These two components of the scheduler are called the validator and the issuer respectively.

3.6.4.1 Functional Requirements: Validator

- **Validator.Buffer** The command buffer shall contain space for at least 5000 commands.

- **Validator.Format**

- **Validator.Overflow**
- **Validator.Invalid**
- **Validator.Editor**
- **Validator.Telemetry**

3.6.4.2 Functional Requirements: Issuer

- **Issuer.Resolution** The issue time of a command shall be specifable to the nearest millisecond.
- **Issuer.Accuracy** The issue time of a command shall be accurate to the nearest millisecond
- **Issuer.Errors**
- **Issuer.Delete** After issuing a command successfully, that command shall be removed from the command queue.
- **Issuer.Telemetry**

Some commands put the spacecraft into a new state that changes the overall action of the system.

3.6.5 Telemetry Gatherer

In addition to executing commands sent from the ground, Lunar IceCube must also gather telemetry from the various subsystems and transmit it to the ground. In this context "telemetry" includes science data from the BIRCHES instrument.

3.6.5.1 Functional Requirements

- **Telemetry.Frequency**
- **Telemetry.Buffer**
- **Telemetry.Overflow**
- **Telemetry.Format**
- **Telemetry.Invalid**
- **Telemetry.Errors**
- **Telemetry.Telemetry**
- **Telemetry.Async** The action of the telemetry gatherer shall be asynchronous to that of the scheduler. Telemetry can be gathered from some subsystems while commands are being sent to other (or even the same) subsystem.
- **Telemetry.FileSystem** Telemetry (and science data) shall be stored in a non-volatile file system.
- **Telemetry.StorageTime**

3.6.6 Publish/Subscribe Server

CubedOS has built-in support for only point-to-point message passing between modules. However, some activities are more conveniently controled using a multicasting approach. Accordingly the publish/subscribe server supports multiple message "channels" to which modules can subscribe. When a message is published to a channel, that message is delivered

to all subscribed modules. This provides multicasting as well as decouples the sender and receiver of a message.

3.6.6.1 Functional Requirements

- **PubSub.Channels** The publish/subscribe server shall support at least 16 channels.

3.6.7 Tick Generator

The tick generator provides basic real-time clock services. However, due to the latency and overhead of message passing, modules that require high speed, real-time timing services may need to use internal tasks instead. The tick generator is suitable for slow speed or non-critical timing services.

3.6.7.1 Functional Requirements

- **Tick.RealTime** The tick generator shall use a real time clock as the basis of its timing. Such a clock is monotonic and tracks time independently of any software activity.

- **Tick.Periodic** The tick generator shall provide a service whereby a module can request the delivery of periodic tick messages with a period ranging from 1 ms to 1 hour (3,600,000 ms). The total set of tick messages generated in response to such a request is called a periodic series.

- **Tick.Latency** The time between when a request for a periodic series is received by the tick generator and when the first tick message in that series is sent shall be not more than one millisecond plus the latency due to message passing and processing.

- **Tick.OneShot** The tick generator shall provide a service whereby a module can request the delivery of a single tick message either after a specified delay (in the range from 1 ms to 1 hour), or at a specified absolute time. In this case the response message constitutes a one shot series.

- **Tick.SeriesID** Every request shall contain a series ID, specified by the requester, that identifies a particular series.

- **Tick.MultiSeries** The tick generator shall support multiple series being delivered to the same module. For example a periodic series and a one shot series, or multiple periodic series with different periods, all going to the same module shall be supported.

- **Tick.Message** Tick messages shall contain a counter, starting at one, that indicates which message in the series it is. Also tick messages shall contain the series ID number.

- **Tick.Cancel** The tick generator shall accept messages that cancel a pending or active series. The cancellation message shall contain the series ID. If the ID is invalid, there shall be no effect.

3.7 Other Nonfunctional Requirements

- **Nonfunctional.Requirement** The current demand of other nonfunctional requirements is unknown.

3.7.1 Performance Requirements

- **Performance.Requirement** The current demand of performance requirements is unknown.

3.7.2 Safety Requirements

- **Safety.Specs** Lunar IceCube's flight software shall adhere to The Safety Committee's standards.

- **Safety.Review** Lunar IceCube's flight software shall be reviewed by The Safety Committee to ensure no part of Lunar IceCube's flight software will interfere with EM-1.

3.7.3 Security Requirements

- **Security.ITAR** The flight software shall be assumed to be covered by ITAR.

3.7.4 Software Quality Attributes

- **VandV.RTE** All packages shall be free of flow errors and proven free of runtime errors in the sense meant by SPARK.

- **VandV.Specification** Higher level correctness properties, as encoded by pre- and postconditions and other SPARK assertions, shall also be proved as development resources allow. It is not required to encode and prove the entire specification of the software system.

It is understood that completely proving freedom of runtime errors may not be practical. In that case, failing proofs shall be investigated with focused testing that exercises the code where the proofs fail.

3.8 Other Requirements

- **Other.Upload** It shall not be required to support the ability to upload new versions of the flight software after deployment.

- **Other.Reuse** The software shall be written in as portable manner as feasible, and with an architecure that isolates mission specific components into well defined modules that have clean, generalizable interfaces.

It is our long term objective to develop a general purpose infrastructure for flight software called CubedOS. We intend to use the Lunar IceCube flight software as a basis for that effort.

4 Development Enviroment

4.1 Programming Language

A distingusing characteristic of the IceCube flight software is that most of it is written in SPARK. SPARK is a high integrity dialect of the Ada programming language that, together with the SPARK tools, allows developers to construct mathematical proofs of correctness (McCormick and Chapin, 2015). SPARK verifies correctness at four levels as outlined below.

1. The SPARK language makes certain kinds of errors impossible by virtue of the design of the language itself. For example, SPARK forbids pointers and dynamically allocated memory. Thus errors such as null pointer dereferences, memory leaks, and dangling pointers are impossible in SPARK programs (McCormick and Chapin, 2015).

2. SPARK verifies that no variables are used uninitialized, and that all computed results influence the program's eventual output. The later check ensures full error handling: error return values can't be ignored (McCormick and Chapin, 2015).

3. SPARK can prove "freedom from runtime error". This means that the program analyzed will never raise one of Ada's predefined exceptions. In particular, SPARK shows the program is free of the following errors (this list is not complete):

    - No division by zero.
    - No array access out of bounds.
    - No arithmetic overflow.
    - No violation of range constraints

4. SPARK can prove conformance to a software specification encoded in the program in the form of pre- and postconditions and other assertions. The strength of these proofs depends on the completeness (and correctness!) of the encoded specification.

All of SPARK's analysis is done statically, in that runtime checks are required. In fact, after SPARK shows a program is free from runtime error, it is reasonable to compile the program with an option that removes all of Ada's runtime checking. Those errors are, in theory, no longer possible (McCormick and Chapin, 2015). However, the IceCube flight software is built with the usual Ada runtime checks enabled anyway. There are three reasons for this. First, not all of the flight software is in SPARK. There are certain packages written in more expressive languages (full Ada or C). In addition, certain SPARK diagnostic messages have been explicitly supressed. Although we use traditional testing to verify the correctness of those sections, there is the possibility that the testing is incomplete (McCormick and Chapin, 2015). Second, not all proofs have been successfully discharged. The current state of the art in theorem proving technology can't easily construct proofs for every situation, even when the thing to be proved is true (McCormick and Chapin, 2015). Again, we use traditional testing to verify the correctness of those sections. In fact, SPARK helps us to focus our limited testing resources onto the areas where it is most needed. This is a major benefit to using SPARK (McCormick and Chapin, 2015). Finally, because of the harsh environment where IceCube will be flying, we recognize the possibilty of radition induced failures causing "impossible" situations to arise. Ada's normal runtime checking, along with the exceptions those checks might produce thus give the software a "last ditch" method of detecting and mitigating errors during the mission. It is useful to distinguish between Ada's runtime checks for things like range constraints and arithemtic overflow from the runtime

checking of SPARK assertions like pre- and postconditions(Barnes, 2014). The Ada compiler is able to optimize its runtime checks so their overhead is normally small. However, SPARK assertions can be very expensive to execute, even to the point of changing the asymptotic running time of the subprograms to which they are applied. Thus we do disable the runtime execution of SPARK assertions in the deployed flight software (but not during unit testing). To summarize:

- During development we verify freedom from information flow issues and run unit tests on the development platform with Ada runtime checks and SPARK assertion checks enabled.
- We attempt to prove the program free of runtime error and prove conformance to all SPARK assertions. Typically it is not possible to complete all of these proofs.
- We then review the failing proofs and edit the unit tests, if appropriate, to ensure they exercise the area where the proof is failing.
- We recompile the program with SPARK assertion checking disabled for performance reasons, and deploy the resulting executable to flight hardware.
- Testing continues on the flight hardware as described in Chapter 10.

4.2 Tools

In this section we discuss the tools we use during our software development process and summarize our method of using them.

4.2.1 SPARK

SPARK is both a programming language and a tool set (McCormick and Chapin, 2015). As a programming language, SPARK is a dialect of Ada with certain restrictions designed to make SPARK programs easier to analyze. For example, SPARK does not allow pointer types and exception handlers even though these things are allowed in full Ada (McCormick and Chapin, 2015). Also SPARK supports only a subset of Ada's tasking features (McCormick and Chapin, 2015). The SPARK restrictions are checked by the Ada compiler when the compiler is run in "SPARK mode". The selection of this mode is explicitly mentioned in all compiliation units where it is appropriate. This ensures that the compiler will not inadvertantly be run in normal mode on those units. The SPARK tools can be used interactively in AdaCore's integrated development environment, GNAT Programming Studio (GPS). This allows the programmer to more easily switch between writing code, and working with the result of SPARK's analysis. The tools can also be used non-interactively as part of automated build scripts. We do this with our continuous integration server, Jenkins, described in more detail in **Section 4.2.5.** This has the advantage of regularly doing a fresh, full analysis free of the possibility of confusion from left over analysis artifacts.

4.2.2 AdaControl

AdaControl will be used for controlling proper usage of style or programming rules. It can also be used as a tool to serach for use of various forms of progamming styles or design patterns.

4.2.3 Rational Rhapsody

Rational Rhapsody will be used to create UML diagrams, to create a visual represenation for systems engineers and the flight software developers. It natively supports reading in the Ada language.

4.2.4 GNATdoc

GNATdoc is a documentation tool for Ada which processes source files, extracts documentation directly from the sources, and generates annotated HTML files. It is based on the source cross-reference information (e.g. generated by GNAT for Ada files). This means that you should ensure that cross-reference information has been generated before generating the documentation. It also relies on standard comments that it extracts from the source code. The engine in charge of extracting them coupled with the cross-reference engine gives GNATdoc all the flexibility needed to generate accurate documentation, and report errors in case of wrong documentation.

4.2.5 Jenkins

The IceCube flight software is built, tested, and proved nightly using the open source continuous integration server, Jenkins. Specifically Jenkins does the following each night:

1. Does a fresh checkout of the project's code base from version control.

2. Compiles the project and the unit test programs.

3. Executes the unit test programs.

4. Does a SPARK flow analysis of the code base.

5. Does SPARK proofs of the code base.

The build is considered to have "failed" if any of the first three items fail. Jenkins does not fail the build for the SPARK analysis items because doing so would make it prohibitively difficult to develop the system without having Jenkins in a near-constant state of failure. However, Jenkins does publish reports that we review regularly. In particular, the Jenkins reports are at least reviewed before any milestone release.

5 Architecture

5.1 Traceability

To the greatest extent we shall use Apache Subversion (SVN), Jenkins, and Assembla to document the links between the system's level requirements and the flight software.

5.2 Delta Releases

A delta release is the release of software that includes only those modules which have changed or are new since the last release of software. We shall use SVN, to trace all delta releases for Lunar IceCube's flight software.

5.3 Components

All components for Lunar IceCube shall be modular, in the sense that they can be reused on future misssions. This by nature of CubedOS (the operating system lying on top of VxWorks), and object-oriented programming in general. It is important that components of software should be modular, because in future projects the time for developement would be greatly reduced.

6 Design

In this section we describe the design of CubedOS, including various design trade-offs made. This information is primarily of interest to developers looking to enhance the CubedOS infrastructure itself. Developers interested in only using CubedOS do not need to read this section unless they are interested in the rationale behind some of CubedOS's design decisions.

6.1 Message Manager

One of the central problems facing the message manager is the handling of message mailboxes. Since SPARK does not allow dynamically allocated memory McCormick and Chapin (2015), any method that involves dynamically allocating space for messages is not an option. Instead mailbox space must be allocated statically. This forces users of CubedOS to anticipate the maximum sizes of the mailboxes they require.

We considered three strategies for managing mailboxes.

- ***Define a Mailbox type that can hold a fixed number of pending messages. Create a separate mailbox for each module.*** This approach has the advantage of being simple. It also separates the total mailbox space by module. The value of the separation is that should one mailbox fill due to a dead or slow module, that won't impact the communication between other modules using different mailboxes.

  The main problem with this approach is that it tends to waste space. We assume most modules won't require a large number of pending messages (a large mailbox size). Yet if even one module does require a large size, the —Mailbox— type must be adjusted

to accommodate it, causing unnecessarily large mailboxes in most cases.

- ***Define a single Mailbox object that holds pending messages for all modules together.*** This approach addresses the disadvantage of the previous approach by consolidating the free space into a single mailbox object. Modules requiring a large number of pending messages can in effect "borrow" free mailbox space from modules with few pending messages.

  Implementing this design is more complicated, but a bigger disadvantage is that modules can interfere with each other. If one module is slow and gets backlogged, the (one and only) mailbox could fill up with messages for that module, thus preventing communication between other, unrelated modules. We feel this is a serious enough problem to rule out this design choice.

- ***Define a Mailbox type that is discriminated on its size. Create a separate mailbox for each module, tuned for that module's needs.*** This is a variation of the first approach except here the mailbox instances can each have a different size. The author of a module defines the size of the mailbox needed for that module; large mailboxes are only used where necessary and space is conserved. This approach also maintains the freedom from interference provided by the first approach.

The third approach is appealing but unfortunately does not allow replies to arbitrary modules in an environment where pointers are prohibited. Although a target mailbox can be mentioned by name when sending a message, the destination of any reply must be computed at runtime. Without pointers there is no way to create a map from module ID number to the mailbox for that module. The first approach can work around this problem by putting all

mailboxes in a common array and using the module ID number as an array index. However, that also requires all mailboxes to be the same size.

A more subtle problem with the third approach is that, in general, the author of a module can't know the maximum number of pending messages that are appropriate to specify for the module. The correct number is a system-wide artifact that depends in large measure on the number and rate of message send operations done by other modules. Setting the mailbox sizes is something best done during system integration, and that is more easily accomplished if all mailboxes are in their own package.

For these reasons CubedOS follows the first approach, using a generic package to supply the mailboxes. The package can be instantiated differently for each CubedOS application with mailbox sizes set appropriately for that application.

Selecting the most appropriate size for a the mailboxes is the one major issue remaining. If a mailbox fills during operation, sending further messages to that mailbox will fail. Senders do not block because doing so would require two entries on the Mailbox protected type, a violation of **Core.Ravenscar** (and an invitation to deadlock). Thus failure to send a message due to a full mailbox creates awkwardness for senders who must manually retry the send operation. Furthermore senders will not likely want to retry sending indefinitely and risk being tied up doing a send operation that will never succeed. This leaves open the question of how the failed send should ultimately be handled.

It would be better if mailbox sizes were tuned so that sending could never fail. However doing this requires high level reasoning about the flow of messages in the system. We propose constructing a formal model of module communication that includes finite sized mailboxes, and then use a model checker, or some other tool, to find a configuration of mailbox sizes

where no send operation can ever fail.

In lieu of such an analysis, human reasoning could be used to estimate appropriate mailbox sizes followed by rigorous testing. Of course, such an approach is likely to be more error prone than the formal approach.

If it can be formally shown that no send operation can ever fail, all calls to the Send procedure could be replaced with calls to UncheckedSend which does not return an error indication. Then all error handling in the modules related to failed sending could be removed. This would be a great simplification. However, we do *NOT* recommend this approach. Despite any formal modeling we might do in CubedOS's initial design, we foresee several possible ways mailboxes might overflow anyway:

1. Our formal model is incorrect.

2. We never get around to creating a formal model that works.

3. Users of CubedOS might not be in a position to augment the formal model to account for their own, application-specific modules.

4. A module thread dies for reasons outside the software's control (such as hardware failure). Although it might be possible to incorporate this scenario into the formal model, forcing freedom from mailbox overflow in this situation might impose unreasonable requirements on mailbox size or application architecture (or both).

For these reasons we advocate continued checking for failed send operations even if none are expected in normal operation.

6.2 Message Encoding

To remain generic the messages sent between modules are essentially nothing more than arrays of raw octets. Any structure on the information in a message is imposed by agreement between the sender and receiver of that message. This creates a degree of type unsafety and requires modules to defend against malformed messages at runtime. To mitigate this problem, each module contains an API package that provides convenience subprograms that encode and decode messages. The parameters of these subprograms are strongly typed, of course, so as long as they are used consistently compile-time type checking is retained.

Since the problem of encoding and decoding typed data has been solved before we leverage an existing standard, name External Data Representation (XDR). This standard defines a "wire format" (in our case a message data format) for various types and simple data structures. The advantage of using XDR is that it is well specified and understood. Using it simplifies our specification needs and invites the use of third party tools to manage message data.

We also considered using Abstract Syntax Notation One (ASN.1) as a message data format. Unlike XDR,(ITU, 2008) ASN.1 includes type information in the data stream allowing for more flexible message structures. However, ASN.1 also has more overhead, both in terms of space used in the message and time used to encode and decode the messages. For this reason we elected to use the simpler XDR approach.

The encoding/decoding subprograms in the API packages must contend with the richness of Ada's type system. In particular they should be able to handle multiple parameters of various types, called messageable types, where a messageable type is defined inductively as

follows:

- An integer type, an enumeration type, or a floating point type.
- An array of messageable types.
- A record containing only messageable type components.

Roughly a messageable type is any type allowed by the XDR standard (although we do not aim to provide support for XDR discriminated union types at this time).

The encoding and decoding of messageable types is tedious and repetative. Supporting subprograms to handle basic scalars and simple arrays are provided by way of a library package **CubedOS.Lib.XDR**. The handling of XDR arrays and records can be done using the lower level subprograms provided by the library.

Since the creation of the API packages is largely mechanical, CubedOS comes with a tool, **xdr2os3**, that converts an textual representation of the message types into a corresponding API package written in provable SPARK. In effect, **xdr2os3** plays a role similar to that played by **rpcgen** in many ONC RPC implementations. In particular, it converts a high level interface description into code that encodes/decodes messages to/from the implementer of that interface. However **xdr2os3** differs from other similar tools in several important respects:

- It targets CubedOS; the generated code makes use of CubedOS's XDR library for handling primitive types.
- It generates provable SPARK 2014 code.

- It contains extensions to the language that support Ada's ability to define range constrained scalar subtypes.

A CubedOS application developer defines the messages she needs for a module in an XDR file and then uses **xdr2os3** to generate the API package and related materials. The developer can then the subprograms in that package to encode messages for that module or decode messages from that module. In addition, **xdr2os3** provides subprograms the module author can use to decode incoming messages and encode replies.

Currently it is still possible for a develper to directly access message data but doing this is strongly not recommended.

7 Test Plan

To the greatest extent Vermont Technical College and Morehead State University will test the software, using two techniques:

- Software-in-the-Loop Simulation
- Hardware-in-the-Loop Simulation

7.1 Software-in-the-Loop Simulation

Software-in-the-Loop (SIL) Simulation is the process of creating mathematical simulations and models of the physical subsystems, and be allowed to execute on a host's computer. The purpose for SIL is defined as (Werner et al., 2015):

- Enable the inclusion of control algorithm functionality for which no model exists.
- Increase simulation speed by including compiled code in place of interpretive models.
- Verify that code generated from a model will function identically to the model.
- Guarantee that an algorithm in the modeling environment will function identically to that same algorithm executing in a production controller.

7.2 Hardware-in-the-Loop Simulation

Hardware-in-the-Loop (HIL) Simulation is the process of integrating all of the individual subsystems with the flight computer, execute and vent your flight software. This often requires a high fidelity model of each subsystem, so it tends to be more expensive than SIL

(University College of Southeast Norway, 2016). The purpose of using HIL vs SIL is defined as that HIL has all of the above for SIL as well as:

- Enhancing the quality of testing.
- Shortening development schedules.

8 Release Plan

8.1 Flight Software

The flight software deltas shall be released through Vermont Technical College's SVN server as they are upload and tested by Jenkins. This is to ensure that only personel who have access to the server shall have access to the software. The flight software repository is updated continually by Vermont Technical College's Dr. Peter Chapin. SVN will also allow us to traverse through previous versions in case there is a mistake (Collins-Sussman et al., 2011).

8.2 CubedOS

Lunar IceCube's flight software shall not be open source, due to it being under the restrictions of ITAR, however CubedOS (Lunar IceCube's underlying operating system) will be released to the public, although the exact date is unknown for now. For now only a select few shall have access to CubedOS's repository.

9 Risk Analysis

All risk analysis is already taken care of by SPARK (McCormick and Chapin, 2015). Due to it's formal checking, it will check for static run time errors and formally prove run time errors. All of SPARK's analysis is done statically. No runtime checks are required. In fact, after SPARK shows a program is free from runtime error, it is reasonable to compile the program with an option that removes all of Ada's runtime checking. Those errors are, in theory, no longer possible (McCormick and Chapin, 2015).

10 Fault Management

Fault management will fall under three categories:

1. Run Time Errors

2. Old Software

3. Sporadic Subsystem Behavior

10.1 Wrong Version of Software

In the event that we do not get every last piece of software written, it is important that we are able to upload new code to Lunar IceCube. To do so, we would replace the old version of software with new software through the Lithium-1 radio as our software interface.

10.2 Sporadic Subsystem Behavior

In the event that a subsystem may act outside of normal and expected behavior, we will implement Finite State Machines (FSMs) for each subsystem. These FSMs will have a default state that they will go to in the event of an error for a subsystem so that they may not cause Lunar IceCube's flight software to become put into an unknown state.

11 Verification

For verifying Lunar IceCube's flight software, we will use unit tests for all of the math procedures, that Lunar IceCube will utilize. Unit tests are taking known quantities and running them through a function, so that we may compare the results to the expected value. They are a part of a paradigm of programming known as Xtreme Programming (Huizinga and Kolawa, 2007). They are incredibly useful as they allow us to rapidly test and debug our procedures.

References

AdaCore (2016). Gnatdoc 18.0w documentation. `http://docs.adacore.com/gnatdoc-docs/users_guide/_build/html/`, [Accessed: 2017-27-04].

AdaLog (2016). Adacontrol user guide v1.18r9. `http://www.adalog.fr/compo/adacontrol_ug.html`, [Accessed: 2017-27-04].

Barnes, J. (2014). *Programming in Ada 2012*. Cambridge University Press, 1st edition.

CCSDS (2007). CCSDS 727.0-B-4. `https://public.ccsds.org/Pubs/727x0b4.pdf`, [Accessed: 2017-27-04].

CCSDS (2015a). CCSDS 734.1-B-1. `https://public.ccsds.org/Pubs/734x1b1.pdf`, [Accessed: 2017-27-04].

CCSDS (2015b). CCSDS 734.2-B-1. `https://public.ccsds.org/Pubs/734x2b1.pdf`, [Accessed: 2017-27-04].

Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and Weiss, H. (2007). *Delay-Tolerant Networking Architecture*. Network Working Group, 4838 edition.

Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2011). *Version Control with Subversion For Subversion 1.7*. Creative Commons License, 1st edition.

Eisler, M. (2006). *XDR: External Data Representation Standard*. Network Appliance, Inc., 4506 edition.

Huizinga, D. and Kolawa, A. (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE, 1st edition.

IBM (2016). The rational rhapsody family from ibm. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=BR&infotype=PM&appname=SWGE_RA_YA_USEN&htmlfid=RAB14010USEN&attachment=RAB14010USEN.PDF, [Accessed: 2017-27-04].

ITU (2008). Introduction to asn.1. http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx, [Accessed: 2017-27-04].

Jenkins (2016). Jenkins documentation. https://jenkins.io/doc/#jenkins-documentation, [Accessed: 2017-27-04].

Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2nd edition.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 1st edition.

McCormick, J. W. and Chapin, P. C. (2015). *Building High Integrity Applications with SPARK*. Cambridge University Press, 1st edition.

River, W. (2009). *VxWorks 6.8 Device Driver Developer's Guide*. Wind River, 1st edition.

Thurlow, R. (2009). *RPC: Remote Procedure Call Protocol Specification Version 2*. Network Working Group, 5531 edition.

University College of Southeast Norway (2016). Hardware in the loop simulation. http://home.hit.no/~hansha/documents/lab/Lab%20Work/HIL%20Simulation/Background/Introduction%20to%20HIL%20Simulation.pdf, [Accessed: 2017-27-04].

Werner, S., Masing, L., Lesniak, F., and Becker, J. (2015). Software-in-the-loop simulation of embedded control applications based on virtual platforms. *IEEE*.

Michael Glaser-Garbrick

Cell: (937) 545-0909
mglaserg@gmail.com
9530 Cutlers Trace
Dayton, Ohio 45458

Education

Morehead State University 2015–2017

Pursuing Masters in Space Systems Engineering.
Thesis consists of writing the Software Requirements Specifications for Lunar IceCube.
Anticipated graduation date is May 2017.

Morehead State University 2011–2015

Received Bachelors of Science with an Area of Concentration in Space Science.
Thesis consisted of designing an X-Band Up-Converter for CubeSats.

Morehead State University 2012–2015

Received Bachelors of Arts with an Area of Concentration in Mathematics.

Pending Publications

Development of Ground Station Software (GSSW) for the Cosmic X-Ray Background Nanosat-2 (CXBN-2) — IEEE

Toward an Improved Measurement of the Cosmic X-ray Background: Characterization and Adaptation of A CZT Imaging Array to be Flown on the Cosmic X-ray Background Nanosatellite-2 (CXBN-2) — JAI

Work

Graduate Assistantship at Morehead State University Fall 2016–Current

- Completed the flight software and was a key member of integration for CXBN2
- Working as part of a team for the software development for Lunar IceCube
- Developing the flight software using a P400K from Space Micro with VxWorks as the operating system

Research Fellowship at Morehead State University Summer 2016

- Wrote the flight software for CXBN2 using an MSP430
- Interfaced a Blue Gecko Bluetooth module with a Cortex-M processor
- Wrote a terminal serial program in C++ used to debug CXBN2

Graduate Assistantship at Morehead State University Fall 2015–Spring 2016

- Presented the software design at Critical Design Review for CXBN2
- Software development lead for CXBN2
- Designed a Reed-Solomon encoder and decoder for CXBN2

Research Assistantship at Morehead State University — Summer 2015

- Worked on the data analysis software for CXBN2 using C/C++
- Finished X-band up-converter originally done for senior thesis
- Analyzed the demagnetization factors of the magnetorquers for CXBN2

Internship at Vernon F. Glaser Inc — Summer 2014

- Learned about Agile and Scrum methodologies and its application to software development
- Used Extreme Programming practices for a MIME upload script
- Wrote an AES encryption and decryption program in Python

Research Fellowship at Morehead State University — Summer 2012

- Worked with Bob Twiggs on designing the PocketQub
- Designed the structure for PocketQubs using SolidWorks
- Helped develop the software for Eagle-1

Extracurricular

Phi Sigma Pi – National Honors Fraternity — 2013–2015

- Met on a weekly basis to discuss and solve issues within our chapter
- Volunteered at a soup kitchen to help the homeless
- Coordinated a fundraiser to help raise money for our philanthropy

Phi Eta Sigma – National Freshman Honors Fraternity — 2011–2015

- Must have had a minimum GPA of 3.5 by the end of freshman year
- Part of Region XI of the Phi Eta Sigma

Student Goverment Association — 2011–2012

- Freshman Representative Class of 2015
- Part of the economic subgroup to distribute fundings to other organizations on campus
- Learned to argue in a constructive way to achieve diplomatic solutions

Skills

C/C++ Python MATLAB Linux VxWorks Forward Error Correction Verilog

Points of Reference

Name: Kevin Brown
Title: Spacecraft Engineer
Phone: (408)373-2756
Email: kz.brown@moreheadstate.edu

Name: Charles Conner
Title: Electrical Engineer
Phone: (240)821-0611
Email: c.conner@moreheadstate.edu