Fault Tolerant and Flexible CubeSat Software Architecture

Greg Manyak

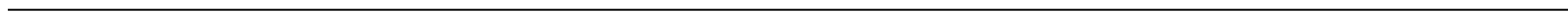
PolySat

California Polytechnic State University

A thesis submitted in partial fulfillment for the degree of

Master's of Science, Electrical Engineering

June 2011



© 2011
Greg Manyak
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Fault Tolerant and Flexible CubeSat Software Architecture
AUTHOR:	Greg Manyak
DATE SUBMITTED:	June 2011
COMMITTEE CHAIR:	Dr. John Bellardo Computer Engineering Assistant Professor Computer Engineering Department California Polytechnic State University
COMMITTEE MEMBER:	Dr. Jordi Puig-Suari Aerospace Professor Aerospace Department California Polytechnic State University
COMMITTEE MEMBER:	Dr. John Oliver Computer Engineering Assistant Professor Computer Engineering Department California Polytechnic State University

Abstract

The CubeSat pico-satellite is gaining popularity in both the educational and aerospace industries. Due to a lack of experience and constrained hardware capabilities, most of the university missions have been educational in nature. Cal Poly's project, PolySat, has gained significant experience from the launch of five CubeSats and has designed an entirely new hardware platform based on the knowledge gained from these missions. This hardware is a significant upgrade from what the previous missions used and has greatly increased the capabilities of the software, including supporting the use of the open source operating system Linux.

Leveraging the previous PolySat experience, a new design approach has been followed for the development of a fault tolerant and flexible software architecture. As a result, a set of processes and custom libraries that run within Linux have been designed and implemented. Furthermore, an emphasis has been placed on fault tolerance with two features: a software watchdog and digital command signing capability. Lastly, a survey of related CubeSat projects and software fault tolerance papers has been conducted to determine that this new system is sufficient to meet the desired goals.

To my parents, Sue and George Manyak, who've never discouraged me from trying anything and made this college experience possible.

Acknowledgements

This design would not have been possible without the significant guidance and contributions made by Dr. John Bellardo, PolySat's software and my thesis advisor. I would also like to acknowledge the past and present PolySat members who have dedicated countless hours to bring this project to the level it has reached today. Particularly: Austin Williams, who has designed the hardware platform that this software runs on; Sean Fitzsimmons, who has provided constant insight and commentary on the software design; and Dr. Jordi Puig-Suari, the fearless PolySat advisor who has pushed us to produce these amazing satellites.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 CubeSats . . . 1

1.2 PolySat . . . 1

1.3 CubeSat Software . . . 2

1.4 Thesis Organization . . . 3

1.5 Scope of Thesis . . . 3

2 First Generation Software Architecture 5

2.1 Design . . . 5

2.1.1 Modes of Operation . . . 6

2.1.2 Communications Controller . . . 7

2.1.3 Command and Data Handling Controller . . . 8

2.1.4 I^2C Fault Tolerance Upgrade . . . 9

2.2 Results . . . 9

2.2.1 CP3 . . . 9

2.2.2 CP4 . . . 9

2.2.3 CP6 . . . 10

2.3 Lessons Learned . . . 10

2.3.1 Development Environment . . . 11

2.3.2 Hardware versus Software Modularity . . . 12

2.3.3 Modular Command System . . . 12

2.3.4 Collect More Telemetry . . . 12

3 Second Generation Software Design Approach **15**
3.1 Design Guidelines 15
3.2 Requirements 17
3.2.1 Functional 17
3.2.2 Non-Functional 18
3.3 New Design Philosophies 23
3.3.1 Software Modularity 23
3.3.2 Encouragement of Parallel Development 25
3.3.3 Pre-Existing Code 25

4 New System Architecture **27**
4.1 Software Overview 27
4.1.1 Operation Modes 29
4.1.2 Inter-Process Communications 29
4.2 Processes 30
4.2.1 Beacon 31
4.2.2 Communication 32
4.2.3 System Manager 33
4.2.4 Data Logger 34
4.2.5 Watchdog 35
4.2.6 Mission Specific Static Processes 35
4.2.7 Temporary Processes 35
4.3 Abstraction Libraries 36
4.3.1 Event Handler 37
4.3.2 Command Handler 38
4.3.3 Inter-Process Communication Abstraction 38
4.3.4 Configuration Management 39
4.3.5 Standard Debug/Error Interface 40
4.3.6 Cryptography 41
4.3.7 PolySat Library Base 41
4.3.8 Database 43
4.4 Results 43
4.4.1 Memory Footprints 43
4.4.2 Common Library Event Latencies 45

4.4.3 Scheduler Accuracy . . . 46
4.5 Design Success . . . 47

5 Fault Tolerance 49
5.1 Cryptography: Digitally Signed Commands . . . 49
5.1.1 Goals . . . 49
5.1.2 Requirements . . . 50
5.1.3 Implementation . . . 51
5.1.4 Results . . . 54
5.2 Software Watchdog . . . 55
5.2.1 Goals . . . 55
5.2.2 Requirements . . . 56
5.2.3 General Implementation . . . 57
5.2.4 Static Process Watching Implementation . . . 58
5.2.5 Status Verification Implementation . . . 59
5.2.6 Temporary Process Watching Implementation . . . 60
5.2.7 Command Validation Implementation . . . 61
5.2.8 Detection and Recovery Limitations . . . 61
5.2.9 Results . . . 62

6 Related Satellite Projects 65
6.1 Summary of Projects . . . 65
6.2 BEESat . . . 66
6.2.1 BEESat Spacecraft . . . 66
6.2.2 Software Architecture . . . 66
6.2.3 Results . . . 67
6.2.4 Analysis . . . 67
6.3 Cute-1.7 . . . 67
6.3.1 The Cute-1.7 System . . . 68
6.3.2 Radiation Protection . . . 68
6.3.3 Analysis . . . 69
6.4 iTU-pSAT I . . . 70
6.4.1 Software . . . 70
6.4.2 Analysis . . . 70
6.5 KySat-1 . . . 70

6.5.1 System, Requirements, and Software Infrastructure 71
6.5.2 Flight Software Overview 71
6.5.3 Fault Tolerance 72
6.5.4 Analysis 74
6.6 MEROPE 75
6.6.1 C&DH Hardware 75
6.6.2 Software 76
6.6.3 Analysis 76
6.7 PW-Sat 77
6.7.1 Hardware Design 77
6.7.2 Software Design 77
6.7.3 Analysis 79
6.8 QuakeSat 80
6.8.1 Operating System Selection 80
6.8.2 Satellite Software 81
6.8.3 Analysis 82
6.9 STUDSAT 82
6.9.1 Software Architecture 83
6.9.2 Analysis 84
6.10 UWE-1 and UWE-2 Satellites 84
6.10.1 UWE Platform 84
6.10.2 UWE-1 Software 85
6.10.3 UWE-2 Software 85
6.10.4 Analysis 86

7 Related Works 87

7.1 *Software Implemented Fault Tolerance* 87
7.1.1 Model 87
7.1.2 Watchdog 88
7.1.3 Fault Tolerance Library 89
7.1.4 Multi-Dimensional File System 89
7.1.5 Results 89
7.1.6 Analysis 90
7.2 NASA's Software Fault Tolerance: a Tutorial 90

7.2.1 Single-Version Software Fault Tolerance Techniques 90
7.2.2 Multi-version Software Fault Tolerance Techniques 92
7.2.3 Applied Techniques 92
7.3 Flexible Fault Tolerance in Configurable Middleware 93
7.3.1 Configurable Fault-Tolerant Mechanisms 93
7.3.2 Results 94
7.3.3 Analysis 95
7.4 *Great Watchdogs*, Version 1.2 96
7.4.1 Technical Report Summary 96
7.4.2 Watchdogs in the PolySat System 98

8 Conclusion 99
8.1 Overall Design Success 99
8.2 Implementation Progress 100
8.3 Future Work 100
8.3.1 Open Source Libraries 101
8.3.2 Radio Communication Optimization 101
8.3.3 Scheduler Power Management 101
8.3.4 Multi-Version Software Watchdog 102

A PolySat Coding Standard 103
A.1 Indentation 103
A.1.1 Number of Spaces Per Tab 103
A.1.2 Curly-Brace Location 103
A.1.3 Maximum Line Length 103
A.1.4 If-Statements 103
A.1.5 Break Up Code 104
A.2 Identifiers 104
A.2.1 Single Letter Variable Names 104
A.2.2 Multiple Word Variable Names 104
A.2.3 Abbreviations 104
A.2.4 Hungarian Notation 104
A.2.5 Variable Name Length 104
A.3 Constants 105
A.3.1 Magic Numbers 105

A.3.2 Use . . . 105
A.3.3 Constant Naming . . . 105
A.4 Source Files . . . 105
A.4.1 Contents . . . 105
A.4.2 Description . . . 105
A.4.3 Function Headers . . . 105
A.5 Header Files . . . 107
A.5.1 Structure . . . 107
A.5.2 Header File Exclusion . . . 107
A.6 Comments . . . 107
A.6.1 Style . . . 107
A.6.2 Flow Control . . . 108
A.6.3 Microcontroller Specific Instructions . . . 108
A.6.4 General Instructions . . . 108
A.7 Horizontal Whitespace . . . 108
A.7.1 Keywords and Commas . . . 108
A.7.2 Operators . . . 108
A.7.3 Between Functions . . . 108
A.7.4 Function Names . . . 109
A.8 Functions . . . 109
A.8.1 Local Variables . . . 109
A.8.2 Error Handling . . . 109
A.8.3 Parameters . . . 109

B PolySat Formal Code Review Process 111
B.1 Work Product . . . 111
B.2 Participants . . . 111
B.3 Preparing for Inspection . . . 112
B.4 Meeting . . . 112
B.5 Rework . . . 113
B.6 Follow-Up . . . 114

References 115

List of Figures

2.1 First Generation Hardware Block Diagram 6
2.2 I^2C bus event data from CP6 11

3.1 Second Generation Hardware Block Diagram 19

4.1 Software Architecture 28
4.2 Event Handler State Machine 37
4.3 Command Handler Flow Diagram 39
4.4 Example Configuration File 40
4.5 Process Initialize Flow Diagram 42

5.1 Signed Command Flow 51
5.2 Command Signature Generation Flow 52
5.3 Command Validation Generation Flow 53
5.4 Static Process Watching 58

LIST OF FIGURES

List of Tables

3.1 Non-Functional System Requirements 21
3.2 Non-Functional Process Requirements 22
3.3 Non-Functional Personnel Requirements 23

4.1 Various Component's Non-Volatile Storage Requirements 44
4.2 Static Process Runtime Memory Requirements 44
4.3 Latency of Common Library Events 45
4.4 Scheduler Accuracy Measurements 46

5.1 Digital Command Signature Verification Timing 55
5.2 Validated Requirements for Watchdog 63
5.3 Watchdog Command Validation Timing Results 63

6.1 Comparison of specifications of CubeSats 65

1

Introduction

1.1 CubeSats

CubeSat class spacecraft have become increasingly popular since their introduction in 1999 by Cal Poly professor Dr. Jordi Puig-Suari and professor Bob Twiggs, then at Stanford [1], thanks to their limited volume of 10cm^3 [2].The standardized size requires some creativity but more importantly it makes the cost of development and launch relatively affordable for nearly any organization looking to build a spacecraft because it resulted in a standardized deployment device, the Poly Pico-satellite Orbital Deployer (P-POD) [2]. This low cost has resulted in the CubeSat's prevalence in both the educational and private sectors. This is especially true at Cal Poly, whose multidisciplinary satellite project, PolySat, has been at the forefront of the CubeSat community since the specification was first introduced.

1.2 PolySat

PolySat has launched a total of 5 CubeSats since the start of the program in 1999: CP1 through CP4 and CP6. All of the satellites after CP1 used the same flight software, with minor revisions. The overall system was designed with more of an emphasis on hardware than software modularity, and thus the software was fairly simple, with no operating system and mission specific components were frequently intermixed with basic functionality. Although this system was relatively effective at meeting the requirements of these missions, far more ambitious missions are

now regularly being proposed to the PolySat team.

Recently, PolySat was approached with a new mission, LightSail-1, for which a more powerful and robust avionics platform was desired. This presented PolySat with the [funding] opportunity to leverage the experience gained from the first generation platform and build something entirely new. The new base hardware consists of a small number of very capable, yet low power, components all on a single board. This type of design is suited perfectly for the CubeSat because it minimized volume and maximized potential payloads, thanks to both the improved computing power and the available space.

1.3 CubeSat Software

At the earliest stages of the design, the previous software team was consulted to determine what areas could be best improved upon to facilitate development since the current team had mostly been involved in modifying and extending the existing platform. The most common suggestion was to find a platform that had significant community support and pre-existing code that can be utilized. As a result, the new microprocessor was chosen with support for the open source operating system, Linux. This selection has far-reaching implications for the software architecture; some concepts from the past designs can be preserved, but a wholly new software system should be implemented to maximize the benefit.

A CubeSat software architecture is vital to the development of the spacecraft and careful attention must be given to each design decision. A proper CubeSat software system will require minimal, if any, revisions between each unique mission and thus enable quick turn around times to support the demands of the customer and launch scheduling. During the design of a software architecture, it is important to consider this concept because once a hardware platform has been established, updated software may be the only significant component necessary to support a new mission.

The architecture can easily dictate the success of the entire mission, as well. This is particularly true in CubeSats because they are highly volume constrained systems that are encouraged to utilize minimal hardware to increase payload volume. This differs significantly from the typical, large satellite that would include a variety of redundant components in order to protect itself against the

harsh environment of space where ionizing radiation can cause bit flips or even stuck bits [3]. Thus, it is critical that the software architecture provides features that insure reliable operation, in spite of potential radiation effects.

These two goals are the driving forces behind the design of the new PolySat software architecture, which aims to be both extensible and fault tolerant.

1.4 Thesis Organization

This thesis details the new software system and is organized as follows. Chapter 2 revisits the first generation platform and some of the lessons learned from the software. Chapter 3 describes the new design guidelines, requirements, and philosophies for the new software system. Chapter 4 details the system architecture, focusing on the two major layers of the new system: processes and libraries. Chapter 5 discusses the two primary fault tolerant features of the system including their goals, requirements, implementation, and results. Chapter 6 looks at a number of other CubeSat projects by analyzing and comparing their software systems to the one presented in this thesis. Chapter 7 looks at various technical and published papers discussing different fault tolerant mechanisms that were considered for this design. Chapter 8 summarizes the results and discusses the implementation progress and future work.

1.5 Scope of Thesis

This thesis discusses the inspiration and the design of the new flexible and fault tolerant software architecture for PolySat's newly designed avionics hardware platform, in addition to a review and comparison of a number of other CubeSat projects. The design of the modules is mostly limited to the high level and conceptual design; the watchdog and cryptography components are the only significant aspects completely designed and implemented as part of this thesis. This is primarily due to the considerable number of PolySat students who have participated in the implementation and low level design of some of the other modules thus far.

2

First Generation Software Architecture

PolySat's first generation software architecture was based on Cal Poly's second satellite's flight software and was expanded to be more generic for the purpose of supporting future missions more easily [4]. This software obtained flight experience on the CP3, CP4, and CP6 spacecraft with varying degrees of success. The last spacecraft to use it will be the CP5 mission, which is intended to launch sometime in mid-2012.

2.1 Design

Due to the tightly coupled nature of hardware and software, it is important to understand the hardware design first. The overall platform design was built with a fault tolerance architecture mostly by utilizing redundant hardware. The reason for this choice is that the majority of the components were inexpensive, in both cost and power consumption, for redundancy to be possible [5]. The biggest example of this is on the Command and Data Handling (C&DH) board, which was comprised of 3 separate PIC18F6720 microcontrollers interfaced via I^2C. One controller was responsible for the C&DH related processing, including interfacing with EEPROMs for non-volatile storage and the payload, while the other two primarily dealt with the communications of the spacecraft, each running the exact same software. The C&DH was responsible for toggling between the two communication chains. A generalized block diagram of the first generation

hardware is shown in Figure 2.1.

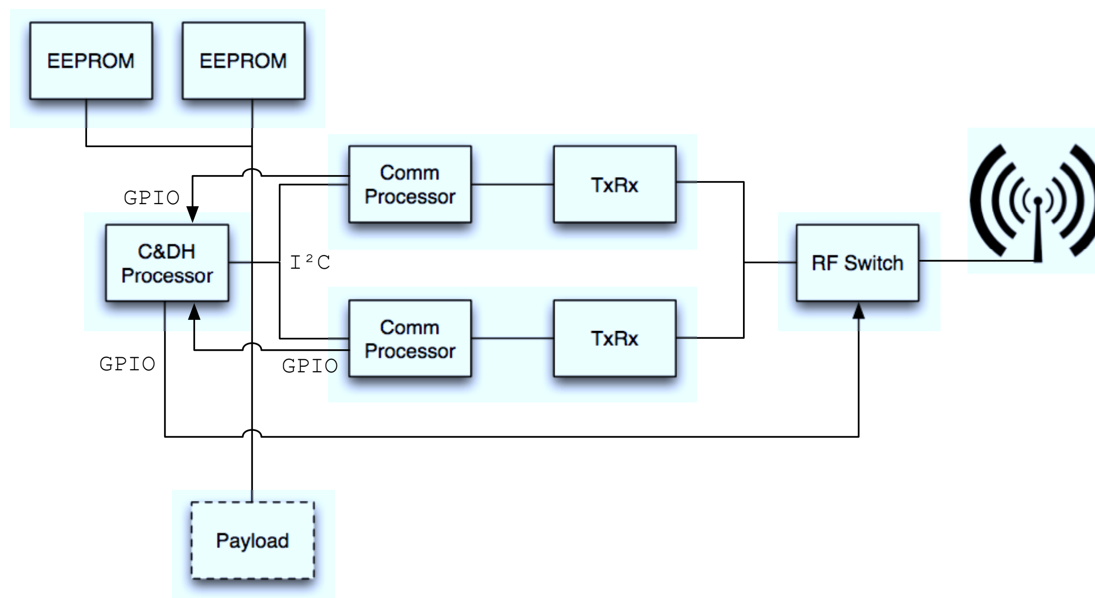


Figure 2.1: First Generation Hardware Block Diagram - This figure shows the redundant communication systems and the memory/payload interfaces to the C&DH microprocessor of the first generation system.

A simple, interrupt-based software architecture is used in all of the controller software: a `while(1)` loop continually checks status flags, which are set in interrupt service routines. These interrupts can be triggered by one of two things: an I^{2}C message, or a timer expiration.

Typically, there is a fast timer, for events that are very frequent or require very accurate deadlines, and a slower timer, for less frequent events with looser deadline requirements. Inside of the timer interrupt handler, timer generated interrupts are counted until the number of overflows to meet the desired time have occurred and then a global flag is set, which will be read in the main loop.

2.1.1 Modes of Operation

The entire spacecraft has three primary modes of operation: pre-ops, normal-ops, and contingency. Although the modes of operation have the same names for both the communications and C&DH controllers, they are selected independently.

Pre-ops is the mode in which both controllers operate when the spacecraft is first ejected from the P-POD. In this state, the spacecraft merely collects

telemetry data and beacons it at a regular rate, until a command is uplinked from the ground station. Once a command has been received, the communications processor will move into the normal ops mode for the remainder of the mission. A contingency mode was originally part of the design, where the communications processor could control the entire spacecraft, however, this was never implemented in a flight unit.

As for the C&DH processor, normal ops mode will only be entered after a specific command is sent up. It was designed such that normal ops enabled autonomous transmissions to trigger at a higher rate and it also allowed other, more complex commands to be executed (i.e. those dealing with the payload or attitude control). The C&DH will return to the pre-ops state after a certain amount of time (typically on the order of days), as to reset the beacon rate and essentially stay in a lower power mode.

2.1.2 Communications Controller

Both the communication software and hardware subsystem were mostly developed as a senior project by Chris Noe [5]. This software has remained mostly untouched through all of the CP flights due to its complexity and relative effectiveness.

Traditional ground stations utilize a hardware terminal node controller (TNC), which is the radio control interface, and this design was used in some of the earlier CubeSat communication systems, however, this requires significant power and volume. For CP2 the more appropriate solution was found to be using a software TNC instead. The primary function of a TNC is to forward data to and from the transceiver. In this system, the data is typically going to be sent to the C&DH controller. Each communication controller is interfaced to its own transceiver via a two wire bus and will toggle a GPIO to indicate to the C&DH that data has been received. The code for the transceiver driver is very complex because of the intermixing of a “bit-bang” protocol and the communication encoding scheme, which was required due to the speed-sensitive nature of this interface.

The communication controller examines the command byte of each received packet so that a few basic commands can be executed before forwarding the data to the C&DH. For example, a command can be executed by the C&DH to reset the communication controller. There is also a command that can be issued to cause the communication controller to transmit a basic morse-code beacon,

which can be requested up to 10 times and can be useful for tuning the radios to find the transmitting frequency of the satellite. Some missions also implemented a command to enable the payload from this controller; even if the C&DH has failed, this allows a one-off experiment to be triggered.

2.1.3 Command and Data Handling Controller

The C&DH controller was responsible for a wide variety of aspects on the system including spacecraft telemetry data collection, attitude determination and control experiment execution, and high level communication system and payload control. The precise functionality evolved and changed on a mission-by-mission basis, but these overall goals generally remained the same.

All three satellites had autonomous sensor data collection with a customizable rate at the granularity of the lower speed timer interrupt. The sensor data was collected and stored in I^2C EEPROMs, which limited the total amount of stored data to 256KB. In order to deal with this limitation, the operator could issue a command to the spacecraft to treat the EEPROMs as circular buffers, or stop the saving of data after the capacity has been reached.

The sensors were divided up into distinct groups to allow for different rates for each, based on the needs of the mission. There was also an I^2C real time clock device on the C&DH board, which was used to add timestamps to the sensor snapshots.

Furthermore, the C&DH was responsible for controlling and monitoring the other major subsystems it was connected to: communications, attitude determination and control (ADCS), power, and payload. One of the timed events on the C&DH was toggling between the two redundant communication chains, so that in the event of one failing, the other would be used soon afterwards and communication would only be temporarily interrupted. Though the ADCS functionality was limited on the most recent versions of the legacy flight software, the C&DH was responsible for executing an attitude control algorithm, B-dot, for de-tumbling the CubeSat. As for the power subsystem, the C&DH mostly played a supervisor role by interfacing with battery monitors and recording voltage and current levels of the batteries. Lastly, the C&DH was usually interfaced with a payload subsystem controller and would forward ground station commands intended for it.

2.1.4 I^{2}C Fault Tolerance Upgrade

The functionality of the I^{2}C bus is imperative to the success of this system because of the link between the C&DH and communication controllers. In order to make this connection more robust, a CRC was added to each inter-process message as part of a senior project [6]. The primary benefit of this was that unintended messages should no longer be acted on, which could potentially put the spacecraft into an unintended state. As a result, a local retry was also implemented so that commands that originated from the ground did not have to be resent. This feature also enabled the collection of I^{2}C telemetry, allowing for further characterization of the bus in space.

2.2 Results

The three spacecraft that have flown thus far were using very similar versions of the software and hardware. CP3 and CP6 shared the same primary mission, but CP6 had some receive sensitivity improvements in an attempt to improve uplink from the ground station, plus an additional payload from Naval Research Labs. CP4 was a re-fly of CP2, which did not reach orbit. The CP5 spacecraft, which is manifested on the ELaNa VI mission, is slated to be launched in mid-2012 and uses a modified C&DH board with only a single communication processor and radio.

2.2.1 CP3

CP3 was launched in April 2007 and successfully demonstrated a majority of the basic functionality. As of writing this thesis, it still regularly communicates with the Cal Poly ground station, but the communication is mostly one way because uplink is very difficult to achieve as a result of very poor receive sensitivity. Unfortunately, this limited the uplink to simple commands and payload operations were never able to be completed.

2.2.2 CP4

The CP4 spacecraft went into orbit with CP3, but unfortunately after a short time it experienced a critical failure preventing C&DH or payload commands from

being executed. It was suspected that an I^{2}C bus failure was to blame [6], which would have disabled communications between the C&DH and communication controllers and effectively shut down the spacecraft.

2.2.3 CP6

PolySat's most recently flown spacecraft, CP6, was launched in May 2009. CP6's improved receive sensitivity resulted in arguably the most effective PolySat mission. The majority of the command set was able to be transmitted and executed on the spacecraft, including the demonstration of the B-dot de-tumbling algorithm.

CP6 was also the first mission to heavily involve amateur radio operators around the world. This was primarily thanks to a freely distributed program that decoded packets and also sent the data back to a central database at Cal Poly. In fact, the data sent to this database through this program was over double the amount of data of Cal Poly, based on numbers from a couple months after launch [7].

Unfortunately, a few months after the launch, CP6 stopped responding for a significant period of time and as of April 2011, only one of the communication processors responds to simple commands, and there is no response from C&DH commands.

CP6 was the only mission thus far to fly with the I^{2}C telemetry collection enabled. The gathered results over the lifetime of the mission are shown in Figure 2.2. The data shows that there was an approximately 8% error rate in the bus transactions. Although this number is higher than ideal, it does not seem to definitively indicate that the I^{2}C bus is the source of the CP4 and CP6 failures.

2.3 Lessons Learned

Multiple generations of students have used this architecture in CubeSat designs since it was first developed and although minor improvements were made for each new mission, there have always been some lingering concerns. The most problematic of these existed as a result of the development processes rather than the design.

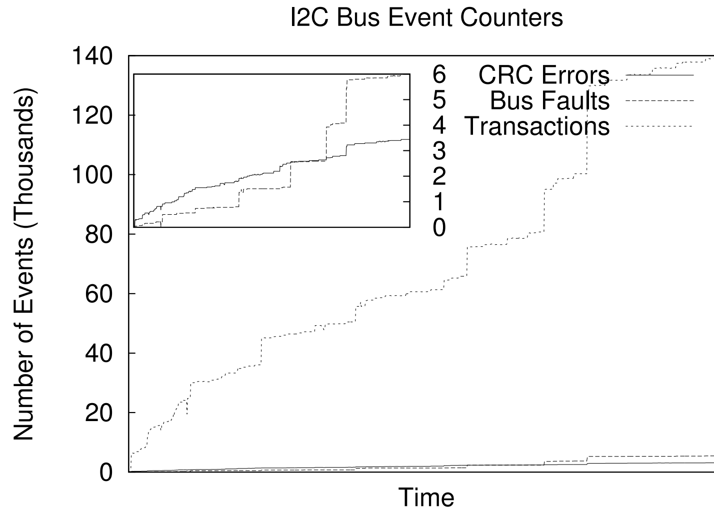


Figure 2.2: I²C bus event data from CP6 - Inset in upper left shows only the CRC error and bus fault event counters. Time progresses left-to-right, but not proportionally due to periods of time when no data was downlinked.

2.3.1 Development Environment

One of the most common source of problems was the software provided by the microcontroller manufacturer. For example, the host machine seems to spontaneously lose communication with the interface to the target microcontroller and the IDE will provide only cryptic error messages that sometimes can be fixed by restarting the software, but other times require a full system reboot. These type of issues are especially prevalent when attempting to use the single-step debugger, which itself is an appreciated option, but is not always worth the trouble when these problems arise so regularly.

Unfortunately, the problems with the development environment did not stop there. When re-imaging the host machines in the lab, a slightly newer compiler was installed and resulted in completely non-functional software. This implies that there are strong dependencies to the version of the compiler and thus would require PolySat to stay with this, now quite old, version of the compiler in order to maintain backwards compatibility with the old projects.

Based on these issues that we have had with the development software and compiler, the general consensus in the lab was to move away from the PIC microcontroller platform.

2.3.2 Hardware versus Software Modularity

A significant strategy of the first generation system design was to favor hardware modularity over software modularity, so that fault tolerance through redundant hardware could be provided. Hardware redundancy is a fairly common design in full size spacecraft and one of the common concerns about CubeSats is they do not support such designs very easily. However, PolySat and other universities (see Chapter 6) operational history have shown there is negligible benefit with hardware redundancy, at the cost of significant complexity added to the system. Specifically with the PolySat system there has been no known event where the C&DH was able to continue operating while one of the communication chains had failed.

2.3.3 Modular Command System

The command handling module in the first generation software architecture existed entirely in one source file, based around a massive `switch` statement. Each command had its own case and often all of the code to execute the command was there, rather than making a function call. There was also a large amount of repeated code for transmitting negative acknowledgements upon incorrect usage of commands.

As the capabilities increased with each revision, so did the file size: with 40 commands, it was nearly 1500 lines for CP3 and then for CP6, it grew to almost 2000 lines with 60 commands. This file violated numerous software engineering guidelines, but its growth could not be stopped because it was already deeply ingrained in the software's architecture. The lesson from this behemoth of a source file is that emphasis should be placed on a modular command system from the initial design because it is guaranteed that commands will be added for future missions.

2.3.4 Collect More Telemetry

A lesson that was learned in part before CP6's launch was that more telemetry is a good thing, as demonstrated by the additional I^2C telemetry collection. However, there's no reason to stop there; this policy should be applicable to any

interface, software or hardware. With a larger variety of telemetry sources, the characterization and debugging capabilities are vastly increased.

3

Second Generation Software Design Approach

To support the second generation software design, an updated set of guidelines, requirements, and philosophies were created. These were developed as a culmination of the experience from developing and utilizing the first generation system, in addition to some completely new concepts as a result of new mission proposals and improved hardware capabilities.

3.1 Design Guidelines

Two major goals of the CubeSat platform are to be inexpensive and responsive. In order to support these goals, it is necessary for the CubeSat developer to have a flexible hardware and software system. PolySat refers to the combination of the basic electronics packages (power and computing) and a standard software platform as the avionics system. It is important that all of these components are flexible so that a minimum amount of time and money is spent for each new mission re-designing the basic architecture. A component whose importance is seemingly often over-looked in the CubeSat community is the software architecture, which this work is solely focused on designing.

The primary goal of this software architecture is to provide a design that is both extensible and robust, in order to support the overall goal of having a flexible CubeSat avionics platform. In order to meet this goal, there are a few rules that guide the design of the software system: a flexible and effective command set

must be provided, all major software components need to be aware of radiation events, and the more telemetry collected, the better.

The method of interaction with the CubeSat after it has been launched is through commands, which is a message sent from an external source that is intended to trigger some type of action and is sent from a ground station. A ground station is the location at which data can be transmitted over the air to the spacecraft. Commands can also be intra-satellite, from one software module to another. Both sources of commands can achieve the same results, such as starting an experiment or change a parameter. As a rule, the majority of the parameters of the different aspects of the spacecraft should be easily modified through a command.

The next design guideline is that all components should be aware of potential radiation events. This is important because of the effects the space environment can have on the spacecraft and ultimately the software. Radiation events, such as Single Event Upsets (SEU) that typically result in bit flips [3], can necessitate the restart of software modules or even the entire processor, when detected. What this means for the software modules themselves, is that they need to be able to tolerate potentially unwarned termination or restarting. Mechanisms like storing state information in non-volatile memory can help reduce the issues that could arise from unexpected termination. Software modules need to contain the necessary storage and recovery mechanisms in order to be able operate in the unpredictable environment of space.

The last guideline states that the more telemetry collected, the happier the mission operators and system developers will be. Often, only the minimum amount of telemetry is collected, and can make failures very difficult to diagnose because the necessary information was just not available, as was the case with CP4 (discussed in Section 2.2),. This problem can be avoided by preemptively collecting all potentially useful telemetry and providing the option to downlink it. Hopefully, if everything works as intended, the information will not be necessary. However, in the event of a failure, the telemetry can be used to diagnose the problem. Thus, all modules are responsible for collecting relevant and important telemetry.

These guidelines have been generated as a result of experience with the previous generation system and a survey of other CubeSat software architectures

(Chapter 6). They have been emphasized in the new software design in an attempt to avoid requiring significant revisions to the core components, in order to add these type of features during the lifespan of the project. In general, the less the platform has to be revised, the easier it will be to use for different missions.

3.2 Requirements

The requirements for the new software system are driven mostly by previous experience and the general goals for the new system. However, since the second generation platform is being developed in parallel with some of the missions that plan to utilize it, LightSail-1 [8] and CP7 [9], some of their requirements have served as guides, as well. The requirements are separated into functional and non-functional, where the functional requirements are those that the software system must be able to accomplish when it is completed, and the non-functional are items that must be considered and supported throughout the design and its life span.

3.2.1 Functional

A few basic functional requirements exist for this software system. The majority of these address the required operations of the spacecraft itself. These requirements are small in number because there are only a small set of functions that the satellite must be able to complete for all types of missions. It is a relatively simple task to derive the functional requirements for a single mission. However, it is a much more complex problem when considering a larger number of missions and the common components must be found to facilitate optimal reuse, which is what is required when designing a platform.

First and foremost, it is required that the system must also have the capability to periodically collect telemetry and store it in non-volatile storage. This requirement enables the spacecraft operator to have the capability of downloading a more detailed history that may not be available from just beacon data.

The next requirement is that the system must have a beaconing capability. A beacon is a periodic broadcast of vital telemetry data, which can be used for tracking the spacecraft after its been put into orbit and also provides valuable information without requiring a specific command.

Along the same lines, the system also must have the capability to transmit AX.25 packets, at least for the first revision, because the existing ground station infrastructure at Cal Poly supports this protocol. This requirement exists primarily because PolySat does not have the funding to upgrade the ground station to support other protocols that may be more efficient. Moreover, it is commonly supported by amateur radio ground stations around the world that can provide additional downlink opportunities and have been vital in data collection for past PolySat missions (see Section 2.2.3).

To support the use of the spacecraft and these functions, there are also a few vital commands that must be implemented. These include resetting the spacecraft, changing the rate of telemetry collection, downlinking the telemetry, and changing the rate of beacon transmission.

Lastly, the system must be able to execute these commands either immediately or at some time in the future. Commonly, certain events may need to occur at a time when the satellite is not passing over the ground station, meaning that no uplink is available. In order to enable this type of event scheduling, the system needs to be able to store commands and execute them a specified amount of time later (either as a timestamp or time delta). The optimal support for command scheduling would include both an offset from time of uplink or a time stamp.

3.2.2 Non-Functional

There are also numerous non-functional requirements for the new software system design, which are divided into system, process, and personnel related. These requirements are very important to the platform because they detail all of the non-behavior factors of the system. A particular emphasis has been placed on those requirements dealing with development on the platform because if the platform is simple to develop for, it is more likely the next generation of PolySat students will need to continue to use it.

Prior to the overall design of the software architecture discussed in this thesis, both the hardware and software operating environments had been selected. The software will run on a AT91SAM9G20 microprocessor, which is an ARM9-based system on a chip that is clocked at 400MHz. Supporting this chip is 64MB of SDRAM, a 16MB non-volatile phase change memory, and 512MB of non-volatile NAND flash memory. The size of these memory chips will potentially scale with

future mission needs because of the variety of available drop-in replacements since they are fairly standard packages. For example, 128MB SDRAM chips that are a drop in replacement for the existing chip have been found, in the event a mission requires more volatile memory. There is also a microSD slot, which can add up to 32GB of storage, at the time of this work. This system has been measured to require approximately 180 mW while the CPU is idle and about 300 mW when processing a workload that requires frequent access to the SDRAM.

Other important hardware components include two hardware watchdog timers (WDT): one internal to the processor and another that is external. The internal one has a shorter maximum timeout of 16 seconds and triggers a soft reset (only the processor is rebooted); whereas the external watchdog has a longer timeout of 60 seconds and results in a hard reset of the entire spacecraft by temporarily removing power from all components. A block diagram of the hardware configuration is shown in Figure 3.1.

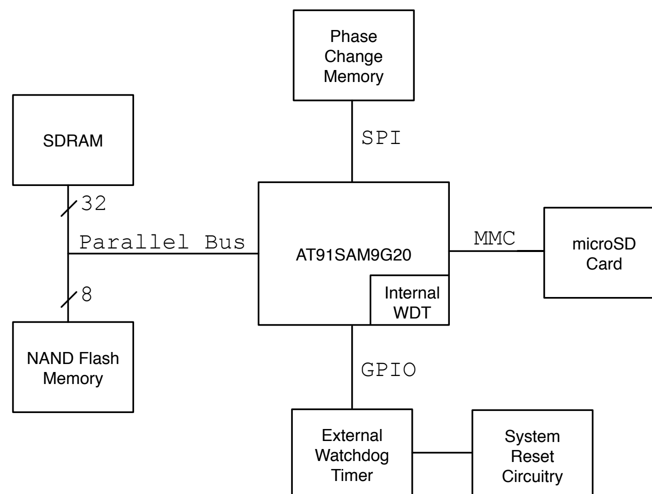


Figure 3.1: Second Generation Hardware Block Diagram - This figure shows the new processor, memory, and hardware watchdog configuration. The memories are on different types of buses: SDRAM and NAND Flash are on a high-speed parallel bus, phase change memory on a SPI bus, and the micro-SD card is connected via the MMC bus.

The microprocessor was chosen because a port of the Linux operating system already existed for it, which was sought out due to the the wide variety of open source software and extensive community support that is available for free. As a result of choosing Linux for the base software operational environment, the

software produced from this thesis exists primarily as libraries and processes that run in user space.

The system requirements cover the non-functional characteristics of the software architecture, and they are itemized in Table 3.1. The first set of system requirements deal with the performance as it relates to time and space. Fortunately, none of the functional requirements of the system necessitate hard real-time deadline capabilities because the periodic functions of the satellite, such as the beacon, do not require high temporal accuracy to be useful. The alternative to this is being able to meet soft deadlines, which do not require strict timing to be correct and appropriately has been chosen as the minimum time performance requirement.

In addition to performance, there are also some space requirements for the code and data storage on the satellite. Phase change memory (PCM) technology is multiple orders of magnitude more radiation tolerant than NAND Flash [10], thus the boot code and related software must require less than 16MB of storage so that it does not exceed the total storage available in PCM. The remainder of the flight software must fit in the 512MB of NAND flash. The entire amount of microSD card storage can be used for data storage.

The next group of system requirements describe the general characteristics. The software should maximize the uptime of the system, so that a link with the satellite can be established whenever it passes over a ground station that is attempting contact. Furthermore, the system should be robust to the common radiation events so that it is capable of providing this reliability guarantee. The primary issue that can prevent operational uptime is single event upsets (SEUs) and single event latch-ups (SELs), which are due to ionizing radiation in space; thus, the system must be able to detect and react to these events. SEUs can be recovered from by refreshing the affected memory bit, but SELs require power to be removed from the component to allow enough time for the power plane of the integrated circuit to return to ground [11].

Another group of requirements deal with the correctness guarantee that the system can offer. For this system, only the intended commands should be executed in order to avoid putting the spacecraft into an unintended state. Each software module must also be tested and validated to guarantee that it executes the correct operations upon receiving a command. The next requirement is also

Requirement Group	#	Name	Details
Operational Environment	1.1.1	Hardware	400MHz AT91SAM9G20, 64MB RAM
	1.1.2	Software	Linux compatible
Performance	1.2.1	Time	Soft real-time deadlines
	1.2.2	Space	16MB boot s/w, 512MB other file system data
General Characteristics	1.3.1	Reliability	Maximize uplink time
	1.3.2	Robustness	Autonomous recovery from SEUs and SELs
	1.3.4	Correctness	Only intended commands will be executed
	1.3.5	Security	Other groundstation commands limited
	1.3.6	Extensibility	Simple for future developers to add/modify
	1.3.7	Power	Minimize power consumption when inactive

Table 3.1: Non-Functional System Requirements

related to commands: the system must be secure enough to ensure commands that have been corrupted by SEUs or have been sent from unauthorized ground stations will not be executed.

The last two general system requirements are extensibility and power versus performance. Extensibility is a very important aspect of the software system so that the system can adapt as the needs of PolySat change based on the missions. Thus, it must be straightforward to add new modules and features. As for power versus performance, the system has very few compute bound aspects so performance is not typically an issue, but power definitely can be, since the typical one unit CubeSat has a power budget of about 1.5 watts per orbit. The software needs to be aware of the power state of the system and do what it can to regulate power consumption when necessary.

The process requirements specify those related to the development on the platform and are shown in Table 3.2. For overall development on the system, the goal is to enable a 6 month turn around time for new missions. If the system is appropriately constructed, the developer should spend the majority of the time working on the payload software rather than modifying base system code to support it. Furthermore, since the developers are students, all development tools and environments should be free so that there is no base cost to the organization or the developer. This requirement is met fairly easily thanks to the numerous

#	Name	Details
2.1	Development Time	Enable 6 month mission turn around
2.2	Development Cost	Free development tools/environment
2.3	Standards Conformance	PolySat coding standards
2.4	Testing and Validation	Black-box demo & reviewed by colleagues

Table 3.2: Non-Functional Process Requirements

free tools available for development on the Linux operating system.

The last process requirements specify the acceptance procedure for flight code. First and foremost, all code should be written following the pre-exiting PolySat coding standard (Appendix A), which enables Doxygen generated documentation and will result in a consistent coding style. The developer is responsible for validating any internal functionality to him or herself, but upon completion two different events must occur: a functional demonstration to a senior member of the PolySat software team and a formal code review. After the functionality of the code has been verified, the developer should ensure the code is up to the coding standards of requirement 2.3, and execute a formal code review with a few members of the software team. There is no required experience level of the reviewers because it is a mostly an aesthetic review to insure the code meets the coding standards. This code review format also existed prior to the development of this software architecture and the entire review process is explained in detail in Appendix B.

The last set of requirements deal with those related to personnel and are summarized in Table 3.3. These requirements are relatively unique to the PolySat organization because all of our developers are students who are mostly volunteers and work on the project in addition to their coursework. Due to this educational environment that does not have a strict time commitment, the requirements for developers are fairly limited and basic. Specifically, there should not be any major credentials required to be a developer on this system other than being at least a college student. Additionally, minimal certification should be required to develop on the platform. It will be a loose requirement (or more of a strong suggestion) that students have taken and passed the introductory systems programming course, so that they have an existing knowledge of programming for Linux or a

Requirement Group	**#**	Name	**Details**
For Developers	3.1.1	Credentials	Any college-level developer
	3.1.2	Licensing/Certification	Intro. to Systems Programming

Table 3.3: Non-Functional Personnel Requirements

similar operating system. The class CPE357, "Systems Programming", is taken by all computer engineering, computer science, and software engineering students early in their Cal Poly career, and should be sufficient experience to develop for the new software architecture. Thanks to courses like these, the learning curve for satellite software development is significantly reduced because most students will join the organization with some related experience.

3.3 New Design Philosophies

In addition to the requirements that bound the design, a number of design philosophies were utilized that were not emphasized in the previous design. First and foremost, a strong emphasis is placed on software modularity for this design to isolate components and support extensibility. It also supports the next design principle of encouraging parallel development, which can greatly reduce development time. Furthermore, now that the Linux operating system is being used, there is far more pre-existing code that could potentially be useful for the project and we hope to leverage as much of it as possible.

3.3.1 Software Modularity

As defined by the *Linux Dictionary*, software modularity is: "A programming style that breaks down program functions into modules, each of which accomplishes one function and contains all the source code and variables needed to accomplish that function" [12]. This style of programming significantly simplifies the maintenance and debugging of the system because individual modules can be isolated for revision or testing without affecting the entire system. In a large, modular software system finding the source of problems can be easier as well because the problematic function should be able to be isolated into a minimal number of modules. A consistent usage of this this philosophy will enable the

new PolySat system to be easy to revise and thus encourage its continued use in future projects.

The emphasis on software modularity is also to encourage the reduction of hardware modularity in CubeSat projects. Hardware modularity is a common design philosophy in CubeSats, likely because it gives the appearance of a clean design from the perspective of the systems and electrical engineers. However, since CubeSats are highly volume constrained platforms, hardware modularity can be very expensive due to the typical volume requirements. Emphasizing software modularity can help reduce the volume requirement of the avionics system by implementing some of the modules that are traditionally implemented in hardware, in software.

Favoring software modularity over hardware modularity also reduces the complexity of the system because the number of custom interfaces is reduced. Typically, in order to interface with different hardware modules, custom software interfaces must be written. Furthermore, these custom interfaces are code that is rarely reusable and exist for no other purpose than to support the hardware modularity. In some cases, these modules may only be used for some missions and thus a significant amount of effort may be invested in code that neither directly relates to the mission nor will be used in future missions.

PolySat experienced this first hand with the previous design, where the concept of hardware modularity was used to separate the communications and C&DH components. The correct operation of this interface was critical to the system because it was the path to and from the transceiver. A custom driver was required for this interface and was even upgraded in later revisions of the system to include additional robustness (see Section 2.1.4). Unfortunately, this interface was very specific to the microcontrollers that were being used and thus the code could not be reused if similar interfaces were desired with different processors. Furthermore, this interface was hypothesized to be the root of the cause for both the failures of CP4 and CP6, where the C&DH could no longer communicate with the ground (further explained in Section 2.2). These events are a driving force to this design philosophy: by favoring software modularity we can limit the number of custom hardware interfaces, maximize code reusability between unique missions, and potentially increase reliability.

3.3.2 Encouragement of Parallel Development

PolySat's team of software developers has grown significantly since the first generation system was developed; while it once was comprised of two or three core members, there may now be more than five people working on a single project. Most of the software development on the prior system was needed to be completed at a bench in the lab to be able to load the software onto the microcontroller for testing. As a result the number of simultaneous developers was limited by access to the lab benches. However, with the selection of Linux as the new operating system, these ties to the bench have been severed because a significant amount of code can be compiled and executed in any Linux environment. Therefore, a very important design philosophy for this new software architecture is that parallel development should be both enabled and actively encouraged.

This concept directly supports the development time requirement, 2.1, which states the system should enable a short turn around for new missions. When more students that can work on the project at the same time, the software should be produced at a faster rate.

3.3.3 Pre-Existing Code

The total user base of Unix-like operating systems is considerable and as a result, a large amount of code has been written for it and is available for free, under an open source license. Linux supports a significant amount of this code thanks to its support of various open standards, such as POSIX. Utilizing these packages will not only reduce development time by reducing the amount of code that needs to be written, but also potentially improve reliability since the packages are in use in other projects. OpenSSL is an excellent example of this, which was first released in 1998, but is still actively being developed for and used in a wide variety of applications [13].

It also is worth acknowledging that the operating system and standard C libraries provide a non-trivial amount of pre-existing code. This is a significant source of added reliability due to the prevalent use of the Linux operating system and also greatly reduces development time. Throughout the new design, there are examples of leveraging Linux-provided interfaces, from the fundamental file abstraction system calls up to the inter-process communication mechanisms heavily

leveraging the provided socket APIs.

Pre-existing code from other sources is also used when possible to aid the development process. This decision has been paramount to being able to develop a prototype of the software platform in a reasonable amount of time, in addition to inspiring confidence in our customers by being able to specify that this code is used in other, pre-existing platforms. The OpenSSL library, as mentioned previously, provides all of the core functionality for the cryptography component of the new architecture (Section 5.1). Source code was also taken directly from other open source projects, such as the priority queue heap implementation in the event handler (Section 4.3.1). These particular examples have been vital to the success of the design because they provide features that would have been very difficult and time-consuming to develop from scratch.

4

New System Architecture

The new software system's architecture looks far different than that of the previous generation's because of the emphasis on software modularity and the options afforded to us by the utilization of Linux. The system is now comprised of two primary layers: processes and libraries. The processes are responsible for the actual functionality of the satellite, and the libraries provide middleware-type services to the processes. This type of architecture improves upon the previous because it allows features and functionality to be added and upgraded without having to recompile or rewrite other modules.

4.1 Software Overview

The software architecture has been fundamentally re-designed as a result of the choice to utilize a new operating system. Linux provides a simple mechanism for address space isolation and protection, the process. This is important because it provides a natural level of granularity for modules. Further, processes simplify parallel development since each process can be developed as a black box and without understanding the entire system, as was needed with the first generation software architecture. Thus, processes have been utilized to implement all of the basic functions of the system and provide one of the major levels of software modularity in the system.

The system processes have been defined as shown in Figure 4.1. Each of these processes are defined in Section 4.2. These processes utilize many common mechanisms and functions, which have been designed into a set of shared libraries that

are discussed in Section 4.3. The last defining characteristic of this architecture is that there are two different modes of operation, the selection of which is based on whether or not the NAND Flash is functioning properly.

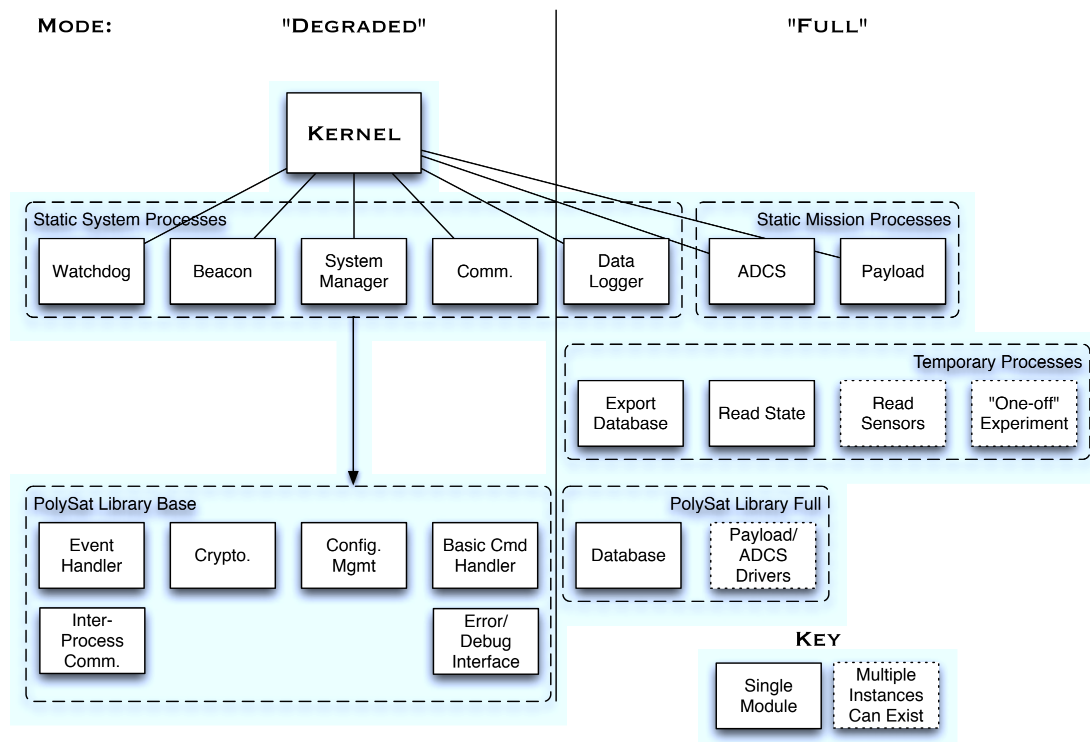


Figure 4.1: Software Architecture - This figure shows all of the software modules, separated by minimum mode of operation where they are available. The satellite is always at least in degraded mode, which occurs when the satellite first boots. Full functionality mode is entered after the NAND flash has been verified as functional.

Some of the other new features were not enabled by just Linux itself, but also the open source packages that are available for it. Specifically, a SQLite database is used for storing telemetry and the OpenSSL library is utilized for its various security and fault tolerant features. These packages meet both the design philosophy of leveraging pre-existing code and numerous requirements regarding capabilities, correctness, and security.

4.1.1 Operation Modes

Two basic modes of operation that dictate the capabilities of the system have been chosen based on the effects that radiation have on the different memories. The phase change memory (PCM) has been shown to be tolerant up to 30 MRad [10], which is orders of magnitude greater than that of the NAND Flash. To maximize the utility of this phase change memory part, the minimum amount of functionality required to communicate with the satellite and execute basic commands will be stored on the PCM part to avoid corruption from radiation and enable a degraded mode of operation. The satellite will enter this mode by default until the the integrity of each file on the NAND flash has been confirmed.

Degraded mode provides the basic processes: beacon, system manager, communications, and the software watchdog. These processes are sufficient to allow the ground station operator to interact with and diagnose the spacecraft. The drivers available are limited to those related to the vital subsystems, such as the transceiver and processor related sensors. Lastly, since the state of the NAND flash is unknown in this mode, access to it is not enabled. Work is being done to enable the upload of full kernel and filesystem images so that the spacecraft could potentially recover if the spacecraft is stuck in state.

Full functionality mode is entered after the NAND flash has been verified and enables all of the features of the spacecraft, which includes the data logger process and when necessary, mission specific processes. In this mode of operation, the data logger (Section 4.2.4) is autonomously collecting and storing telemetry. To support this, additional library functionality is also provided in this mode (discussed in Section 4.3), as well as the drivers for the rest of the components on the satellite. The satellite is expected to spend a majority of its time in this mode.

4.1.2 Inter-Process Communications

It is important that processes can communicate with one another in a straightforward manner. This is another opportunity to leverage significant pre-existing code because the Linux kernel provides a number of different options for inter-process communication, such as TCP/IP, UDP/IP, and Unix Domain Sockets. The latter is intended only for inter-process communications, while the two for-

mer also enable networked communications.

The first instinct was to use the simplest, lowest overhead option, Unix Domain Sockets. Unfortunately, upon further investigation of this option, it was realized that an additional packet format would need to be designed on top of these messages in order to support this. Some of the notable desired features included checksums and numeric identification of processes. The first design had a custom module for receiving packets from the transceiver and converting them into the alternate packet format and there was also a significant amount of custom code required for both sending and receiving these types of messages.

Quickly, we arrived at the realization that we were redesigning the IP stack and since the intention was to configure the Unix Domain Sockets as datagram packets, the existing UDP/IP stack could be utilized and require far less custom code to support our desired features. Specifically, UDP uses ports for destination addresses and there are checksums in both the IP and UDP layers that are handled by the kernel. Furthermore, by using an IP based protocol, it will be simple in the future to support standard IP-based applications such as Telnet and Secure Shell (SSH). Using applications like these can be immensely helpful for debugging the system because it would enable command line access from the ground.

Utilizing UDP enables a number of things that simplify the design of the system. Standard Linux systems utilize the file `/etc/services` to match service names to ports, and vice versa; an API is provided to accomplish this, within the standard libraries. The PolySat system has a number of pre-defined services, in the form of the processes, and thus fits well with this system. All of the processes will be added to the file with a port number, so that in order to communicate with a particular process, the only requirement is to know the destination process' name.

4.2 Processes

The processes are responsible for the core actions of the satellite. They have been selected such that each primary function is implemented in its own process. Processes have the distinct advantage over threads by providing a separate process address space for each and thus protecting the modules from each other's memory corruption. There are two different types of processes: static and temporary. The

primary difference, other than the duration, is that static processes are based around the custom PolySat libraries (Section 4.3) and are written in C; whereas the temporary processes can be developed in any Linux-supported language (see Sections 4.2.4 and 4.2.7).

There are a couple of shared high-level commands for each static process, in order to facilitate different aspects of the satellite and also increase the amount of reused code. Rather than have each process implement its own system information status command, one is implemented in the standard library that provides all of the memory and CPU usage information, since the acquisition of this information is the same regardless of the specific process. The second command is the generic status command, which provides any soft state related to the specific process. Soft state information includes any values related to the operations of the process, such as a count of an action specific to that process. This command is utilized in a variety of ways, but generally it ensures that the useful state values of a process can be queried from both the ground and other processes to facilitate debugging and telemetry gathering.

4.2.1 Beacon

The Beacon module is implemented as a static process and is responsible for one distinct function: periodically initiating the broadcasting of key satellite telemetry.

This is very important to all spacecraft because it provides a mechanism for tracking and identifying the CubeSat when it is ejected from the P-POD. When the satellites are first tracked in orbit, the ones that came out from the same P-POD or even sometimes the same launch vehicle can be very close together. When this happens, it's difficult to identify exactly where each spacecraft is and thus a lot of time is spent aiming antennas at the different objects in order to determine the spacecraft's location. In order to locate a satellite without a beacon, commands must be sent to the satellite in hopes that a response is issued. However, CubeSats are not known for their receive sensitivity (see Cal Poly's history in Section 2.2), so this method is very difficult since a response may not be elicited even if the antennas are aimed at the correct spacecraft.

The solution to this is to have the satellite configured to be transmitting periodically so that the ground stations only have to track each object as long

as the period of this transmission. This resulting periodic transmission is the beacon.

The beacon is also useful for information gathering via amateur radio stations around the world. The CP6 satellite was a very successful demonstration of this capability: a limited packet decoder was distributed to amateur radio operators around the world, which enabled them to downlink packets and then submit them to a database. This information primarily consisted of beacon packets and totaled to a larger amount of data than was collected at Cal Poly's own earth station! Based on the success of this strategy, PolySat hopes to provide similar software packages for each satellite and also ensure that the beacon has a large amount of useful information in order to take advantage of this global data collection method.

The beacon is one of the simplest processes in the system and only has one unique command, `Change Rate`, which has one argument: the new desired rate for the beacon.

The specific format of the beacon is rigid per mission, due to the ground station requiring a precisely known packet format for every downlinked packet. This unfortunately typically changes for each mission because there is frequently payload-specific data in the beacon. However, in degraded mode of operation, there are no payload processes, so the standard beacon is constructed by querying the other active processes status and putting the responses into a packet.

4.2.2 Communication

The communication module is essentially a software modularity answer to what was previously implemented through hardware modularity (see First Generation Design, Section 2.1.2) with a PIC microcontroller. At a high level, the software module is responsible for the same thing as the microcontroller was: encoding and decoding packets that are going to or coming from the transceiver, respectively. It also manages the control and configuration of the transceiver hardware, so it is responsible for both the data link (L2) and physical (L1) layers of satellite communications.

Unfortunately, both layer schemes are dictated (and limited) by the PolySat ground station support because significant dependencies exist in both the hardware and software infrastructure for communicating with the satellite while it is

in orbit. All previous missions have utilized the AX.25 packet format [5], and that is what will be initially implemented for this system, as well, to be compatible with the existing ground station. The primary issue with this is that the AX.25 protocol has a relatively high overhead (14 bytes per 256 byte packet), so the implementation needs to be flexible enough to allow for potential upgrades in the future with a more minimal protocol.

The communication process is responsible for all of the low level network management, from configuring the transmission parameters in the transceiver up to medium access control. Before any communication can occur, the transceiver must be configured to input and output the anticipated type of radio frequency signal. Once the transceiver is initialized, packets can be received or transmitted by the radio. Also, since our current hardware design only enables half-duplex communication, meaning you can only transmit or receive at any given time, the communication process may need to do medium access control to avoid collisions. The best example of this would be when the transceiver begins receiving a packet, the communication process will not send any packets for transmission until the packet has been completely received, or after a timeout that would indicate a partial packet has been received.

The next challenge for the communications process is the routing of packets to and from other processes within the satellite. This is another situation where reliability is increased and development time was decreased, by leveraging the pre-existing network stack within the Linux kernel. The `tun/tap` interface has been used to inject raw IP packets into the kernel after they have been received, and to acquire packets that have been sent from other processes that are intended to be transmitted. This solution has greatly simplified the communication process design.

4.2.3 System Manager

The system manager process is primarily responsible for maintaining the state of the avionics system, both hard and soft state. This includes all of the different kernel statistics that are available through the `/proc/stat/` interface, basic hardware state information (temperature and power consumption of the processor), and side panel sensor data. Additionally, the system manager enables a variety of administration actions through simple commands, such as `Kill Process` and

`Reboot System`.

These capabilities are grouped together to provide a focal point of access for all of the state information of the satellite. If the database were the single point of access for this information, it would be impossible to acquire in the degraded mode from the ground. This is not acceptable because it is very important that this state information and the system administration capabilities are available in all modes of operation because it is vital to diagnosing and potentially solving problems.

4.2.4 Data Logger

There are a number of events in the spacecraft that occur at a regular frequency throughout the entirety of the mission, such as telemetry storage. Also, it is common that payloads would like to run one-off experiments that return some type of data, upon completion, which needs to be stored in non-volatile memory. The shared characteristics of these events are utilized in the data logger process, which spawns temporary processes that collect and output the resulting data. This data is then captured by the data logger and stored into the database. Unfortunately, as a result of this dependency to the database, the data logger cannot run in degraded mode.

The chosen output format from spawned processes is key value pairs, printed to standard out, because it enables the temporary processes (see Section 4.2.7) to be developed in whatever language is convenient. The key value pairs will be parsed by the data logger, the key looked up in the database, and then the value stored with that key. The database does not have to have any key-specific code as long as the string is unique, which it should be.

Another capability of the data logger includes being able to schedule experiments for some time (an offset or UTC time) in the future and repeat experiments. An experiment will exist in the form of a temporary process that is executed at the scheduled time, which could be a binary, or merely a script. This type of scheduling request is very common for state-less payloads, such as CP7 [9], and thus eliminates the requirement of a process devoted to this common type of payload.

There are also some limitations to the data logger, particularly related to the rate of spawning processes. The selected temporary events should not exceed a

rate such that it costs more power and CPU time to fork off the process than it would to just add a static process. Processes that require high rates of telemetry collection or running may fall under the category of mission specific, which are described in Section 4.2.6.

The data logger is being developed as a senior project and a detailed design document should be published as a result, approximately at the same time as this thesis.

4.2.5 Watchdog

The software watchdog process is one of the cornerstones of the software fault tolerance of the system and is described in detail in Section 5.2. It watches all of the processes in the system and attempts to validate that they are operating as intended, periodically. In order to accomplish this, a query is regularly issued to all of the processes for their status and then the response (or lack thereof) is analyzed to verify that the process is operating correctly. The software watchdog is also responsible for tapping both the internal and external watchdogs. The last responsibility of this process is to validate configuration changes of other processes during run-time, to prevent the inquiring process from using a corrupted value.

4.2.6 Mission Specific Static Processes

The other potential static processes are those specifically related to the mission, most likely in the form of ADCS or payload control. ADCS is good candidate for a static process because often relatively high frequency sensor readings are required. For example, CP6 demonstrated an implementation of the ADCS algorithm, B-dot, which needs sensor updates at approximately 10 Hz. The overhead of spawning a process 10 times a second with data logger would exceed that of just using a static process, and thus it is a good candidate. Payloads that require this type of control would also be eligible for a static process.

4.2.7 Temporary Processes

Most missions will likely fly a number of temporary processes because of the simple interface to the database through the data logger. Also, thanks to the wide variety of languages that have Linux support, these temporary processes

can be written in whatever language the developer is most comfortable in and supports the timeline. Using scripting languages like Python can significantly improve development time for new devices and payloads.

Some temporary processes have already been designed into the system, as shown in the software architecture diagram (Figure 4.1). The data export process is currently being written so that a compressed version of the database can be downlinked and maintained on the ground. The read state process will be used to collect the state information from the individual processes, for storage in the database. Other temporary processes are likely to be developed, such as the sensor readings or "one-off" experiments, which will likely be written on a hardware and mission-specific basis.

4.3 Abstraction Libraries

A standard set of libraries have been designed and implemented to provide a standard set of features for the static processes described in Section 4.2. These libraries facilitate the development of the processes and also encourage code reuse with features like time and file event handling, configuration files, and inter-process communication command handling. The library is compiled as a shared-object to simplify version management; once the library is installed, the same version will be used by every process running on that system.

One of the considerable non-functional advantages of using these libraries is that developers can also compile them for their personal machines, if they are running a Unix-type operating system (including popular choices such as, but not limited to OS X and Ubuntu). This is very much in line with the parallel development philosophy because process developers are not limited to using the custom hardware, but can now develop on personal computers.

Similar to the modes of operation, the libraries are separated into two distinct categories: base and full functionality. The base library contains all of the components necessary for running the degraded mode processes and has a minimal footprint, whereas the full functionality library adds additional features that are too large in size to fit on the phase change memory or require access to the NAND flash, such as the database, which is stored there.

4.3.1 Event Handler

The satellite is an event-driven system by nature and thus an event handler was designed to be at the core of every process. In the satellite, there are generally two types of events that occur: timed and command-initiated. Since this system utilizes UDP for all inter-process communication, the only way a command can be received is via the process's socket. This fact can be exploited to develop an event handler based around the Linux system call, `select`, which is easily configured to watch file descriptors for data available, as well as wait for a certain amount of time, for timed events.

The general design for the event handler will cause the process to block until either a timed event is ready to execute, or a file descriptor event is ready (typically a read or write). The process can add any type of timed or file-descriptor based event with a corresponding callback function, so the event handler can be customized to the process's needs. It is important that within these callback functions no blocking function calls are used because that would significantly delay the amount of time until the process returned to the event loop and thus potentially delay registered timed events. A simple state diagram for this system is shown in Figure 4.2.

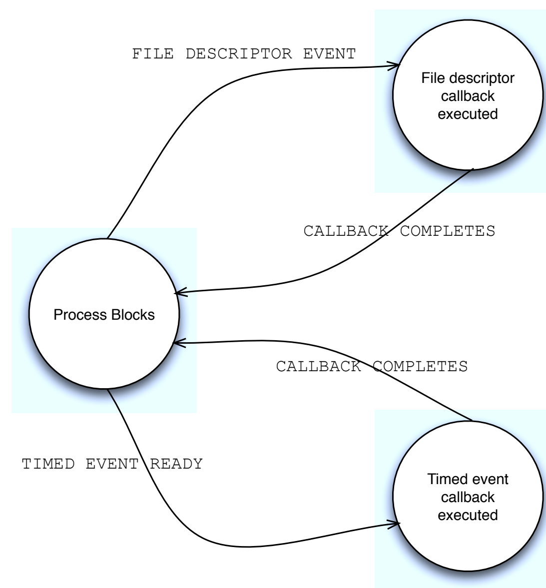


Figure 4.2: Event Handler State Machine - The figure shows the basic states of the event handler.

4.3.2 Command Handler

All of the commands for a process will be received on its well-known UDP port and a function is typically assigned to each different command (a process designer could choose to have two commands map to the same function). Rather than requiring each process to (re)implement a command handler to parse each command, a standard one has been designed utilizing the first byte of the incoming message to determine the function to call.

Since the first byte of the incoming message has been designated the command byte, it can be used as an index into an array of function pointers. This array of function pointers is constructed in the process initialization sequence (see Section 4.3.7) based on a simple command configuration file that matches the function name to the command value. Each function pointer in the array is initialized to point to a function that will reply with a negative acknowledgment (NACK), in order to handle incorrect commands. Issuing a NACK enables the ground station operator to know that the packet was received by the satellite and be alerted an invalid command was issued.

The command handler also utilizes the cryptography aspect of the library (briefly described in Section 4.3.6 and in detail, Section 5.1) in order to validate protected commands before allowing the process to handle it. If a command is protected, that means it is required to be signed and the source of the command must have the appropriate permissions to execute it. If the command fails to pass these checks, the corresponding function is not called. The basic procedure of receiving a command is shown in a flow diagram in Figure 4.3.

4.3.3 Inter-Process Communication Abstraction

The inter-process communication (IPC) component of the library provides a simplified abstraction of the existing Linux API for creating and using non-blocking UDP sockets. These abstractions are directly utilized by the command handler and other aspects of the library to receive and send commands, but can be used in any process to communicate on any UDP port. This enables processes to utilize additional channels of communication, if necessary, without having to implement the low level socket interactions.

The IPC component also provides process name lookup abstractions, so that

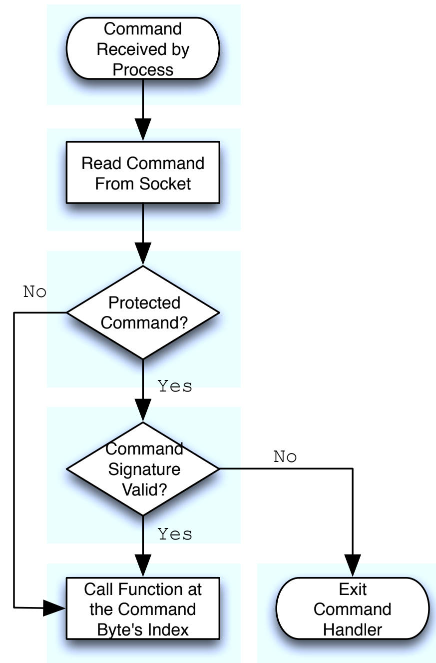


Figure 4.3: Command Handler Flow Diagram - Software flow diagram for the command handler.

the port of another process can be looked up in the `/etc/services` file. A function for the reverse is also provided: a name can be used to acquire a socket address structure with the port. The socket address structure can then be used to send messages to the process with that name.

4.3.4 Configuration Management

Another feature enabled by the library is utilization of configuration files. This allows the process to have various pre-configured states, as well as simplify testing on the target platform by enabling the configuration without having to recompile and transfer the new process binary to the board.

The actual implementation for this system enables a variety of variable types, such as integers, strings, and internet protocol addresses. These variables can also be placed in objects, with syntax akin to XML, which produce a single structure (a struct, in C) in the code. In the source code that is using a configuration file, the expected variables are pre-defined, so that they can appear in any order and in arrays within the configuration file itself. An example command configuration

```
# Sample Beacon Command Configuration File

<CMD>
    PROC=BEACON
    NAME=STATUS
    FUNC=beacon_status
    NUM=1
    CMDOID=10
    GRPOID=30
    PROTECTION=0
</CMD>
```

Figure 4.4: Example Configuration File - An example configuration file; this one shows a portion of the command configuration file for the Beacon process.

file for the Beacon process is shown in Figure 4.4.

4.3.5 Standard Debug/Error Interface

One of the more important standard practices for software development is a unified debugging and error logging interface. Linux already provides an error output stream, `stderr`, which can be utilized for debugging during development when the target board is connected to a terminal output. However, output to the terminal is not very useful while the satellite is in space and thus some type of logging capability is necessary.

Fortunately, once again Linux already has this capability built-in with the system logger, `syslog`. Thus, the debug and error logging interface provided for processes will print to both `stderr` and the `syslog` so that debugging can be done on the ground and in space. Using both is important so that the code that will be utilized in space is also ran on the ground for testing purposes. It is easy to make the mistake of ignoring a space-only feature and it is hoped to increase the testing of flight functionality throughout testing.

Other Linux features were leveraged for this debug interface, as well. Linux system calls and a number of standard library functions utilize a standard error reporting global variable, `errno`. When an error occurs in one of these functions,

it gets set to a number that corresponds to a type of error, which also has an associated error message. As part of this debug interface, these error messages are printed and logged via the use of a simple macro that wraps the function call.

4.3.6 Cryptography

The cryptography portion of this library provides a simplified interface to particular aspects of the OpenSSL library. There are a few primary functions provided from this portion of the API: data hashing, message signing, and message validation. The hashing provides a secure hashing capability for use wherever data integrity or security is necessary. This is also used by message validation and signing, which is used for ensuring critical commands are received by the satellite uncorrupted. The justification and implementation details are provided in Section 5.1.

4.3.7 PolySat Library Base

All of the above components are wrapped into single basic library component and their capabilities are available in all modes of operation. A few higher-level functions, which utilize a number of components, are provided to further simplify the usage of the library features.

When each process starts it will call an initialization function, which obtains and initializes all of the important aspects of the process. A flow diagram of this is shown in Figure 4.5. This includes creating the event handler object, opening a socket bound to the process's port number, and registering with the watchdog. It also creates a file based on the processes name with the .pid extension that contains its process ID, and another file in a separate location based on the process id with the .proc extension that contains the process name. These files are used by the software watchdog (see Section 5.2.4). This initialization will insure that all absolutely necessary actions that are common among all static processes are completed. A process cleanup function is also provided, which frees the memory allocated and removes the files created during the initialization sequence.

In addition to the initialization and cleanup functions, a non-blocking IPC function is provided. This is not implemented within the IPC component of the library because it requires utilization of the event handler. The event handler is

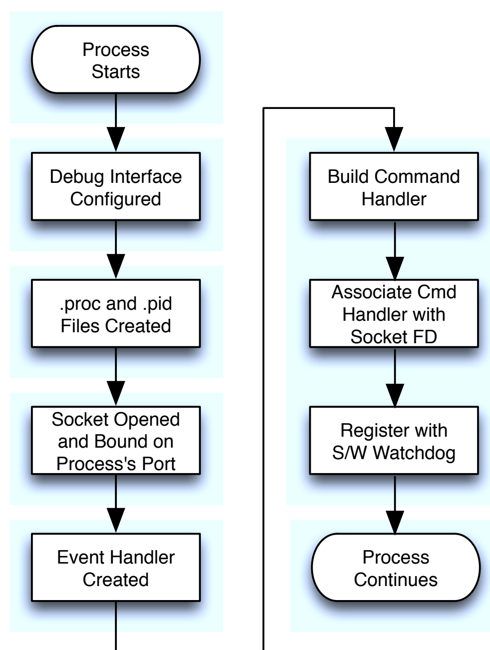


Figure 4.5: Process Initialize Flow Diagram - Software flow diagram for the process initialization sequence.

used to schedule the write operation when it will not block. This is very important to insure the correct operations of the process because any blocking operations could significantly delay the process' return to the `select` call within the event handler and result in missed deadlines.

These high level command interfaces provide the process with a choice to send a message to another process one of two ways: by name or address. When a message is sent by name, the IPC lookup function is used to acquire the socket address. In order to avoid having to do this lookup every time, the option is also provided to use the standard socket address structure. This structure can be acquired by completing the lookup at the beginning of the process, to avoid requiring it each time. This may be used if a process intends to communicate regularly with another process to reduce message latencies (demonstrated in Results, Section 4.4).

4.3.8 Database

One of the initial challenges of the new system design was determining how to store all of the telemetry and experiment data. In the previous system, there were very few options due to limited storage and code space. However, with the relatively large amount of storage available in the new system, the number of options grew significantly. The first instinct was to utilize a database of some kind because it is a typical option for data storage in most computing systems and there is also a course at Cal Poly that teaches database systems. The choice was made fairly easy when looking at the different Linux compatible solutions and it was found that the open source package SQLite was very prevalent in similar embedded situations, such as a variety of modern mobile devices [14]. Selecting this package is in line with our intentions of leveraging pre-existing code and minimizing custom code because SQLite provides a simple C API that requires just a wrapper to be developed to enable the use of a database within our system.

In order to allow a large amount of telemetry in the database, the raw database is to be stored on the NAND flash non-volatile memory. However, as a result of this decision, the database API can only be used in full operations mode since there is no read/write access to the NAND flash part of the file system in degraded functionality.

4.4 Results

In order to assess the ability of the system to meet some of the non-functional system requirements (see Table 3.1), such as time performance (1.2.1) and footprint (1.2.2), some of the completed components of the system have been characterized. All tests have been run on the target platform hardware and timing tests have utilized the internal time keeping functionality of the Linux kernel.

4.4.1 Memory Footprints

The memory footprints of the components are important for two reasons: there is minimal non-volatile storage available for degraded mode of the spacecraft and reducing the volatile memory footprint reduces the power consumption. The storage requirements of the have been summarized in Table 4.1 and the volatile

memory requirements in Table 4.2.

Aspect	Component	Stored Size
Libraries	PolySat Base	69 KB
	OpenSSL	1.56 MB
	SQLite	451.1 KB
Processes	Beacon	11.5 KB
	Watchdog	18.2 KB

Table 4.1: Various Component's Non-Volatile Storage Requirements - These are the size of some of the static processes and library binaries when compiled for the processor.

The base custom library (Section 4.3) itself has been shown to have a very small footprint of only 69KB, while the processes themselves are even smaller, with the beacon (Section 4.2.1) at 11.5KB and a more complex process, the software watchdog (Section 4.2.5) requires 18.2KB. These file sizes are when compiled without debugging symbols, since they are unnecessary in flight software. The largest components of the system are by far the other open source libraries that are utilized, with OpenSSL weighing in at 1.56MB and SQLite at 451.1KB.

Both processes measure required just over two megabytes of RAM while running, however, it is important to note that most of the memory is used the libraries, which accounts for 1840 KB or nearly 80% of the memory needed for each process. This information was collected by using the `/proc/<pid>/status` interface while the process was running.

Process	Run-Time Memory
Beacon	2312 KB
Watchdog	2416 KB

Table 4.2: Static Process Runtime Memory Requirements - This table shows the total virtual memory allocated for these processes. This includes the memory required for utilizing both the custom and standard libraries, which accounts for 1840 KB.

Event	Average Time
Static Process Initialization (Default)	15.2 ms
Static Process Initialization (Improved)	9.1 ms
UDP Message (Process Name – Default)	16.2 ms
UDP Message (Process Name – Improved)	10.0 ms
UDP Message (Socket Address)	7.2 ms

Table 4.3: Latency of Common Library Events - The default latencies were achieved by adding the process port numbers to the end of `/etc/service` file, whereas the improved time is a result of relocating them to the beginning of the file.

4.4.2 Common Library Event Latencies

Some of the different key actions in the system have been timed and are shown in Table 4.3. The presented results are an average of numerous attempts, with a single process running in the system. Although the actual system will have multiple processes executing in the system, at any given time, it is more likely for them to be blocking than actively running due to the periodic nature of their actions and the relatively long time between events (e.g. a typical beacon rate is no faster than 30 seconds), meaning events are unlikely to be compute bound.

The process initialization step gives an idea of how long it takes the custom library to generate all of the standard objects that each static process uses (see Section 4.3 for more information on this sequence). Since the last step of initialization is to tap the watchdog, it is important this time is minimized. Based on the measurements taken, this sequence takes approximately 15.2 ms on average. At worst, this means it would take a five static processes approximately 76 ms to all boot up. This is an acceptable latency because it is far less than 16 second internal watchdog timeout, meaning the software watchdog can safely wait for all processes to boot up and register with it before sending a heartbeat signal.

The next measurements are the end-to-end latency of sending a message from one process to another, a number of components are used from the library for this to occur: the IPC abstraction sends the message, the event handler recognizes there is data available on the socket, and then the command handler reads the data and finally calls the corresponding function. The simplest interface is the one where the destination process name is provided, however, a significant amount of

Event	Period	Average Absolute Error	% Absolute Error
Beacon	5s	4.98ms	.10%
Watchdog Status Query	10s	1.87ms	.02%

Table 4.4: Scheduler Accuracy Measurements - These results show the measured error of the event handler's scheduler with two processes actively running in the system.

time can be attributed to the lookup of the port name in the `/etc/services` file. This is demonstrated by the results with the socket address, which only take 7.2 ms on average.

The IPC named port lookup function (Section 4.3.3) is used in the initialization sequence to lookup the port for the process, so a slight tweak has been applied to the `/etc/services` file to improve performance. The lookup performed in the file is evidently approximately linear because when placing the process name at the beginning of the list, both the messaging and initialization times improve significantly. The average latencies have been reduced by at least 5 ms; improved messaging is now 10 ms and the improved initialization time has been reduced to almost 9 ms.

4.4.3 Scheduler Accuracy

The accuracy of the event scheduler (Section 4.3.1) has also been analyzed to provide an idea of how accurate the deadlines it can provide are, these results are summarized in the Table 4.4. The configuration was chosen to simulate a potential satellite operations environment. A the beacon and watchdog process was running, with a 5 second periodic "beacon" event and the watchdog collecting statuses every 10 seconds. The accuracy of all of these three timed events has been collected by analyzing data over a runtime of approximately 450 seconds, with minimal other activity in the system.

These events are the primary scheduled events in degraded mode of the spacecraft, but are running at a rate far higher than would be typical in flight. In spite of this, the beacon is still being executed at less than a tenth of a percentage of deviation from the desired rate; whereas the watchdog query event is even more accurate, with closer to one hundredth of a percentage of deviation. Even with both processes running simultaneously and actively completing their tasks, there

is minimal strain on computing resources, which allows the events to run very closely to the desired rate. This is similar to the system's operation in degraded mode where few other periodic events are occurring.

It is worth noting that as more activities occur within the system, the average error is likely going to increase due to the additional strain on the computing resources. If fairly high frequencies are necessary, it is likely worth considering a kernel module with direct interrupt service routine access rather than using a static process, to provide better timing guarantees. Regardless, these results show that if required, the event handler can manage the majority of the scheduling needs for static processes in the system with good accuracy.

4.5 Design Success

The success of the design can be assessed by looking at how well the software architecture has met the design guidelines, functional requirements and philosophies.

An extremely flexible command system has been enabled by providing each individual process with its own command space via the command handler. The command handler also helps with radiation awareness by validating signed commands transparently, along with the software watchdog which monitors the processes for correct operation to insure no radiation events have affected functionality. The last of the guidelines, telemetry, has also been addressed in a number of modules: every process has both system and operational state information, the system manager enables access to vital telemetry in degraded mode, and the data logger provides a simple interface for storing telemetry of any kind.

The functional requirements for the system have also been sufficiently satisfied. The data logger process has been designed to specifically support the first requirement of telemetry collection and storage, by spawning processes to collect telemetry and then storing the resulting data into the database that is saved on the NAND flash non-volatile memory. The next requirement, a beacon, also has a dedicated process and in order to transmit this data, in addition to meeting the following requirement, the communication process has been designed to support transmitting and receiving AX.25 packets. As for the requirement dictating the specific commands, those have also been implemented; downlinking teleme-

try is available in a variety of methods, such as through the querying individual processes for function specific information or the system manager for satellite state. The final functional requirement has been built into the core functionality of the custom libraries with the event handler that can schedule a variety of events, which in turn enabled the data logger to easily schedule the execution of commands after their reception.

The design philosophies are also prevalent throughout the system. Software modularity is fundamental to the concept of the architecture, as all major components are isolated into processes that can be developed as a black box with clearly defined interfaces (commands for static processes and data logger interface for temporary ones). This also lends to the next philosophy of simultaneous development, which has been further supported by the custom libraries that enable development on operating systems other than embedded Linux, which PolySat team members are more likely to run on their own computers. The last philosophy of utilizing pre-existing code is evident throughout this design by the usage of a variety of open source and other sources of existing code. It especially common in the custom library that provides abstractions of the standard library, OpenSSL, and SQLite APIs.

By successfully following the outlined design approach, the PolySat team should be able to rely on this software architecture for any mission in the foreseeable future.

5

Fault Tolerance

Many mechanisms for providing fault tolerance to the spacecraft were researched and considered (see Chapter 7), but the result is that the majority of these add significant complexity and require far more man-hours than an educational project like PolySat can provide. A number of fault tolerance techniques have been implemented in the core of the system architecture, such as error logging (Section 4.3.5) and error detection checks with checksums on IPCs (Section 4.1.2). However, this chapter focuses on two components implemented specifically for their fault tolerance: digitally signed commands and the software watchdog process. Through these two aspects of the software system, it is believed that the majority of radiation events (SEUs and SELs) can be detected, logged, and reacted upon appropriately.

5.1 Cryptography: Digitally Signed Commands

5.1.1 Goals

The goal of the cryptography component of the PolySat library (Section 4.3) is to guarantee only valid commands are executed on the spacecraft. The definition of valid in this case has two different aspects: correct and allowed. A command is correct if no portion of the command was changed in transit from the ground station to the main processor and it is allowed if the ground station operator has been permitted to execute that command on the satellite.

Both of these goals are very important to the overall security of the satellite. If a command that had a single bit flipped was forwarded to a process on the

spacecraft, there is the risk of an unintended command being executed by that process. In some cases, this may not be a cause for concern; for example, if the beacon rate is adjusted by 15 seconds instead of 14 seconds (if the least significant bit were flipped). However, there are some incidences where this could be catastrophic if it were a mission-critical command, such as an attitude control maneuver.

Executing only permitted commands is vital to the security of the spacecraft for similar reasons, but a failure to provide this guarantee could enable rogue ground station operators to execute critical mission commands. As the public awareness of these CubeSat missions increases, there is greater potential for interference from unaffiliated ground stations. Thus, it is important to be able to provide a guarantee of security to ease the minds of those who have invested in the mission and also to insure this mission is not unintentionally compromised by executing an out-of-sequence command sent from a curious amateur radio operator.

It is important to specify that there is no intention to obscure the commands being sent to the spacecraft, but rather protect a certain set of commands that are vital to the success of the mission.

5.1.2 Requirements

The requirements of the interface are fairly straightforward: a simple API must be provided to digitally sign and validate messages sent from the ground station.

There are also requirements that bound the implementation of the digital signatures to maximize the security. An asymmetric (public/private key) algorithm should be used for generating the signatures and the private key utilized should only be installed on trusted ground stations. This system relies on this concept because if the key is distributed outside of its intended ground station, it would enable other ground stations to send commands under the guise of the trusted one and provide access to secured commands. This requirement provides the guarantee that signed commands originated from a trusted off-satellite source.

5.1.3 Implementation

The implementation of the cryptography relies heavily on the OpenSSL v1.0.0 library. It is worth noting this is another aspect of the satellite where reliability is significantly increased through the reliance on pre-existing code. Furthermore, this feature in particular would have been very difficult to implement an equivalently robust solution without this library. The general process that this cryptography component of the PolySat library supports is generating and subsequently sending a command with a signature, which is then transmitted from a ground station to the satellite, and then verified on the satellite for both correctness and security. This flow is shown in Figure 5.1.

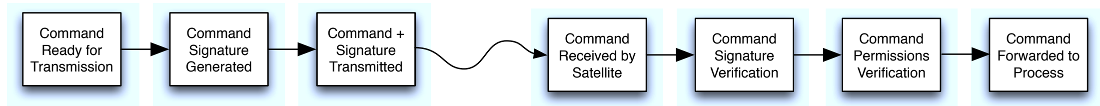


Figure 5.1: Signed Command Flow - High level flow diagram of a signed command.

Throughout the message signing and validating process, the data digest is a key component; the digest is created by generating a hash of the data. The SHA-256 hash algorithm has been chosen for generating the digest because the more common SHA-1 has been shown to be a more likely source of collisions [15]. It is important to note that while SHA-1 may be extremely secure, security is not the only feature we are looking for in this cryptography module, we also want to be able to guarantee any bit flips are detected. If two separate bit strings hashed to the same value, a collision, then it is possible an unintended command could be allowed to be executed in the system. Fortunately, the OpenSSL library provides a SHA-256 hashing capability, which is used throughout the implementation.

An important aspect of the message signing are the certificates that hold the public keys. For these keys, the X509v3 certificate type has been chosen because it has an extensions capability that can be used for customized security beyond just the keys and it is fully supported by the OpenSSL library APIs. This design utilizes the extensions to allow for different levels of security, such that a single ground station can be given permission to only be able to execute commands on a specific satellite, or even as narrow as a particular command. These certificates will be stored on the satellite so that when a command signature is verified, it can be insured that the source of the message not only has a valid key, but also

explicit permission to execute that command. The OpenSSL library is relied on once again for this, which provides a simple API for reading the certificates and iterating through the extensions.

The simplest function of the cryptography component of the library is the digital signing, for which the steps to accomplish it are shown in Figure 5.2. It heavily leverages the EVP component of the OpenSSL library, which is provided as the high-level interface to cryptographic functions [16]. The general process for signing a message is to create a digest of the data being sent, and then the signature is generated using a particular encoding scheme from that digest. The original data is then transmitted with the signature and the digest is discarded.

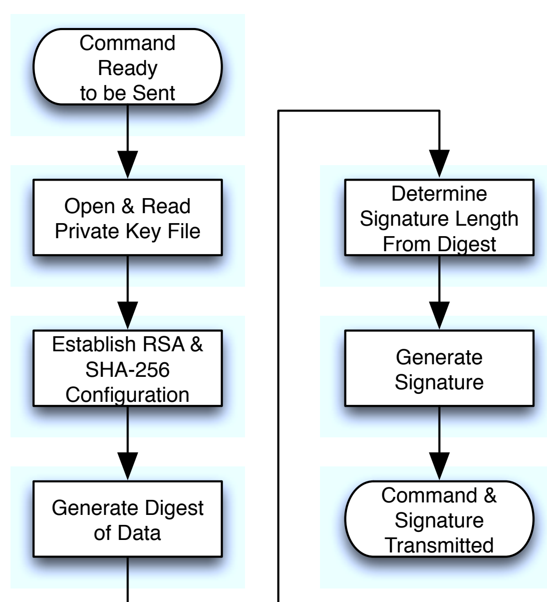


Figure 5.2: Command Signature Generation Flow - Flow diagram for signing a command.

The other primary function is the command validation, which also utilizes the EVP interface and other components from the OpenSSL library. The steps to validate a command are summarized in Figure 5.3. First, a digest is created of the command. Next, a public key must be used to attempt to match the signature to the digest. Since the command could have been sent from any ground station who has been authorized to send commands to the satellite, multiple public keys may need to be attempted until it validates. Once a public key is found that validates the the command, the permissions need to be checked. If no matching public key

is found that validates the signature, no response is given, in order to prevent Denial of Service (DoS) [17] attacks on the spacecraft. This type of attack could occur if an unauthorized ground station user continuously transmitted invalid commands that resulted in a response, thus tying up the radio resource and preventing trusted stations from uplinking to the satellite.

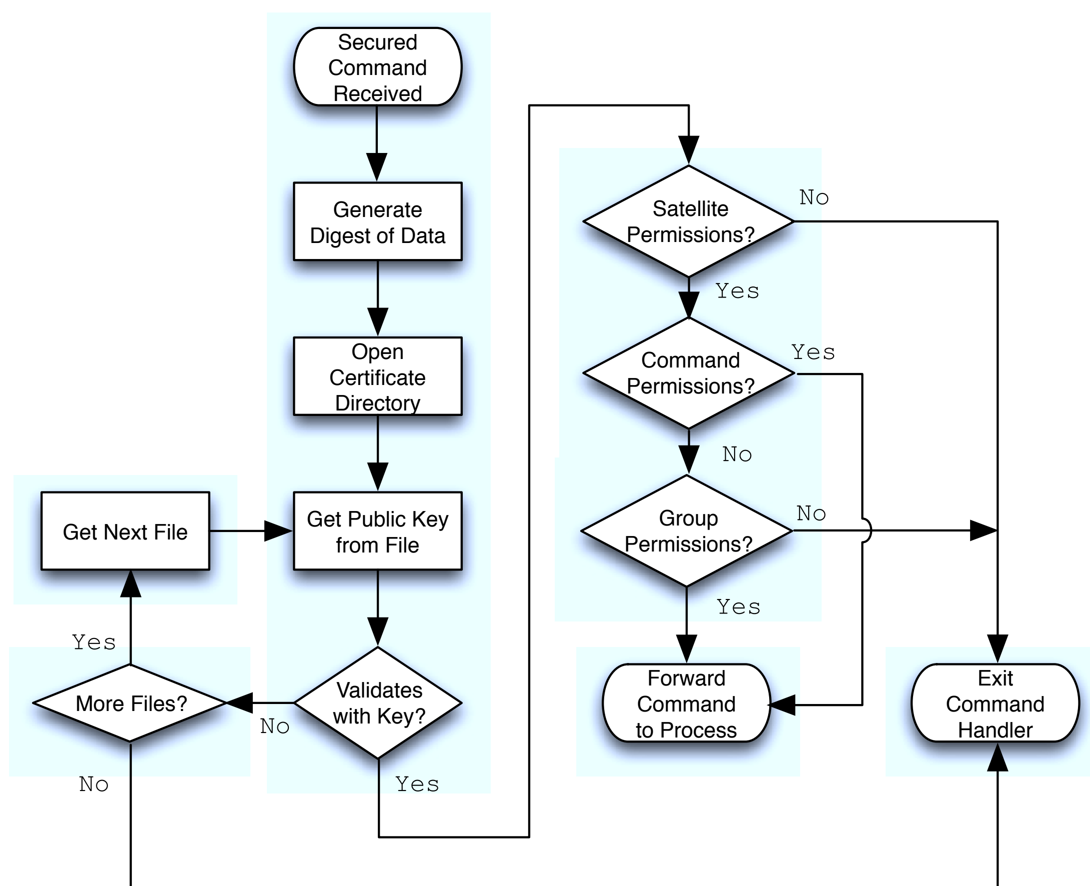


Figure 5.3: Command Validation Generation Flow - Flow diagram for validating a command. A command is forwarded only after both the command's validity and permissions are verified; otherwise the command handler exits, dropping the command.

There are three primary levels of permissions: satellite, command and group. Satellite permissions are checked first, to insure that the sender of the command has permission to execute commands on the satellite that has been sent the command. If that passes, it is checked to see if the ground station operator has been given explicit permission to execute that command. If that fails, then the last check is for group permissions. These type of permissions are customizable

based on the mission, for example, "payload" group permissions could be given to a customer's ground station to enable the use of payload commands, but not other system functionality. Other groups could be used following a similar concept, limiting the scope of access to the spacecraft based on the needs and involvement of the operators.

5.1.4 Results

The resulting digital signature is 128 bytes that needs to be appended to each command that requires security. This may seem like a large amount, but given a 9600 baud transmission rate with an encoding scheme that only requires one symbol per bit to the spacecraft (as proposed for LightSail-1 and CP7), only an extra tenth of a second is required to transmit the signature over the air.

The other primary downside of adding this feature is the potential additional latency added to command response and execution. Timing tests have been completed to show approximately how much this overhead actually is, and results are summarized in Table 5.1. These tests were set up and run on the AT91SAM9G20 microprocessor that is used in the target hardware platform. The tests were run to look at the cost of requiring a different number of certificates to be checked against the signature until one is found, or there are no more certificates to be tested. If only a single certificate has been installed on the spacecraft, then it will be the only one checked against signed commands. Therefore, the time will be about the same for both a valid and invalid signature because the most expensive operation is the signature verification. As a result, the number of certificates attempted can be interpreted as either the signature was verified after that number of certificates, or that was the total number on the spacecraft and no match was found.

The results show that overall there is very minimal cost in both time and transmit bandwidth, especially when there are a low number of certificates. With a single certificate on the satellite, only an additional 6.5 ms of time is required before a signed command can be executed. As more certificates are added, this time increases, but not linearly because the overhead of the other processing is being amortized, as indicated by the average time per certificate reducing. Also shown are the maximum number of bits that could have been transmitted (in response to the command) while this process is occurring. The goal is to show

Certs Attempted	Exec. Time	Time Per Cert.	Max Transmit Lost
1	6.5 ms	6.5 ms	62 bits
2	11.2 ms	5.6 ms	107 bits
3	15.7 ms	5.2 ms	150 bits
10	48.1 ms	4.8 ms	461 bits

Table 5.1: Digital Command Signature Verification Timing - Results from timing the signature verification operation with a varying number of certificates attempted.

that if the amount of information that could have been sent over the air is fairly minimal, even with a large number of certificates.

In spite of the delays necessary to support this capability, it is important to note that this feature will not be used for a majority of commands, or even a large number. It is specifically intended for those commands that are critical to the success of the mission, such as attitude maneuvers or one-time only mechanisms, like a mechanical deployment. Also, if coupled clever organization to reduce the number of certificates attempted for the most common ground station operator certificates, these costs can be minimized while still being required infrequently. Furthermore, until other ground stations are outfitted to be able to communicate with PolySat spacecraft, only a single certificate would be necessary for PolySat operators. This means that for a missions where only the Cal Poly ground station would be communicating with the spacecraft, this overhead is arguably negligible since only a maximum of 62 bits would be lost.

5.2 Software Watchdog

5.2.1 Goals

The software watchdog is the first line of defense in the series of the watchdogs contained within the new PolySat avionics system. The software watchdog's overall goal is to verify that all custom software components of the system are operating correctly.

Its first goal is to detect and respond to single event effects (SEE) in the digital components, which have affected the running software. These events are a direct result of the satellite's exposure to radiation while in orbit and can have

catastrophic results to the system if not handled.

There are two types of radiation events that the watchdog should be particularly concerned with: single event upsets (SEU) and single event latch-ups (SEL). SEUs are the more easily dealt with because they usually only result in a bit flip [3], which is not a permanent fault because it can be solved with a refresh of the affected memory. Stuck bits caused by SELs, on the other hand, are far more dangerous because a latch-up can only be cleared by removing the power [11]; there is also a risk of permanent damage to the IC, if not handled in a timely manner.

The other goal for the watchdog is to validate operational parameter changes after a reboot, so that any configuration change away from the default is verified to be correct. Providing this additional validation method through the watchdog provides a correctness guarantee for state changes and will prevent the satellite from entering an unknown or incorrect state after a reboot.

The final goal is that the software watchdog is the single point of interaction with both the internal and external watchdogs to limit the potential for unintentional tapping. These hardware watchdogs are the final line of defense; when the watchdog cannot do anything to recover the system, they should be able to initiate a reset of the processor or the entire system.

A related question posed is, "Who watches the watchdog?". Fortunately, the answer is simple: the internal watchdog! If the software watchdog stops functioning properly, the internal watchdog should initiate a reboot of the processor and result in a restart of the watchdog process. If the watchdog continues to fail to start properly, then the hardware watchdog should cause a full system reset.

5.2.2 Requirements

The requirements for the software watchdog can be separated into the three distinct responsibilities of the module: watching processes, tapping the hardware watchdogs, and validating configuration changes.

The primary requirement of the process watching capability is that it must be able to cover both static and temporary processes.

The temporary watching of a process will be utilized frequently by the data logger to watch the one-off processes, which should have a fairly deterministic completion rate. However, it can also be used by other processes who are going

to execute an important task that has an approximately known time to complete, but it is not periodic. Thus, a temporary watch should be able to be requested any number of times, for any process.

When watching static processes, the watchdog should be capable of detecting erroneous behavior by looking at the actual behavior of the process, where possible. If an error is detected, the offending process should be killed. The watchdog does not need to concern itself with re-spawning the process because other Linux system mechanisms can be utilized for this, such as `init`. If a process has been killed a significant number of times, this may suggest a latch-up exists in the system memory and a graceful shutdown should be initiated. A graceful shut down is one where the processes are alerted to clean up and prepare for termination.

In all cases other than when a shutdown has been initiated, the software watchdog is required to tap the external watchdog. The hardware watchdog should not be tapped if a potential SEE has been detected, so that a hard reboot of the system will be initiated and hopefully clear all SEEs. A hard reboot will cause power to be removed from the entire system for a duration of time sufficient to cause the power plane to drop down to ground and clear any latch-ups.

In addition to the external watchdog, the software watchdog is also responsible for tapping the internal watchdog, which will initiate a soft reboot when not tapped before the timeout. The software watchdog must only start tapping the internal watchdog after the system has been confirmed to have booted into minimum functionality, so that the system is not stuck in a half-booted state.

The last requirement for the watchdog is that it must have a command to verify configuration changes. The watchdog should be able to receive both the command and the signature from the ground, so that it can validate that the reconfiguration is both a permitted and correct change. If the configuration change is valid, the watchdog is responsible for informing the inquiring process that it can use the desired change.

5.2.3 General Implementation

The watchdog has been developed as individual static process, so that it can leverage all of the standard PolySat library capabilities. This section details the general approach for the implementation of the major watchdog features, such as the process watching and configuration validation.

The event handler in particular is heavily utilized: the static process watching is based around one regularly scheduled event, each registered temporary process has its own scheduled event, and there is a task for tapping the internal watchdog. Also, in order to register a process for watching or request command verification, commands are sent to the watchdog on its well-known UDP port. Lastly, in order to accomplish the verification, the API provided by the cryptography component of the library is used.

5.2.4 Static Process Watching Implementation

The most complex aspect of the software watchdog is the static process watching with four distinct stages, shown in Figure 5.4.

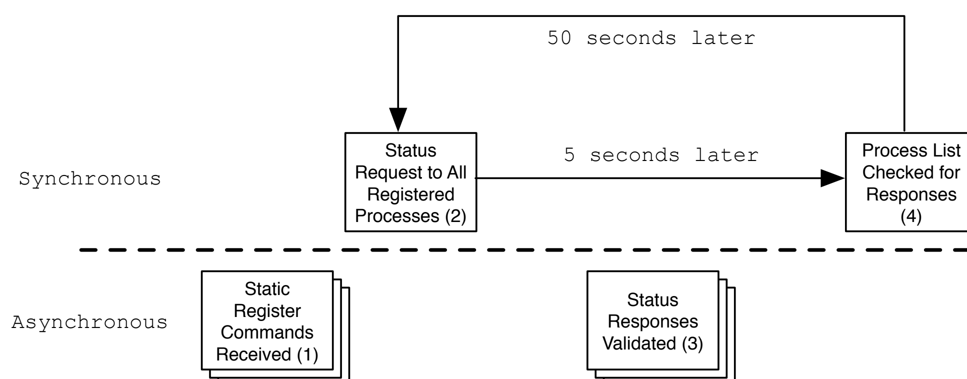


Figure 5.4: Static Process Watching - Flow Diagram of Watching Static Processes.

The first stage is asynchronous, triggered by incoming requests from other processes to be watched. The request includes the process name, which is stored in a hash table where the port number is the key. A fairly small table is able to be used because the static processes in the system will likely never exceed ten. This data structure also includes the state of the process, which is utilized in each of the different stages. When the request is received, the “watched” state bit is asserted, so that the next stage will know to send it a status request.

In the status request stage, the hash table is iterated through and all processes that have the “watched” state bit asserted get a status command sent to them. After the command is sent, the “status requested” state bit is set. This is important so that in the final stage only processes that have been sent a status command are expected to issue a response. After all status queries have been

issued, stage (4) is scheduled for five seconds in the future.

The next stage, status validation, is asynchronous because it depends on the other processes to respond to the status query. When the status packet is received, it is analyzed to verify that the sending process has been operating correctly since the last query. The analysis implementation is detailed in the following subsection. If the process is found to be operating correctly, a status bit is set to indicate this. Otherwise, if the status has been found to be invalid, it is indicated in a separate field that counts the number of invalid responses.

The final stage occurs precisely five seconds after the status queries have been issued to give the processes a reasonable amount of time to respond and also so that the external watchdog can be tapped at a regular rate. In this stage, the list is iterated through for the final time and the status bits are examined to ensure that all processes who were queried for a status have provided a valid response. This state bit is checked to guarantee that a process who has requested to be watched in between states (2) and (4) is not expected to issue a status response, even though it will be marked as watched.

If a process has not been indicated to have provided a valid status response, the watchdog will confirm that it is still running based on continuity between the .pid and .proc files (generated for all processes in the initialize process, see Section 4.3.7). If the process has already exited cleanly, these files will be have been removed. It also protects against one of the files being corrupted or another .pid file created with the same id because the two files need to match before a kill operation will be initiated.

Proceeding an attempted shutdown of a process, a variable in the process table entry is incremented that indicates how many times it has not provided a correct response. If this number exceeds five, a system wide signal will be issued to gracefully shutdown and then the entire system will get a hard reboot after a significant amount of time initiated by the external watchdog. If the system has not been requested to shut down, the external watchdog will be tapped to prevent a reboot.

5.2.5 Status Verification Implementation

The status verification process is key to the software watchdog's ability to determine the state of the watched processes.

Generally, the status for a process will contain a variable that is incremented after some kind of useful, regular event has been completed. However, some processes do not have periodic events, so receiving a status packet is deemed sufficient to indicate that the process is operating correctly because in order to issue a response, the primary components of the process must be working (e.g. event handler, command handler, and inter-process communications).

The contents of the status packet varies based on the actual function of the process who sent it, but the beacon is a simple example to examine. The beacon status contains two important fields: beacon rate and beacon count. The watchdog will keep track of the beacon count between each status query so that it can find the total beacons issued since the last request, and then use the beacon rate to derive what the actual number of beacons issued should have been. If the beacon count delta is found to be within the allowable window, it is marked as validated.

5.2.6 Temporary Process Watching Implementation

In order to register a temporary process, the following information is sent to the watchdog: the process ID of the watched process, a task ID, the maximum duration for the task, and an indicator of whether or not the requester should be informed if the process is killed by the watchdog. The process ID is provided so that requesters can start a watch on a process other than itself. The task ID is used to enable multiple watches per process and can be used by the requesting process to provide a meaningful task identification with a particular watch. Additionally, these two values are combined to generate a key into a hash table where this information is stored.

After an entry in the table is created with the parameters, an event is scheduled for the expected duration time in the future with a callback function that will kill the process. This event can be cancelled by sending a command to the watchdog with the process and task ID. The last parameter of the temporary process registration is relevant to this callback function, as well. A message will be sent to the requester of the watch after the process is killed, if its been requested. This cancellation command also has another parameter, which can be used primarily for debugging, indicating whether the watch is being cancelled as a result of successful completion or another reason.

5.2.7 Command Validation Implementation

The last aspect of the watchdog is the command validation, which is implemented as a single command. The implementation of the command is fairly simple, thanks to the cryptography API (Section 5.1) because the watchdog just receives the data and signature from another process, then runs the validation algorithms on it and returns the command to the process if it is validated. Additionally, this provides the watchdog with an opportunity to record the command data if it is relevant to the static process watching.

If the command validates, it will send the command back to the process with the signature intact, and the process will essentially replay the command. On the other hand, if the command does not validate, only an error will be logged and the process does not need to be notified since it would have already been configured to its default operating parameters.

Upon receiving the command back at the inquiring process, it will get validated once again because it will go through the command handler and it must be a protected command if a signature was used (see Section 4.3.2). This second check is necessary because it ensures that no bits had been corrupted after the message was sent back from the watchdog and will guarantee that the process is utilizing a configuration change that is both correct and valid.

5.2.8 Detection and Recovery Limitations

The software watchdog is unfortunately not an ideal solution for dealing with radiation events due to a variety of factors.

First and foremost, the granularity of the watch period is limited by the external hardware watchdog period, which is currently sixty seconds. This makes it difficult to watch processes with regular events that occur at a period that is far greater than this time with high accuracy because a decision needs to be made at a relatively high frequency about tapping the external watchdog.

The watchdog also currently does not have any capabilities to detect if a process is regularly causing reboots of the system, which may indicate a permanent corruption in the process binary. This type of behavior would have to be determined from the ground, as a result of the telemetry collected from the watchdog. A fix for this type of problem would also rely on the currently pending image

upload capability.

Furthermore, the reaction to an isolated watchdog process failure will likely have a far higher cost (in time and power) due to reliance on the internal watchdog to reset the processor in this event. If a second, independently developed, watchdog were utilized, so that each software watchdog could watch each other, the cost of an isolated failure in either would be far lower because the failed module could just be reset like a normal process failure. This is proposed as recommended future work (Section 8.3), to increase the robustness of the software watchdog.

5.2.9 Results

A number of tests have been run on the watchdog to validate the expected behaviors of the module. The current implementation of the watchdog has been confirmed to behave correctly in the situations shown in the Table 5.2, which includes all of the major requirements. Other tests still need to be completed with the hardware watchdogs and the entire set of degraded processes to fully validate these watchdog features. Additionally, timing tests have been completed to determine the time it takes, from start to finish, of validating a configuration change for another process; these results are shown in Table 5.3. General process related statistics for the watchdog, such as stored and run-time memory requirements, are shown in the general system results, Section 4.4.

The timing results were gathered over a series of 144 validations, with varying rates and a few process restarts in between to approximate potential actual behavior. Considering that two UDP messages are sent and the signature is verified at both the destination and the source (by the command handler), the time required is fairly minimal. On average, only 19.6 ms are required starting from when the requesting process sends the message until the command returns. The maximum time was shown to occur only on the first attempt and then there appear to be significant benefits from system caching, as indicated by the average.

The watchdog component provides a few critical features to the software system: it alleviates the burden of validating data from each process through command verification, in addition to observing and validating the operations of the static processes. The implementation results have shown that this is accomplished relatively efficiently and at little cost to the overall system.

Aspect	Event
Static Processes	Single process operating normally
	Multiple processes operating normally
	Single process operating incorrectly
	Multiple process with some operating incorrectly
	After 5 failures, shutdown signal issued
Temp. Processes	Single temp process watch, successful completion
	Multiple temp process watch, successful completion
	Single temp process watch, terminated
	Multiple temp process watched, some terminated
Command Validation	Only correct commands sent for validation are returned to process

Table 5.2: Validated Requirements for Watchdog - Requirements for the watchdog that have been successfully implemented and demonstrated.

Event	Min.	Average	Max
End-To-End Command Validation	17.5ms	19.6 ms	27 ms

Table 5.3: Watchdog Command Validation Timing Results - Timing results from the command validation watchdog feature, indicating how much time elapsed starting from when the request was issued ..XXX

6

Related Satellite Projects

6.1 Summary of Projects

Satellite [§]	**Organization**	**Main Processor**	**Mhz**	**Volatile Memory**	**Non-Volatile Memory**	**Operating System**
1st Gen PolySat [2]	Cal Poly	PIC18F6720	4	3.75KB	256KB	custom
2nd Gen PolySat [4]	Cal Poly	AT91SAM9G20	400	64MB	528MB	Linux
BeeSat-1 [6.2]	Technical Unv. of Berlin	LPC2292	60	2MB	16MB	TinyBOSS
Cute-1.7 [6.3]	Tokyo Institute of Technology	ARMV4I	400	32MB	128MB	Windows CE.NET
ITU-pSAT [6.4]	Istanbul Technical Unv.	MSP430	8	10KB	55KB	Salvo
KySat-1 [6.5]	Kentucky Space	MSP430	*N/A*	*N/A*	*N/A*	Salvo
MEROPE [6.6]	Montana State	MC68HC812A4	8	1KB	154KB	custom
QuakeSat [6.8]	Stanford University	ZFx86 486	100	16MB	1MB	Linux
STUDSAT [6.9]	India - 7 Academic Institutions	AT91SAM9260	180	64KB	512KB	VxWorks
PW-Sat [6.7]	Warsaw Unv. of Tech.	AT91SAM7X	55MIPS	256KB	2MB	FreeRTOS
UWE-1 [6.10]	University of Wuerzburg	H8S-2674R	*N/A*	8MB	4.5MB	μCLinux

Table 6.1: Comparison of processing specifications and operating systems of CubeSats - These projects were chosen based on availability of published design and specification documents. The section is given where they are discussed in this paper.

A number of related satellite projects were surveyed in order to facilitate and support this design. A list of these satellites and their computing specifications are shown in Table 6.1, including the first and second generation Cal Poly designs. A link to the section is also given where a summary is given of works published about the project's software system, followed by an analysis of the system and a comparison to the new PolySat system, when possible.

6.2 *Dependable Software (BOSS) For the BEESat Pico Satellite*

BEESat is a CubeSat developed primarily by the Institute of Aeronautics and Astronautics of the Technical University of Berlin in Germany. BEESat makes the claim of being the "first pico-satellite with a fault tolerant design" [18], which is implemented at a variety of levels.

6.2.1 BEESat Spacecraft

The satellite contains the typical subsystems, including the on board data handling (OBDH), power, communication, ADCS, and payload.

The OBDH consists of two redundant microprocessors, ARM7 microcontrollers each with 16MB flash memory and a redundant CAN interface for intersubsystem communications. This subsystem controls the typical communication components (transceiver and modem), which transmits to the groundstation in the UHF band. The OBDH also controls the ADCS subsystem that has the capability of 3-axis stabilization through the use of 3 microwheels. The satellite also has a camera as the payload, which communicates with its own microcontroller that is implemented with cold redundancy via a watchdog timer, current protection circuitry, and backup software images.

6.2.2 Software Architecture

The BEESat satellite utilizes the BOSS real-time embedded operating system and its provided middleware. This was chosen because the operating system was designed specifically for safety critical applications with an emphasis on simplicity. Additionally, the BOSS middleware enables communication between any combination of software or hardware elements, which simplifies the development process.

The middleware was specifically designed with fault tolerance in mind. The architecture relies heavily upon asynchronous message passing with a subscriber protocol. Any hardware or software module can subscribe to the messages of another module; all subscribers will receive a copy of the messages from that module. This feature makes it very simple to add logging systems or online

system diagnosis. An extension of this system adds further fault tolerance by providing the capability of inserting a voter between the middleware and the module.

Different tasks in BOSS are implemented as threads, which are scheduled by the kernel. The kernel also provides a time manager service, which provides a 64-bit counter for each node to keep track of the local time. The kernel also provides an interrupt manager to pass interrupts to user level modules. In order for this to function with the specific microcontrollers on BEESat, a small hardware dependent layer is required for interaction with the hardware specific components.

6.2.3 Results

BEESat was launched in September 2009 and entered a 720 km sun-synchronous orbit. After a year on orbit, BEESat was still operating nominally and none of the redundant subsystems were required to be activated [19].

6.2.4 Analysis

Although the BEESat system runs an RTOS and thus the low level architecture is relatively simple, their design philosophies contain some similarities to the PolySat system. Specifically, we also are attempting to utilize generally fault tolerant software mechanisms.

Furthermore, the success of our project is encouraged by projects like BEESat that have had significant on orbit operability without having to fall back on redundant subsystems because we have decided to eliminate major redundancy from this generation of avionics hardware.

6.3 *A PDA-Controlled Pico-Satellite, Cute-1.7, and its Radiation Protection*

The Cute-1.7 pico-satellite is a PDA-controlled spacecraft developed to the CubeSat specification by the Tokyo Institute of Technology [20]. The primary mission for this satellite is to validate the usage of COTS components for future missions. Magnatorquers were also flown to test an attitude control algorithm.

6.3.1 The Cute-1.7 System

The satellite is developed around a term coined by the developers of the Cute-1.7 spacecraft as the "SatelliteCore." This contains all of the primary components of the satellites and can be joined with what is referred to as a "mission container", which holds the payload, to form a whole spacecraft. The SatelliteCore only requires a standard USB interface to communicate with the payload and provides 3.3V, 5V, and unregulated power rails, to suit a variety of needs. The goal of this method is to require only a new mission container to be developed for each new satellite to reduce development time.

The SatelliteCore requires 1U of volume and thus it is expected that when combined with a mission container, the entire spacecraft will be at least 2U.

Advanced features such as attitude control were left out of the SatelliteCore design to optimize for cost and production time, rather than functionality. SatelliteCore aims to enable frequent access to space and adding the complexity that these features require would make that more difficult.

In line with the goal of rapid development, the majority of the components used are COTS devices. Specifically, parts are actually removed from end user products rather than commercially available electronics. For example, the transceiver used is an FM transmitter and receiver from a commercially available handheld. Furthermore, a personal digital assistant's circuit board is used as the main computer. This computer runs the operating system Windows CE 4.1 and the primary peripheral interface is USB, both of which are (or were at the time of the design) common enough to support the rapid development goal of the overall SatelliteCore design.

6.3.2 Radiation Protection

A number of fault tolerance mechanisms were added to the Cute-1.7 system to attempt to provide radiation hardening of the COTS parts. The two primary components are a double watchdog timer system and over current protection circuitry. The watchdog system has two separate timers so that both the computer and the real-time clock, which is isolated from the rest of the system, can be protected.

The goal of these systems are to prevent the computer from failing as a result

of SELs or SEUs. Sometimes an SEL can be indicated by increased current consumption, however, given that current consumption can vary greatly based on the CPU load, this may not be sufficient for reliable detection. The watchdogs can aid in the event current consumption does not spike high enough to trip the protection circuitry and ensure that power is cycled to clear the issue. When an SEU occurs, the effect is similar: the watchdog timer will restart the system in the event of a software halt.

The radiation protection system was validated by using a proton beam, which was wide enough to cover nearly a quarter of the main computer's circuit board. This was used to force SEU and SELs to happen to the computer system and hopefully allow observation of the recovery mechanisms. The results of this experimentation showed that the Cute-1.7 system should be able to function as intended for approximately a year, which is a reasonable amount of time to complete a CubeSat-type mission.

6.3.3 Analysis

The Cute-1.7 is one of the earliest published papers about designing a generic CubeSat platform and thus many of their design principles and goals were similar to those of PolySat's current design. The PolySat design seeks to improve turnaround times by establishing an extremely generic platform and improves upon the Cute-1.7 concept by requiring far less volume, but also providing a larger variety of interfaces.

Similarities are also found in the specific implementation: such as the use of two watchdog timers in the PolySat system, to protect both the CPU and the other components in the system from radiation events. The PolySat system hopes to provide further protection at the software level, as well. Additionally, PolySat also chose a commercially available operating system to enable the usage of common tools and languages, which were the likely motivations for Cute-1.7's usage of the Windows CE operating system.

6.4 iTU-pSat: Istanbul Technical University's Student Pico-Satellite Program

iTU-pSAT I is a 1U CubeSat and the first student-designed pico-satellite of Turkey [21]. It contains two payloads, both developed in-house: a low-resolution camera and a two-axis passive stabilization experiment. Its system is mostly comprised of COTS parts, with the structure and C&DH board acquired as part of the CubeSat Kit from Pumpkin Inc. The satellite was launched was in 2009 and was beaconing regularly as of spring of 2011 [22]. The on-board modem failed shortly after the launch and thus uplink to the satellite has not been possible.

6.4.1 Software

iTU-pSat also utilizes the real-time operating system provided by Pumpkin Inc., Salvo. The guiding design principle for their software architecture was to encapsulate one function or feature per task. The function of the main controller is primarily to control and collect data from other subsystems via I^2C, and then dividing that information into packages for transmitting to the ground. The system has been designed such that it can boot from only the flash memory of the microcontroller in the event of external memories failing.

6.4.2 Analysis

The iTU-pSat satellite is another example of a university project that was made possible by COTS parts, but it is unclear if future missions are planned with this system. Further details are necessary regarding the implementation of their software architecture and fault tolerance to determine the extensibility or flexibility of the system.

6.5 KySat-1: A Kentucky Space 1U CubeSat

The KySat-1 CubeSat was developed as a collaborative effort, by the Kentucky Space consortium. It has a 1U form factor and uses almost entirely all COTS parts, including both the electronics and structure from Pumpkin Inc. and Clyde Space Ltd [23]. The primary mission of KySat-1 was to attract the interest of

K-12 students in the STEM field through various interaction with the satellite while in orbit. KySat-1 was launched with the ELaNa-1 [24] mission in March 2011, but unfortunately did not reach orbit due to launch vehicle failure [25].

6.5.1 System, Requirements, and Software Infrastructure

KySat-1 utilizes an MSP430 microcontroller with the Salvo OS, both provided by the Pumpkin Inc., for their C&DH subsystem. Salvo is an RTOS specifically designed for small applications [26]. The software that ran within this operating system was designed by the Kentucky Space team and developed as part of their goal to create a "reliable and reusable CubeSat [system] architecture" [27].

The team developed a set of functional requirements, which included both the general and payload specific functionality. They dictated various high-level functions such as being able to configure the satellite before launch, providing command scheduling and execution, telemetry reporting, and capturing photographs.

In order to support meeting these requirements, Kentucky Space needed to create a software development infrastructure, since one did not exist prior to this project. Some of the decisions to develop this infrastructure included deciding to utilize a common version control system, SVN, standardize the development environment, and run a static analysis tool, Cleanscape C++ Lint, on all flight code.

Several strategies were also employed to aid development. The software was built in revisions, such that each revision was a potential flight candidate, if the schedule necessitated it. Four flight-ready revisions were released of the flight software, including the final fully-functional one. The software was also developed with a highly modular architecture, in order to facilitate dividing work amongst developers and simplifying future code modifications. Extensive work was put into designing the module interfaces to insure they were simple and intuitive to use. Lastly, each module was required to be reviewed by another programmer before being accepted as flight code.

6.5.2 Flight Software Overview

The KySat software architecture is divided up into a large number of modules, where each exists as a task or a set of tasks within the operating system. To

multiplex access to the hardware within tasks that used shared peripherals, global semaphores are used; in order to share data between tasks, queues are utilized.

The system is driven primarily by groundstation commands: data is received by one of the tasks that interfaces with a radio, it is decoded by the corresponding driver, then sent to a packet receive task where it is sent to queue that feeds to a command executor or to the transmitters, if the packet is to be "repeated" (sent back over the air). Once the command executor receives the command, it is parsed and it can either be scheduled to be executed immediately, or at a later time. When an action is taken, the command executor is also responsible for issuing an acknowledgement to the ground.

There are also several modules implemented for specific commands. This includes both a digital and continuous wave beacon, a few file system related commands, and a telemetry window command, which initiates a high rate of data recording for a set of sensors. The output of the telemetry window command is written to an ASCII-formatted file, which can be downlinked during or after the window is completed. Other commands utilize files, as well, such as those related to images and audio playback; all files are stored on an SD card.

The approach for general telemetry storage is somewhat unique in that they simulate a database through the use of macros that point to different indices in one large array. Each telemetry point has a specific macro that is used, which contains the specific index for that point. These telemetry points are updated at a rate of once per second, enabling other software modules to read from the database in RAM to get a reasonable recent value rather than reading from the sensor itself. Most modules in the system write to at least one telemetry point.

6.5.3 Fault Tolerance

Most of the components in the KySat-1 system have little or no flight heritage and thus several fault tolerant mechanisms have been utilized to help mitigate the risks.

The system relies on two watchdogs to protect against hang-ups that can occur as a result of radiation events. The first watchdog is internal to the microcontroller and is configured to timeout in just a second, but it is tapped at a rate double that, unconditionally, within a system task. When this system task is running, it indicates that the scheduler is functioning properly, which is a key

to the correct operation of the entire system. The second watchdog is external to the microcontroller, has a longer timeout of sixty seconds, and is able to perform a hard reset of the system. This watchdog is also tapped at a rate double the timeout, but only when all of the global semaphores in the system have been freed, indicating that there is no dead-lock. It is noted that this does add complexity, but that it is worthwhile to insure the system can recover from improper semaphore usage.

In order to protect against radiation events that can alter data in the EEPROMs, where satellite parameters and settings are stored, the hardware library stores all values in three locations within the memory. A voting system is used whenever a read of the EEPROM is executed, and in the event that one of the three values is incorrect, the correct value can still be returned, and it is also written back out to fix the corrupted location. If more than one location has become corrupted that is associated with a single value, the values will get set to their default/launch value. A system reset command is also implemented to allow for all of these values to get reset, in case the satellite appears to be operating incorrectly in spite of these mechanisms.

There is also a module that is dedicated to monitoring the activity of the system. Generally, the satellite will be interacted with at least once every ten days, thus, if no packets have been received within ten days, it is probable that some kind of critical error has occurred. When this is detected, this error scanning module will utilize the system reset command to return the satellite to its default configuration, in an attempt to clear the error. The error scanning module is also responsible for sniffing parameter changing packets that come to the satellite, and if a parameter is found to be requested that is outside of the acceptable range, the parameter is restored to its default value.

An additional mode has also been designed in the event of no communications for ten days. When the satellite reboots into the default configuration mode, as a result of the error scanning module, the worst case is assumed and antenna deployment is re-attempted. After this, if no packets are received for another three days, the satellite is booted into a system recovery mode. In this mode, a minimum set of hardware and software is used in hope that none of the critical components have been permanently damaged. Additionally, in this mode, the system toggles between the two primary radios and issues continuous wave bea-

cons every 4.5 minutes. Only two commands are available to the ground station operator at this point: normal reboot, or normal reboot with a new default radio. The beacon contains information to determine which of these are the appropriate choice for recovery.

6.5.4 Analysis

KySat-1 is another project whose software system was developed with similar overall goals to the new PolySat system: an extensible, modular, generally fault tolerant software architecture. The approach differs a bit due to the usage of a lower power microcontroller, with the Salvo RTOS. This design choice has required the KySat-1 team to seemingly integrate a lot of non-generic code inside their command path, in order to handle forwarding commands to payload specific modules. It is unclear whether or not a generic method of implementing this was utilized. This was a pitfall of the first generation PolySat design, and intentionally avoided in the new design by enabling each process to simply handle their own commands with the provided command handler.

Furthermore, the use of an RTOS required extensive use of low-level operating system objects and services, such as semaphores and queues, which they admitted adds fairly significant complexity. The PolySat system was able to avoid this by leveraging abstractions of similar concepts that already existed in the Linux kernel and standard libraries. The same can be said for the custom "database" implementation for KySat-1; rather than a custom implementation, the PolySat system was able to leverage a commonly used implementation, SQLite. Furthermore, using an RTOS at Cal Poly would likely fundamentally result in a steeper learning curve, because there are no lower level courses that teach programming in such an environment, and until recently there was not even a higher level one.

As for the fault tolerance methods, KySat-1's usage of watchdogs is extremely similar to PolySat's. Specifically the use of two hardware watchdogs, where one is unconditionally tapped at a high rate, and another with a condition tapped at a lower rate. The implementation differs a bit though, in that the functionality of PolySat's software watchdog exists partially throughout the KySat-1 system, such as the parameter validation. However, there is no mention of validating the correct operation of the modules, which is a feature provided by PolySat's software watchdog and can be used for detection of radiation events.

KySat-1 does provide some fault tolerance mechanisms that are not included in the PolySat system, such as the redundant parameter storage. It is believed that the checksums used to validate the file system in the bootup process are sufficient to detect permanent faults and that voting systems such as this add significant overhead and complexity in a place that has not fundamentally been identified to be a consistent source of errors.

The failure modes are also an interesting concept that the PolySat system has implemented a minimal form of (degraded versus full functionality modes), but has avoided due to the lack of operational history suggesting such modes are necessary. The KySat-1 implementation mostly focuses on communication system failure, which the Cal Poly missions seemingly identified as a consistently reliable system, whereas it was the C&DH component that was mostly likely to fail. Thus, the majority of efforts seem to be best spent making the primary computer robust to failures rather than the components around it.

6.6 MEROPE: Montana EaRth Orbiting Pico-Explorer (MEROPE) Cubesat-class Satellite

The MEROPE spacecraft was a 1U CubeSat that was launched on the failed DNEPR-1 rocket in 2006 [28]. It was the first satellite developed by Montana State University and its primary goal was to be a modern day version of the Explorer-1 science payload [29].

6.6.1 C&DH Hardware

The project was entirely student run and low budget, which necessitated the use of mostly COTS components. Since no radiation hardened components could be afforded, goals and requirements for the mission were decided appropriately. The mission was only required to last four months and only a 40 KHz data rate from the payload was needed. The microcontroller chosen to support this mission was the 16 bit MC68HC812A4 (commonly referred to as the HC12), running at 16 MHz with 1KB of RAM, a 4KB EEPROM for program storage, and a 150KB FIFO RAM device for external memory.

6.6.2 Software

The software was programmed in assembly, which was chosen for its minimized code space requirements in comparison to C. As a result, the software is a fairly simple interrupt driven architecture. The main loop is responsible only for telemetry monitoring and storage, tapping the watchdog, and checking the receive buffer for data. All other activities on the satellite are initiated by interrupts. Interrupts insure that the desired data rate can be achieved.

Two different modes of operation were programmed for the MEROPE system: probe and on-orbit. The probe mode is used primarily during development and is entered when a serial cable is attached and the batteries are being charged. In this mode, systems diagnostics run, and the chip can be reprogrammed. In on-orbit mode, all activities of the satellite are autonomous. There are three phases in this mode, the first is when the satellite is first deployed from the P-POD and it must wait a certain period of time to deploy the antennae, to prevent collisions with other spacecraft. The second phase is normal operations, which is entered after the antennae have been deployed, and the standard main loop is run. The third and final is the transmitting phase, which occurs when the satellite is overhead Montana State University's ground station and data is being sent over the air from the satellite to the ground station.

6.6.3 Analysis

This is one of the earlier CubeSat missions and thus, it has more parallels to the first generation PolySat design than the newer one. The concept of a main loop that manages the telemetry, and all other activities are triggered by interrupts is very similar. However, given that it was programmed in assembly, it is even simpler in implementation.

This project is included to show the significant technology leaps that CubeSat C&DH electronics and software have made in just the past decade.

6.7 *PW-Sat on-board flight computer, hardware and software design*

The PW-Sat is a CubeSat developed by various faculty of the Warsaw University of Technology in Poland [30]. It's primary goal is to demonstrate a satellite deorbiting mechanism based on a shape memory alloy sail, which could be used a cheap method for removing spacecraft from low earth orbits. In addition to this experiment, they are also flying an RTOS-based software system, which runs on an ARM7 microcontroller.

6.7.1 Hardware Design

The on board computer for PW-Sat consists of an Atmel AT91SAM7X ARM7 microcontroller, which operates at a speed of 55 MIPS. For telemetry and scheduled command storage, a 16 megabit DataFlash component is used. A one-wire bus is utilized to interface the microcontroller with a temperature sensor and the radio communication module.

Based on the analysis of their intended orbit, they expect to receive 1 krad of total ionizing dose every 30-days and it is believed that the commercial CMOS components that have been selected should continue to operate nominally.

6.7.2 Software Design

The software design is mostly based around the FreeRTOS real time operating system. One of the stated reasons for choosing this system is that it can be used on a variety of hardware platforms.

The software architecture is highly modularized, with each module existing as a FreeRTOS task. Communication is accomplished by using different built-in services and objects like queues and semaphores. There are ten unique software modules, which each accomplish very specific tasks for the spacecraft.

The first module is the "USART Communication Module", which is responsible for the RS232 interface and low-level control of the satellite's radio. After commands are received by this task, they are sent to the "Telecommand Parser and Validator", which first authenticates the commands using a keyed-Hash Message Authentication Code utilizing the SHA256 algorithm. Each command has a

32 character signature, which relies on the secrecy of the key that is used to generate and validate the signature. After verification of the signature, the module converts the command into binary and sends it to the "Telecommand Scheduler".

The scheduler is responsible for queuing received commands for execution, either in a relative time or absolute. The capabilities for repeating the commands with a specified period, removing commands, and viewing the currently scheduled commands are also provided. One of the challenges faced in the implementation of this module was providing persistence between system resets. The approach they ended up using was periodically writing out the contents of the queue to their flash memory. The downside of this option was that commands between the last save and a reset would be lost, but the frequency was high enough to reduce the likelihood of losing a significant number of commands. With the command state, a persistent time stamp was written out as well, in lieu of an onboard real time clock, to enable consistent time keeping. Once a scheduled command is to be executed, it gets sent to the "Telecommand Router", which is a very simple module that merely places the command in the appropriate queue.

The other modules provide higher-level functionality, such as the sensors module, which gathers voltage and temperature information from the on-board sensors. The "House-Keeping Module" also uses this sensor data, in order to monitor the state of the spacecraft and send out alerts to the system if sensor values exceed certain upper or lower bounds.

The next module is responsible for overall system health and observation, the "System Module". It contains a software watchdog component, which collects heart beats from every module in the system, in order to detect deadlock. When every heart beat is collected, a hardware watchdog is tapped.This module also keeps track of the reason for reboots, logging the past 50 reset causes, and also the past 50 failing modules as detected by the watchdog.

The last two modules are for collecting and archiving telemetry from other modules. The general telemetry module is solely responsible for sending telemetry to the communication module. Priorities are given to each telemetry packet: low, normal, or high. High is used exclusively for data only packets, but the typical telemetry packet will be assigned a normal priority. The data is removed from the queue based on the priority as soon as the on board computer has been informed the radio is ready to transmit. Since the opportunity for transmitting

data to the groundstation is very limited, these queued telemetry packets need to be archived, which the second module is responsible for. Each archive entry has a time stamp and is saved inside the 2MB memory, which is treated as a circular buffer (a maximum of about 15000 packets can be stored at a time).

The final aspect of the software discussed is software redundancy. In general, the system relies on the hardware watchdog to recover from single event upsets and latch-ups. However, if there have been 15 consecutive restarts, the system will enter a safe mode. This safe mode utilizes redundant code and redundant SRAM memory, in order to maximize reliability, and implements the minimum functionality required to generate and broadcast telemetry packets. After 15 minutes of operation in this safe mode, the system will forcibly restart and attempt to reenter the normal mode.

6.7.3 Analysis

The PW-Sat software architecture bears many resemblances to PolySat's new design. Other than the underlying operating system difference (FreeRTOS versus Linux), the approaches are nearly identical: reliance on a higher power processor to enable significant software modularity. Parallels can be found in nearly all of the modules, such as the communication, telemetry, and system/watchdog modules.

However, there were a number of modules that the PolySat design did not require a custom implementation thanks to an existing one in Linux. For example, the majority of our command routing is provided by the Linux kernel with the well-known port numbers rather than implementing a custom module as PW-Sat was required to do. Additionally, they required a custom format for storing their telemetry packages, whereas our design was able to leverage the SQLite package, which enables numerous other features besides efficient storage.

The software redundancy strategy is also fairly similar to that of PolySat's system, with a reliance primarily on the watchdog to initiate restarts upon detection of a radiation event. The similarities do not stop there either, both systems have a degraded operation mode which can be used to attempt to recover from significant failures. However, the implementation differs slightly due to the intended recovery option. The PW-Sat does not have any mentioned code upload capabilities and thus the only way to recover from a serious failure is to wait it out. As

for the PolySat system, work is being done to be able to upload an entirely new image, which would enable recovery from both transient and permanent faults in the software image.

The PW-Sat software design is an interesting precursor design to the new PolySat design. The new system hopes to relieve the burden of requiring the developer to learn a new operating system by utilizing Linux instead of FreeRTOS. FreeRTOS should provide the better performance overall, but the limited selection of open-source/free code available and the likely increased learning curve are significant detractors in an educational environment.

6.8 *QuakeSat Lessons Learned: Notes from the Development of a Triple CubeSat*

QuakeSat [31] was a three unit CubeSat that succesfully flew powered by a Linux-running CPU. The duration of the entire project was 18 months and in that time it was succesfully built, launched, and operated for a number of months. This paper details their lessons learned from this mission, specifically regarding the differences of developing a CubeSat versus a larger satellite.

6.8.1 Operating System Selection

The relevant portion of the paper to this thesis is the discussion of their software system. The QuakeSat mission selected a COTS single board computer, which came with a fully functional Linux OS. The availability of this OS was a driving factor in their decision because they did not have to write any I/O drivers and thus could focus on higher-level software and payload integration. The choice of Linux as an operating system was based on the availability of third-party software, such as AX.25 support, ease of development, and compatibility with existing ground systems. Furthermore, they found numerous utilities for software telemetry and also utilized the existing IP network stack within Linux. Thanks to all of the existing Linux code-base and support, the QuakeSat team only had to write 10,000 lines of code, only 30% of which was device driver code.

The QuakeSat team did recognize some disadvantages to using Linux though, for example they found more testing was required due to the greater flexibility

provided by the existing code. Additionally, they acknowledged the time was not taken to validate the large number of files that are required to support the Linux kernel and thus they were not able to guarantee the validity of the operating system.

6.8.2 Satellite Software

The satellite software architecture primarily consisted of independent programs. The programs had few or no interdependencies and thus enabled concurrent development with minimal interfacing between the developers. As a result, this reduced "development cost, development time, and software complexity". One of the other stated benefits of this type of architecture is that the overall risk is reduced for the programs because no interaction means little chance of locking up from a race condition that went untested.

The software architecture can be described as having four distinct layers: operating system (Linux), device drivers and watchdog, command handling, and applications. The command handling is done by the "QuakeSat Executive", which receives a command and sends a respond, to ensure for every packet up there is one packet down. Three primary application types are used: beacon, which sends telemetry information, file upload/download programs, and worker programs, for executing "time-tagged" sets of commands (i.e. a multi-day experiment). Additionally, scripts were used to perform a large number of operations on the spacecraft, such as maintenance, control other programs, and sometimes execute payload commands.

After the launch of QuakeSat, testing revealed that the existing flight software needed to be modified. Due to the modular structure of the software, the team was able to upload new code without bringing the system down entirely. Typically, the replacement file was uploaded with a new name and then the appropriate scripts were modified to use the new name, allowing the previous copy to exist as a backup.

As the mission progressed, scripts were also modified to move towards automated operations as the mission became a "highly-refined set of predetermined tasks."

6.8.3 Analysis

Given that both QuakeSat and the new PolySat avionics software system run on the Linux kernel, there are a number of direct comparisons that can be made between the two. The majority of the reasons given by the QuakeSat team for using Linux, such as the existing IP network stack, were factors in PolySat's decision, as well. However, PolySat opted out of some of the third-party software, like the AX.25 support, in favor of a lightweight, custom solution, that will be optimized and potentially replaced in the near future for higher network efficiency.

QuakeSat's software architecture running atop Linux also contains a number of similarities to the PolySat design. At the highest level, both systems utilize processes that are mostly independently functioning. The primary difference lies in the fact that QuakeSat opts for a command handler layer, whereas PolySat chose to have each process individually manage their commands, to allow for maximum amount of flexibility and code isolation. The PolySat process library provides all of the necessary command handling infrastructure in a generic fashion, such that an individual process for handling this is not needed.

Both spacecraft utilize a beacon process for periodic telemetry broadcasts; but, in the PolySat system, for scheduled events the data logger is used as a generic interface for spawning one-off events or experiments rather than a number of "worker programs", as QuakeSat uses. This enables greater flexibility for adding different experiments and does not restrict them to be written in the same language as the core system functionality.

As the architecture is designed, there is no specific infrastructure for code uploading and replacing. However, this is being developed as a future thesis and should be available in the near future after the completion of this work.

6.9 STUDSAT: India's First Student Pico-Satellite Project

STUDSAT is a pico-satellite developed by a group of about 40 students from the Indian Engineering Colleges of Hyderabad and Bangalore [32]. The form factor for the satellite is approximately a 1U CubeSat, as a result of a camera lens that extends out an additional 3.5 CM in the +Z direction (to result in overall dimensions of 10cm x 10cm x13.5cm). In addition to the camera, also included is an ADCS subsystem with pointing accuracy to 1 degree. For the system electronics,

the satellite uses an off-the-shelf power system from Clyde Space, but has a custom C&DH board centered around an AVR32-based UC3A0512 microcontroller, which is supported by a 512 kB Ferroelectric Random Access Memory (FRAM) non-volatile memory.

STUDSAT was launched in July 2010 and was successfully transmitting, however, due to high amounts of noise, no packets were able to be decoded.

6.9.1 Software Architecture

STUDSAT utilizes a widely used RTOS, VxWorks, for its operating system. The architecture they have chosen is known as a shared load ring architecture [33]: on the innermost ring is the operating system, the next level are the applications, and on the outermost ring are the drivers. Their device drivers are independent blocks of code, which are used as the building blocks to construct an application.

The satellite was programmed to have three different modes of execution: mission, launch, and check out. In mission mode, the satellite is performing operations related to the desired mission sequence, such as taking pictures, or establishing a link with the ground station. Immediately after launch, the satellite enters launch mode, which is responsible for executing a series of tasks that need to occur after entering orbit, like antenna deployment and de-tumbling the spacecraft. The other mode is called "check out mode" and is used on the ground for testing purposes. Its operations includes a test procedure to validate the functionality of each of the individual subsystems, and it also enables passing commands to the satellite through a UART or USB connection.

There are also three power modes that affect the operations of the spacecraft and are controlled by the software monitoring the power levels. In emergency mode, all systems are disabled except for the C&DH and power systems; the C&DH is only responsible for periodically checking the power level and logging the telemetry. If the power level recovers, the satellite will enter low power mode, where the beacon module will also be enabled, which will transmit the basic telemetry data every two minutes. The last is "optimum" power mode, where all modules can be used, on an as needed basis. This includes continuing to keep track of the power levels and beaconing, as well as establishing a communication link with the ground station, attitude determination and control, taking images, and performing general housekeeping.

6.9.2 Analysis

This is the only satellite looked at using the VxWorks operating system, but its architecture is fairly similar to the others using an RTOS. Few details are given about the separation of the tasks so it is difficult to say whether or not the STUDSAT team intentionally designed for modularity in their software, but the tasks that have been presented seem to indicate they have mostly done that. Unfortunately, little is presented about which, if any, fault tolerance mechanisms have been used either. Thus, STUDSAT serves as an interesting subject due to its relatively unique hardware and operating system, but it is hard to make a comparison to the new PolySat software system due to the lack of details available.

6.10 The UWE-1 and UWE-2 Satellites

Students from the University of Wuerzburg, Germany developed a 1U CubeSat, UWE-1, which was launched in October 2005. The UWE-1 satellite's primary mission was to test a number of different IP-based protocols such as TCP, UDP, and SCTP in orbit[34]. Following the UWE-1 mission, a significant revision was made to the software system to make it more extensible and portable for the UWE-2 mission[35], which was launched in September 2009 [36].

6.10.1 UWE Platform

The system was built on an H8S microprocessor with 8MB of SRAM, 4MB of Flash, and 512KB of EEPROM memory. Micro-Linux (μCLinux) is the operating system of choice; a full version of Linux cannot be used because the CPU does not have a memory management unit (MMU).

The communication system consists of a modified COTS transceiver, which communicates using the amateur radio band to the ground station. The computer system uses a specific protocol, entitled 6Pack, to interface with the transceiver. For over-the-air communications, the AX.25 protocol is used.

The UWE-1 spacecraft also has some few fault tolerance features such as redundant modules and a watchdog timer.

6.10.2 UWE-1 Software

The UWE-1 software system mostly consisted of a simple program that executed a few critical, yet simple tasks. There were four main tasks of this application: sending beacons, collecting and transmitting sensor data, receiving commands, and charging the batteries. The software constantly iterated through these tasks for the entirety of the mission. The only deviation would be if a command were received, in which case it would execute the command, and then return to the start of the task loop again.

The UWE-1 software system was demonstrated successfully on orbit and the mission was completed after the first few weeks.

6.10.3 UWE-2 Software

When the development of UWE-2 began, the Wuerzburg team faced the challenge of the next generation of developers not being familiar with the previous design. Additionally, the UWE-1 software design was not written with extensibility in mind, and regardless of the familiarity of the new developers, it was very difficult to add features to. In order to solve this problem, the UWE-2 team decided to develop a new system with a focus on stability, modularity, and efficiency.

As a basis for the new software architecture, the ULF system was designed, which is the name for the layer that exists in user space and interfaces with the device specific software components. In order to make it simple to integrate new components to the system, it was decided to implement each aspect of the system in a different module.

At the focus of this design is the "main module", which is responsible for inter-module communication, system initialization, error handling, and direct interfacing with the hardware through the OS. During initialization, each module registers with this module, which will allow it to obtain access to various resources controlled by the main module and allows the main module to keep track of the actions of all of the modules. In addition to this tracking, the main module will check all of the running modules periodically and if a problem is detected, the system can restart that particular module.

A small set of modules were developed to cover the basic functionality of the satellite: the radio control, battery control, housekeeping, logging, and sensor

modules. The idea of including these modules in the new design is that new developers will only have to worry about creating mission specific modules for future missions.

The last aspect of the system that the team wanted to emphasize was the portability. Since the majority of the software runs in user space, there are very few hardware dependencies in the base design. The team hopes that other satellites will be able to use the ULF system because they would only have to exchange the satellite specific hardware drivers, if their platform was able to run Linux, as well.

The UWE-2 software system has also been demonstrated successfully on orbit.

6.10.4 Analysis

The approach that the Wuerzburg team took to upgrade from their first software to their second is remarkably similar to PolySat's. Both organizations had a mostly new team of developers who were looking to improve upon the old system and had a strong desire to make it more extensible. The similarities do not stop with the approach though, as both designs utilized a version of Linux that was leveraged for numerous built-in features. Due to the selection of this operating system choice, the resulting architectures were also analogous: a highly modular design that segments each primary task from one another.

The primary distinction between the two systems is choice by the UWE-2 team to develop a centralized control module for messaging and resource access control. This enables much finer grained control and logging, however, adds significant complexity and likely reimplementation of existing code. The PolySat system relies on the Linux kernel for message delivery via UDP sockets since that network stack is very reliable and robust, plus the necessary features already exist. However, some of the functionality of the UWE-2 ULF module are reimplemented in other components of the PolySat system, such as the process registration, which occurs in PolySat's software watchdog module instead.

The successes of the UWE-1 and UWE-2 software systems are great indicators for the PolySat system because it bears resemblances to both, particularly the former.

7

Related Works

A number of papers were used to aid in the design of the software architecture. These papers are summarized and their contributions to this work are detailed.

7.1 *Software Implemented Fault Tolerance: Technologies and Experience*

This oft-cited paper is one of the earliest ones to discuss a variety of software-only fault tolerance techniques such as a software watchdog, checkpointing, message logging, and recovery to said checkpoints [37]. A focus is placed on methods that are mostly transparent to the application programmers to encourage reusability and efficiency.

7.1.1 Model

The model for their system is a client-server based application that runs in a local or wide-area network of computers in a distributed system. However, it is noted these techniques apply to other types of applications, as well. The server process is the focus of most of the methods and is commonly referred to as *the application*. Furthermore, it is assumed the distributed system is in a circular configuration, such that one neighbor .

There are a few tasks in particular that this design focuses on for fault tolerance. A watchdog that runs on the primary node, watching for the application to crash or hang; in addition to a watchdog on the backup node that watches

the primary node for crashes. If the primary node crashes or hangs, a restart must be initiated on the primary node; when this is not possible, it must run on the backup one. There are also tasks responsible for periodic checkpoints of critical volatile application data, logging of client messages to the application, and replication of the application's persistent data. Moreover, there is a task dedicated to recovery of the application to the most recent checkpoint, followed by a re-execution of the message log and connecting the persistent data to the backup node, if necessary.

All of these tasks designed have been specifically chosen so that they are compatible with other fault tolerance programming methods that are typically completed within the application itself, such as N-version programming.

Four different fault tolerance levels have been designed to enable the system to select the necessary level of active protection mechanisms. The first level, zero, has none of the tasks running and thus a crash or hang may result in an unknown amount of time for a restart, depending on when it occurs. Level one adds the capability for crash detection and automatic restart, to speed up fault detection and thus recovery. In the next level, periodic checkpointing is added so that volatile data consistency can be achieved between restarts. Level three goes a step further to add nonvolatile data consistency, as well. The last state, number four, guarantees continuous operation, essentially requiring replication of the application process to insure no delays on a failure.

7.1.2 Watchdog

The watchdog, `watchd`, is implemented as a daemon process that watches the life of the application process by sending an empty signal to the it and verifying that a connection was established. If this fails, a second attempt is made after a timeout specified by the application on initialization of the watchdog. If this also fails, the process is determined to be hung and action is taken. There is also another mechanism for watching the application, which requires a heartbeat message to be sent regularly to the watchdog. Failure with this method is indicated when the heartbeat has not been received in the anticipated period. The downside of this method is that `watchd` cannot differentiate between a hung process and a very slow one.

Upon failure detection, the watchdog is responsible for recovering the applica-

tion either to its initial state, or to the last checkpoint, depending on the level of fault tolerance enabled. `Watchd` is also responsible for configuring the application to re-execute all of its messages when that feature is in use. Lastly, the watchdog watches itself and is able to recover itself from a software failure.

7.1.3 Fault Tolerance Library

The library `libft` is a set of C functions accessible to user-level applications that provide capabilities such as data checkpointing and recovery, message logging, and exception handling.

Functions are used to allow the application programmer to specify which volatile data is critical so that the library can place them in a reserved region of virtual memory and also regularly checkpoint the value. This implementation minimizes the overhead by only checkpointing the critical values, rather than all of the volatile data. In order to enable message re-execution, the fault tolerance implementation of these functions also saves the messages into a file as they are read and written by the application. The other components, like the exception handling, are implemented in macros and standard C library functions to maximize portability. The entire library has been successfully ported to various UNIX-based operating systems.

7.1.4 Multi-Dimensional File System

The multi-dimensional file system enables the replication of critical non-volatile data by allowing users to specify files that get replicated and placed on backup systems, in real time. The implementation is transparent to the application developer because it functions by intercepting file system calls and modifies their behavior to propagate the data to backup file systems. It also utilizes the `watchd` and `libft`, primarily so that failures can be detected by the watchdog and remain transparent to the user.

7.1.5 Results

By using the fault tolerance features that were designed, up to the third level of software fault tolerance was able to be provided, where detection, checkpointing, replication, and restart and recovery were all utilized. While the application was

live, an overhead of less than 14% was expected. Both the local and remote (neighboring node) detection and recovery features were successfully demonstrated.

7.1.6 Analysis

This papers goals of providing a transparent fault tolerance solution very much align with the new architecture. Some of techniques proposed have been designed into the PolySat system, particularly the software watchdog and error logging interface. The implementation of these are similar, as well: the watchdog also exists as a static process within the system and the error logging interface is provided by a common library. However, their system architecture has the capability to provide a backup watchdog on an entirely other system, but that is not feasible with a single CubeSat, at least at this time. Their library also provides a number of other features, such as checkpointing, which may be implemented in a limited sense for the PolySat system in the future, but currently does not exist in a strict sense.

A number of other worthwhile mechanisms are discussed this paper and although some rely on backup hardware, the multi-dimensional file system and checkpointing features could be considered for future revisions of this software architecture.

7.2 *Software Fault Tolerance: a Tutorial*

This tutorial for software fault tolerance was published by NASA in 2000 and covers a wide variety of fault tolerance techniques [38]. These techniques are divided into two distinct groups: single and multi-version. The single version methods are focused on because they are most relevant to this thesis, due to the significant man hours and training required to implement most multi version fault tolerance mechanisms.

7.2.1 Single-Version Software Fault Tolerance Techniques

The first primary technique for fault tolerance discussed is partitioning, which provides isolation between "functionally independent modules." The two types of partitioning are horizontal and vertical, specifying the dimension of the hierarchy

of the architecture. With horizontal partitioning, each software function is separated into independently executing modules who communicate through interfaces to "control modules" that are solely responsible for coordinating the execution of these functions. The top-down hierarchy of a vertically partitioned architecture places most of the control functions at the highest level and most of the actual processing occurs at the low level. The stated advantages of using a partitioned architecture are "simplified testing, easier maintenance, and lower propagation of side effects."

System closure is the next principle suggested, which states that all actions require explicit authorization. In order to achieve this, each component of the system must not have any more permissions than are necessary for completing its function. The reason for this policy is that errors are much more easily handled if their chance of propagating and thus creating more damage is limited. In a closed system all of the interactions are known and this makes positioning and determining the required error detection checks much simpler. On the other hand, in a more open system with respect to capabilities, a "damaged capability" can result in an unintended execution and therefore unexpected complications between modules.

The next aspect of single-version software fault tolerance approached is error detection. it's proposed that all structural modules should have two basic properties: self-protection and self-checking. A component with self-protection can detect errors in the information passed to it from other components; whereas one with self-checking can detect internal errors and prevent them from propagating to other components. In order to determine the appropriate amount of both of these properties, consideration of functionality, performance, complexity, and safety need to be made in the design process.

A number of different error detection checks are proposed for use. The checks are replication, timing, reversal, coding, reasonableness, and structural checks. Replication utilizes redundant components and compares the outputs of each to determine errors. Timing checks are relevant to modules and systems with specific timing requirements, such as deadlines. It is possible to use these known requirements for error checking; an example would be a watchdog timer. A reversal check is where the output of a component is used to compute the corresponding input, and then verify that this calculated input match the actual input. Coding

checks use redundant information included with data in order to aid with error detection. A commonly used type of coding check are checksums. The reasonableness check uses established properties of the data to detect errors, such as ranges or rates. Lastly, structural checks uses properties of the data structures for error checking. This type of check is most effective when data structures provide redundant information to enable various types of structural checks.

The final, briefly discussed, fault detection device mentioned is run-time checks. Typically, these are provided standard within hardware systems for issues such as divide by zero, overflow, and underflow.

7.2.2 Multi-version Software Fault Tolerance Techniques

Multi-version software fault tolerance is used by implementing more than one version of a certain piece of software, which can then be executed in serial or parallel, to reduce the chance of errors. There are multiple ways that this method can be used, such as an alternative configuration, and in pairs or groups. The driving force behind this fault tolerance method is that if each version of the software is implemented independently and differently, they will have different failure modes. Thus, if one implementation is prone to fail with a certain input, hopefully one of the other versions will not.

7.2.3 Applied Techniques

The entirety of the relevant techniques mentioned in this technical paper were from the single-version software fault tolerance section due to multi-version being far too cumbersome and complex to implement for the PolySat system. However, a large number of these single-version properties were designed into the system.

The basic software architecture utilizes both horizontal and vertical partitioning. It is horizontal in that each software function is distributed into its own process and communication is done via the kernel through sockets. It is also vertically partitioned in that the high level processes are fairly simple and heavily rely on underlying code, the abstraction libraries, to do a significant amount of the work.

The principle of system closure has also been used in the ground-to-spacecraft command system of the PolySat design with the addition of the signatures on

important commands to ensure that only authorized users or systems can modify the state of the spacecraft.

In regards to the error detection checks, a number of these are also being utilized. For example, the software watchdog relies on timing checks and reasonableness checks in process to ensure that the other processes execute at their known, intended rate. Furthermore, coding checks are used in various places in the system: every packet from the ground station and all UDP inter-process messages have a checksum included, also a SHA-256 hash is used for the command signature to validate data integrity, in addition to security properties.

The multi-version philosophy would ideally be applied for the software watchdog in the future because it is one of the most critical components in the system. However, the additional effort required to implement this was out of the scope and timeline of this current design.

7.3 *Flexible Fault Tolerance in Configurable Middleware for Embedded Systems*

This paper proposes the MicroQoSCORBA (MQC) middleware for embedded distributed systems with a variety of configurable fault tolerance mechanisms [39]. By providing the capabilities through a middleware, the ties to specific hardware are removed and increase the options for code reuse. Furthermore, by enabling customization of the different mechanisms, this solution can be tailored precisely to fit the needs of both the application and hardware platform.

7.3.1 Configurable Fault-Tolerant Mechanisms

The designers of MQC decided the following mechanisms would be of most use in their embedded distributed system targets: temporal redundancy, spatial redundancy, value redundancy, and reliability provided through group communication and failure detection.

The first mechanism, temporal redundancy, consists of executing an operation as many times as necessary to produce the desired result. MQC implements this for communication channels, in particular, allowing the user to specify a number of automatic retransmissions. This option is selected in the pre-configuration of

the middleware. The implementation of this mechanism also insures that only unique messages are serviced by keeping track of unique message identifiers.

The next mechanism is spatial redundancy, which is where there are several copies of the same component. This mechanism is also implemented in relation to communications, though specifically the path through which messages are transmitted. This feature enables the user to specify a variety of communication paths that are handled through the same high level abstraction and also uses the same unique message identifiers as in the temporal redundancy so that duplicate messages are ignored in a client. Unfortunately, the same is not true for the server end, where custom code would be necessary to detect duplicate messages.

The final redundancy mechanism, value, adds extra information about the data being stored or sent, typically for improvement in error detection capabilities. This mechanism has been added to the communication chain so that a checksum is calculated and included with each sent message, and validated upon each received message. This feature is also selectable at the pre-configuration stage.

The first of the reliability mechanisms is group communication, which is defined as "a means for provided multi-point to multi-point communication, by organizing processes in groups." Messages are sent to groups via the associated names and all group members receive the messages sent to it. Five different types of messaging are provided: non-uniform failure-atomic, dynamically uniform failure-atomic, FIFO ordered, causal ordered, and totally ordered multicast. The specific type of messaging can be chosen once again in the configuration of the system and are provided so that the user can customize the needs of the middleware to their specific needs.

The final mechanism, reliability via failure detection, covers a wide variety of features including any mechanism that can be used for the detection and diagnosing of failed system components. This feature is specifically provided through the capability of communication timeouts, which will allow the system to monitor the health of a communication group and remove members who are failing to communicate properly.

7.3.2 Results

A server and client was implemented with each of the mechanisms, and the memory footprint and execution times were compared to a base implementation within

a few different systems, including Linux.

Most of the overhead occurred on the server, within Linux, side: spatial redundancy approximately an additional 20 KB and the group communication nearly doubled the size, due to a custom transport implementation, while temporal and value redundancy added negligible amounts. With the client, all features required negligible additional memory except for spatial redundancy, which added almost 10 KB.

Execution times also were affected by the fault tolerance mechanisms. As one can imagine, with temporal redundancy, the execution time scaled fairly linearly with the number of additional re-transmissions. More interesting is that the spatial redundancy increased the execution time by almost 85% when using two communication channels and the value redundancy mechanism added more than 25% extra run time due to the checksum calculation. With the group communication, the execution times varied wildly based on the size of the group and the particular algorithm used, but the minimum execution overhead was shown to be more than 100%. This was explained by the sending of multiple messages to all group members and shown in the results since the times scaled approximately linearly based on the number of group members.

7.3.3 Analysis

Although the target of this middleware is distributed embedded systems in particular, many of the concepts are still applicable to independently operating CubeSats. The idea of implementing a flexible middleware is very evident in the new PolySat software architecture, although the shared libraries provide common functionality in addition to fault tolerant features, since it is intended specifically for CubeSat development. Although there are no specific capabilities like the reliable group communication mechanism mostly because Linux provides a very robust implementation of inter-process communications, the new software architecture is very amenable to such additions in the future if necessary because they can be added to the library, transparent to the user processes.

7.4 *Great Watchdogs*, Version 1.2

7.4.1 Technical Report Summary

"Great Watchdogs" is a technical report about different watchdog techniques, written by a long-time embedded engineer. He argues that watchdogs are integral parts of our system after citing aerospace projects who succeeded and failed thanks to watchdogs (or the lack thereof, respectively) [40]. He continues by plainly stating: "A Watchdog Timer (WDT) is an important line of defense in making reliable products". The bulk of the report is a discussion and survey of a number of different watchdog timer approaches: internal (to the CPU), external, and software. He also details the characteristics of "Great WDTs" and safe watchdog usage practices.

The survey of internal watchdog timers includes a number of variants from Motorola to Toshiba parts, all packaged with common microprocessors. It is argued that at bare minimum internal WDTs should be able to reset the processor. Other indicated favorable features include an external signal when watchdog resets occur, requiring multiple writes to "tap" the watchdog, and write-once watchdog control registers. The author also suggests other requirements that may be difficult to find in an internal watchdog, and thus continues to discuss external watchdogs that do provide these options.

A smaller number of external watchdogs are surveyed, but two primary variants of the external watchdog are discussed: simple and windowed. Ganssle dismisses the simpler watchdogs, which are isolated from the primary clock, yet have no strict timing requirements for being tapped. The more preferred option is the windowed external watchdog, where a reset will be triggered when the WDT input is toggled "too slowly, too fast, or not at all", which accounts for more failure modes.

Following the survey of good and bad hardware watchdogs, there is a discussion of characteristics of good watchdog timers. The author poses the CPU as a threat to the system whenever it is in a glitch state and suggests that especially watchdogs who utilize a non-maskable interrupt instead of resetting the CPU are not sufficient, due to trusting that the CPU can still execute the interrupt service routine. Furthermore, the "effective watchdog" should not be connected to the clock of the CPU either, given that this may fail and then cause both devices to

not function properly. This excludes internal watchdogs because they typically internally hard-wired to share the same clock as the CPU. The characteristics are sufficient for finding the ideal watchdog, but there likely situations where the engineer desires to utilize internal WDT on their CPU.

Also provided are a series of guidelines for use of an internal WDT, so that it provides maximum utility, despite some of the aforementioned downfalls. Two primary design rules are suggested: tap the watchdog after the code has numerous, separate verifiably correct executions, and insure that it is not possible to incidentally tap the watchdog (i.e. as a result of erroneous instruction memory). The developer is urged to be incredibly paranoid when programming with the watchdog timer by considering all possible modes of failure. Furthermore, it's recommended that any tapping of the watchdog is done inside a program's main loop rather than an ISR, which can still run while all other code has crashed. The last suggestion for using internal watchdogs is that it's important to make sure an external signal is generated upon reset, for resetting all external peripherals.

Although an internal watchdog can be useful, it's stated that the optimal watchdog does not have to rely on a processor or its software. This means that an external watchdog should be used, if it can be afforded in board space. Rather than selecting a watchdog specific component, a very cheap microcontroller can be used, which contains many built-in peripherals for constructing a very robust watchdog timer. The software on these type of watchdogs should be very simplistic and they can be interfaced with in the same manner as an internal watchdog.

Selecting watchdog hardware is very important and in an ideal situation the guidance above for tapping the watchdog would be sufficient. However, most complex embedded environments involve multitasking where the simple tapping methods are not sufficient. Thus, mechanisms specifically for multitasked environments are proposed.

First, the interactions with the watchdog must be isolated to a single task to reduce chances of unintended tapping. Inside of this task will be a data structure with an integer entry for each task. This integer needs to be incremented each time the task starts, or it can be incremented every time a task's main loop is completed, if the task is persistent. It is important that these values are incremented atomically, which can be done simply with semaphores.

Periodically, the watchdog task should scan the data structure to ensure that the values inside are reasonable (on a task-by-task basis). The rate at which the scanning occurs must be chosen based on the frequency of the tasks, to ensure that a majority of the tasks will have run in that time. To account for tasks that are not periodic, a min and max value can be used to check the value in the data structure rather than a hard-calculated value. In order to discern what min and max are appropriate, the watchdog can be configured to run in an alternate "debug" mode during testing, to observe and record the actual values in the watchdog task's data structure.

7.4.2 Watchdogs in the PolySat System

Watchdogs are a vital part of the PolySat system's fault tolerance. Both internal and external hardware watchdogs are in use, for ease of use and ensured protection, respectively. In regards to the software watchdog, the concept has drawn heavily from this technical report. The PolySat software watchdog exists in it's own process, as to be isolated from the majority of the system, and it is the only code that touches the external watchdog interface, as suggested by this technical report. Furthermore, it utilizes a very similar data structure to track the count of major events executed on each process and uses min/max values to validate these counts.

8

Conclusion

8.1 Overall Design Success

This new CubeSat software architecture has been specifically designed to provide PolySat with a system that can help the entire organization support a wide variety of missions and do it in a minimal amount of time. The use of Linux and the emphasis on modularity have been exploited since the early stages of its implementation; over ten different students have participated in developing for this software system, some of whom were only second year students. Furthermore, the use of Linux has also helped increase new mission potential, thanks to a variety of open source drivers for cameras and other payload related software that is available for Unix-type systems. This support has also been shown in this design to enable a number of features that were previously very difficult or impossible to implement, such as the digital command signing. Hopefully this is a continued trend for future revisions because there are a large number of features that can be added to the software architecture that take advantage of open source and other existing Linux resources.

The experience that has been gained during my three year tenure as a PolySat member has also been exceedingly valuable for developing this new software architecture. It has resulted in numerous contributions, such as the telemetry gathering and command capabilities, that would have been very difficult to derive as guidelines without extensive exposure to and usage of the first generation PolySat software system. It is hoped that by leveraging this experience, future members of PolySat will be encouraged to use this system and continue utilizing

it for a significant amount of time.

Lastly, it is hoped that the lifespan of the CubeSat in orbit should also be dramatically increased thanks to the new fault tolerance mechanisms. The hierarchy of watchdogs will insure that the system can recover from radiation events, preventing the system from staying in an ineffective state, and the digital command signing will prevent unauthorized access to vital commands, insuring the mission is not compromised. Although the absolute overhead of these components may seem like a lot, it has been made relatively insignificant thanks to the computing power and increased link speed capabilities of the new hardware platform. Regardless, it is a small price to pay to provide further guarantees of mission success.

8.2 Implementation Progress

The implementation of the new software architecture is currently one of the highest priorities of PolySat's software team. As of May 2011, the entire custom library has been completed and is available to developers for use on their personal computers for testing, in addition to the custom platform, when hardware interfacing is required. Additionally, all of the primary functionality of the beacon, watchdog, system manager, and communication processes have been demonstrated. The data logger process is still in progress, but minimum functionality is due to be completed by June 2011. The current goal is to have an engineering model of the system by June, which appears to be feasible at the current rate of progress.

8.3 Future Work

There is a nearly endless amount of future work thanks to the capabilities of this new platform, both in hardware and software. These concepts discussed are ones that are beneficial to the road map of PolySat and the CubeSat community as a whole.

8.3.1 Open Source Libraries

After the custom libraries are finalized, there is significant interest in open sourcing, to encourage and enable other universities to employ similar systems in their CubeSats. Both PolySat and the CubeSat community could benefit from this: PolySat could obtain more feedback from the additional users of the software, to further improve and test the libraries, and the community can leverage these libraries to form the basis of a similar platform, without necessitating the significant experience that was leveraged to design it.

8.3.2 Radio Communication Optimization

Although this design has increased the overhead of communications by utilizing UDP/IP and digital signatures, the system has also enabled various optimizations to communications that could reduce this cost. The communication process (Section 4.2.2) is readily accepting of header compression, which could potentially reduce the overhead of the UDP/IP header to a single byte, as proposed in several RFCs, such as RFC 3095 [41].

Additionally, as discussed in the design of the communication process, the AX.25 protocol has a fairly unreasonable overhead, due to a number of necessary fields that are unused or sub-optimal in satellite communications. Thus, the ideal solution would be to completely re-design the satellite communication protocol, to optimize for efficiency. The infrastructure for this type of solution already exists on both the satellite and some ground station efforts, but would need to extend to the radio transmission components, as well.

8.3.3 Scheduler Power Management

The event handler, as discussed in Section 4.3.1, is the core of all processes and forces them to block, when inactive. This feature can be readily exploited to increase the power efficiency of the processor, by entering a lower power state. Using a mechanism like Google's WakeLocks [42], the scheduler could be modified to only keep the system awake when a process is doing useful work (e.g. inside of a callback function). Preliminary investigations of the savings potential have shown the processor can save 20 mW (about 10% of idle consumption) or more

by entering a lower power state. The less power the avionics system requires, the more it can be used for running payload operations.

8.3.4 Multi-Version Software Watchdog

In order for the system to be robust to failures of the watchdog in addition to the standard processes, two software watchdogs should be running in parallel so that they can watch each other. This avoids relying on the internal watchdog to watch the software watchdog, as is done with the current system. In order to effectively implement this, an additional independent software watchdog needs to be implemented to the same specifications as the existing one. That way, both software watchdogs could watch all of the processes, including each other, and the reaction to a failure inside of the watchdog would be both more efficient and faster, as discussed in the watchdog limitations (Section 5.2.8). This is a difficult task because in order to be truly independent, the second watchdog must be developed without the utilization of the custom libraries, which is the reason it was not included for this current design.

Appendix A

PolySat Coding Standard

This is the C coding standard that has been used for the previous software system development and will be continued to be used by the new one.

Originally compiled and updated by: Kyle Leveque, Keith McCabe, Dave Cuddeback, and Jason Anderson.

A.1 Indentation

A.1.1 Number of Spaces Per Tab

There should be 4 spaces set per tab. To set this configuration in MPLab go to "Edit", "Properties...", "Tabs" and set "tab size" to 4; also set to "insert spaces".

A.1.2 Curly-Brace Location

Curly-braces should be placed at the end of the same line on which an if-statement, while, struct, etc that the brace applies to. Functions should still have the opening brace on a new line.

A.1.3 Maximum Line Length

No line should be longer than 80 columns.

A.1.4 If-Statements

One line if-statements should still be indented and also have curly-braces.

A.1.5 Break Up Code

Uses blank lines where appropriate to break up the code and group statements together.

A.2 Identifiers

A.2.1 Single Letter Variable Names

Single letter variable names are only allowed in for-loops and the only acceptable single letter variable names are i, j, and k. Additionally, i, j, and k must be used as the increment variables in for-loops.

A.2.2 Multiple Word Variable Names

In multiple word variable names, the first letter of the first word must be lowercase and all following words should begin with an uppercase letter. Underscores should not be used at any time.

A.2.3 Abbreviations

If you use an abbreviation for a term, be sure to use that same abbreviation in the rest of the code.

A.2.4 Hungarian Notation

Do not use Hungarian Notation, which prefixes all variable names with a letter representing their type.

A.2.5 Variable Name Length

Variable names should be neither excessively long nor too short; use reasonable judgment.

A.3 Constants

A.3.1 Magic Numbers

Do not use “magic numbers” in your code. A magic number is any number other than 0, 1, -1, and occasionally 2.

A.3.2 Use

Constants should be declared as “#define”s. Do not use “const” to declare constants.

A.3.3 Constant Naming

Constants should be named in all uppercase with underscores separating the words.

A.4 Source Files

A.4.1 Contents

C files should contain only functions. The only “#include” allowed is for the file’s corresponding header file.

A.4.2 Description

Each source file should include a brief description of what the source file does, any specific external documents that will be referenced in function headers and/or comments (and where to locate those documents), the author’s name, and the date the file has last been updated.

A.4.3 Function Headers

This section explains how to document C code. Always use 4 spaces instead of tabs. Line up all params and retval in a table like fashion using tabs. The whitespace distance between the name and the description should be greater than one space. Follow the following examples.

```
Last Updated:   7/19/07
                David Cuddeback
                Jason Anderson
```

This is a file comment which should be listed at the top of every file.

```
/**
 *  @file cdh-comm.c C&DH comm code.
 *
 *  This is the comm code.
 *
 *  @author Chris Noe <cnoe@calpoly.edu>
 */
```

This is a global variable and a #define comment.

```
/**
 *  This is the number of tries to deploy the antennas.
 */
```

This is a method comment.

```
/**
 *  This a brief. Sentence 2.
 *  Enable the selected comm system by setting SEL_RF
 *
 *  @bug This function does not work.
 *
 *  @deprecated This function is not Y2K.
 *
 *  @todo Fix me.
 *
 *  @param  comm    Select which comm system to enable (COMM_A_ENABLE
 *                  or COMM_B_ENABLE)
 *  @param  comm2   This is about
 *
 *  @return An error code.
```

```
 *
 *  @retval OK          Okay.
 *  @retval ERR_I2C     Error during I2C transmission.
 *
 *  @author Bill Gates <bgates@microsoft.com>
 */
```

A.5 Header Files

A.5.1 Structure

Header files should have their sections organized in the following order:

- #include's
- General #define's
- Debugging #define's
- "struct", "union", and "enum" definitions
- Global variables
- Function Prototypes

A.5.2 Header File Exclusion

Every header file should begin with:

```
#ifndef _HEADERFILENAME_H
#define _HEADERFILENAME_H
#endif
```

A.6 Comments

A.6.1 Style

All single line comments should use the "//"-style comments rather than the single lined "/* */"-style comments. Comments that are large blocks may use the "/* */"-style comments.

A.6.2 Flow Control

Every “for”, “while”, “if”, “else”, and “switch” should have a comment explaining the purpose of the control structure.

A.6.3 Microcontroller Specific Instructions

Every instruction involving a microcontroller specific instruction should have a comment explaining what is being done and possibly where the action can be referenced in an external document (such as the microcontroller’s datasheet). Some examples of microcontroller specific instructions include setting special registers, accessing pins, etc.

A.6.4 General Instructions

Every instruction or group of instructions that form a significant operation should be commented. Another programmer should be able to read through your comments without reading a single line of code and know exactly what your code does and how it does it.

A.7 Horizontal Whitespace

A.7.1 Keywords and Commas

Always put a space after each comma; never before a comma or semi-colon. Also put a space around each keyword such as “while”, “for”, and “if”.

A.7.2 Operators

Put spaces around operators when appropriate. Make large expressions as readable as possible (and parenthesis do not hurt either).

A.7.3 Between Functions

Make sure to break up functions with a couple blank lines after each function and also a space after local variable declarations.

A.7.4 Function Names

There should be no spaces between the function name and opening parenthesis, after the opening parenthesis, or before the closing parenthesis.

A.8 Functions

A.8.1 Local Variables

Always put the local variable declarations at the top of the function. The MPLab compiler makes you do this anyways, so if you manage to break this rule you get bonus points and non-compiling code. However, the VisualDSP++ compiler does let you break this rule, so watch out.

A.8.2 Error Handling

All functions should return a char. Functions return “no error” value on success and an “error code” value on failure. This is only required for functions that either access hardware or call a function that accesses hardware (recursive).

A.8.3 Parameters

The first parameters are the input variables and the last parameters are the pointers to the output variables for all functions. (Unless error handling is not required, then no output pointer variable would be needed.)

Appendix B

PolySat Formal Code Review Process

A formal code review process had been established for the previous generation flight software and will also be used for the new software. It may be revised in the future to be better suited to the new development process.

```
CPX Software Inspection Process
[adapted from Dr. John Dalbeys process]
Updated 3/28/06  Kyle & Keith
Created 5/26/04 - Kyle
```

B.1 Work Product

- Must compile without error
- Resides in Polysat/Software/Projects in repository
- Contains line numbers
- No longer than 250 lines total

B.2 Participants

- Author (Code Monkey)
- Moderator (also an inspector)

- An inspector
- Optional Additional inspectors (can also be outside of the software team)

B.3 Preparing for Inspection

- The author must alert the inspectors via email 48hrs before the code review takes place
- The inspectors will download the source code off of SVN
- Each inspector must print out all the source code in a monospaced font
- Each inspector must fill out an Inspection Preparation Log form
- Each inspector must obtain the Coding Standards
- Each inspector must also obtain any other relevant documents (datasheets, design diagrams, etc)
- Each inspector must record their starting time after completing the above items but before reviewing the actual code.
- In one continuous sitting, inspect each line of code, record any defects, and make notes of any questions or unclear code.
- Each inspector should inspect at a pace of 70 to 120 Lines of Code per hour.
- Each inspector should fill out and bring an Inspection Preparation Log to the meeting.

B.4 Meeting

- Only the author, the moderator, and the inspectors attend the meeting; no upper management.
- All inspectors bring their marked source code and their Inspection Preparation Log
- Time limit is 2 hours.

- Purpose is to identify defects and consolidate issues; not to discuss solutions or design.
- The moderator leads the meeting; he directs attention to each section of code and solicits issues.
- Inspectors contribute issues from their Inspection Preparation Logs and those issues are evaluated to determine if they are defects or not.
- The author is only present to obtain feedback; not to present, explain, or defend code.
- The moderator and author both create an Inspection Defect Log.
- The moderators will be posted on the website; the authors will be used to make any corrections to his code.
- These defects are not entered into the defect tracking system as they have never been submitted as final in SVN.
- All marked source code from each inspector is submitted to the author (so trivial defects do not need to be logged).
- The code will be assigned by the moderator an accepted, accepted with minor rework, not accepted, or review incomplete status at the end of the meeting in the Software Inspection Report.

B.5 Rework

- There must be no severe defects, less than 3 minor defects, and less than 10 trivial defects for acceptance.
- If the product was not accepted, all defects causing a non-acceptance must be re-worked by the author and another inspection scheduled.
- If the product was accepted but a couple minor and some trivial defects remain, the author must fix those defects before committing the code to as final in SVN. However, no further reviews are required.

B.6 Follow-Up

- The moderator decides if any follow-up is necessary.
- The author lets the moderator know via email when the necessary changes have been completed. The moderator gets the latest code off of SVN and checks against the Inspection Defect Log to verify if the changes were made.
- The moderator amends the Software Inspection Report to show which defects have been corrected and to update the status of the report.

References

[1] Puig-Suari, J. and Turner, C. and R.J. Twiggs. **CubeSat: The Development and Launch Support Infrastructure for Eighteen Different Satellite Customers on One Launch**. In *Proceedings of the 15th Annual AIAA/USU Conference on Small Satellites*, 2001. 1

[2] Puig-Suari, J. and Turner, C. and Ahlgren, W. **Development of the standard CubeSat deployer and a CubeSat class PicoSatellite**. In *Aerospace Conference, 2001, IEEE Proceedings.*, **1**, pages 1/347 –1/353 vol.1, 2001. 1

[3] Dodd, P.E. and Massengill, L.W. **Basic mechanisms and modeling of single-event upset in digital microelectronics**. *Nuclear Science, IEEE Transactions on*, **50**(3):583 – 602, Jun 2003. 3, 16, 56

[4] Farkas, Jacob. **CPX: Design of a Standard CubeSat Software Bus**. http://polysat.calpoly.edu/PublishedPapers/JacobFarkas_srproj.pdf, Jun 2005. 5

[5] Noe, Chris. **Design and Implementation of the Communications Subsystem for the Cal Poly CP2 Cubesat Project**. http://polysat.calpoly.edu/PublishedPapers/ChrisNoe_srproj.pdf, Jun 2004. 5, 7, 33

[6] McCabe, Keith. **Enhancements to the CPX I2C Bus**. http://polysat.calpoly.edu/PublishedPapers/KeithMcCabe_srproj.pdf, Dec 2007. 9, 10

[7] Castello, Brian and Manyak, Greg. **PolySat: Project Update and Intern Program**, Aug 2009. 10

[8] **LightSail-1**. https://directory.eoportal.org/presentations/10002235/10002236.html. 17

[9] Daniel Walker. **Development of a CubeSat Payload to Model Particle Dampening in Space: Design and Implementation of Software for CP7**. http://digitalcommons.calpoly.edu/cpesp/23. 17, 34

[10] A. Gasperin, N. Wrachien, A. Cester, A. Paccagnella, F. Ottogalli, U. Corda, P. Fuochi, and M. Lavalle. **Total Ionizing Dose effects on 4Mbit Phase Change Memory arrays**. In *Radiation and Its Effects on Components and Systems, 2007. RADECS 2007. 9th European Conference on*, pages 1 – 8, Sep 2007. 20, 29

[11] Schwank, J.R. and Shaneyfelt, M.R. and Baggio, J. and Dodd, P.E. and Felix, J.A. and Ferlet-Cavrois, V. and Paillet, P. and Lambert, D. and Sexton, F.W. and Hash, G.L. and Blackmore, E. **Effects of particle energy on proton-induced single-event latchup**. *Nuclear Science, IEEE Transactions on*, **52**(6):2622 – 2629, Dec 2005. 20, 56

[12] Nguyen, Binh. **Linux Dictionary**. http://www.tldp.org/LDP/Linux-Dictionary/html/, Aug 2004. 23

[13] Young, Eric A. and Hudson, Tim J. **OpenSSL: The Open Source toolkit for SSL/TLS**. http://www.openssl.org. 25

[14] **Most Widely Deployed SQL Database**. http://www.sqlite.org/mostdeployed.html. 43

[15] Schneier, Bruce. **Cryptanalysis of SHA-1**. http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html, Feb 2005. 51

[16] OpenSSL, openssl@openssl.org. **OpenSSL: Documents, evp(3)**. http://www.openssl.org/docs/crypto/evp.html, Oct 2009. 52

[17] Needham, Roger M. **Denial of service**. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS '93, pages 151 – 153, New York, NY, USA, 1993. ACM. 53

[18] Montenegro, Sergio and Briess, Klaus and Kayal, Hakan. **DEPENDABLE SOFTWARE (BOSS) FOR THE BEESAT PICO SATELLITE**. http://www.montenegros.de/sergio/public/beesat_dasia2006_final.pdf, 2006. 66

[19] **BEESat-1**. http://events.eoportal.org/get_announce.php?an_id=11635, 2011. 67

[20] M. Iai, Y. Funaki, H. Yabe, K. Fujiwara, S. Masumoto, T. Usuda, S. Matunaga, J. Katoka, and T. Shima. **A PDA-Controlled Pico-Satellite, Cute-1.7, and its Radiation Protection**. In *In Proceedings of the 18th AIAA/USU Conference on Small Satellites*, Aug 2004. 67

[21] Kurtulus, C. and Baltaci, T. and Ulusoy, M. and Aydm, B.T. and Tutkun, B. and Inalhan, G. and Cetiner-Yildirim, N.L.O. and Karyot, T.B. and Yarim, C. and Edis, F.O. and Haciyev, C. and Aslan, A.R. and Unal, M.F. **iTU-pSAT I: Istanbul Technical University Student Pico-Satellite Program**. In *Recent Advances in Space Technologies, 2007. RAST '07. 3rd International Conference on*, pages 725 –732, Jun 2007. 70

[22] **ITU-pSat**. https://directory.eoportal.org/get_announce.php?an_id=10001879, 2011. 70

[23] Doering, Tyler James. **DEVELOPMENT OF A REUSABLE CUBESAT SATELLITE BUS ARCHITECTURE FOR THE KYSAT-1 SPACECRAFT**. https://archive.uky.edu/handle/10225/1012, 2009. 70

[24] **About KySat-1**. http://ssl.engineering.uky.edu/missions/orbital/kysat1/about-kysat-1/. 71

[25] **KySat-1 Launch Information**. http://ssl.engineering.uky.edu/2011/02/16/kysat-1-launch-information/. 71

[26] Pumpkin, Inc. **Salvo**. http://www.pumpkininc.com/content/doc/press/salvoflyer.pdf. 71

[27] Hishmeh, S.F. and Doering, T.J. and Lumpp, J.E. **Design of flight software for the KySat CubeSat bus**. In *Aerospace conference, 2009 IEEE*, pages 1 –15, march 2009. 71

[28] The Radio Amateur Satellite Corporation. **MEROPE**. http://www.amsat.org/amsat-new/satellites/satInfo.php?satID=86&retURL=satellites/frequencies.php. 75

[29] Larsen, B.A. and Klumpar, D.M. and Wood, M. and Hunyadi, G. and Jepsen, S. and Obland, M. **Microcontroller design for the Montana EaRth Orbiting Pico-Explorer (MEROPE) Cubesat-class satellite**. In *Aerospace Conference Proceedings, 2002. IEEE*, **1**, pages 1–487 – 1–492 vol.1, 2002. 75

[30] Mosdorf, Michael and Kurowski, Michal and Mosdorf, Lukasz and Cichocki, Andrzej and Kocon, Marcin. **PW-Sat on-board flight computer, hardware, and software design**. In *Proceedings of SPIE–the International Society for Optical Engineering*, **7502**, 2009. 77

[31] Bleier, Tom and Clarke, Paul and Cutler, Jamie and DeMartini, Louis and Dunson, Clark and Flagg, Scott and Lorenz, Allen and Tapio, Eric. **QuakeSat Lessons Learned: Notes from the Development of a Triple CubeSat**. 80

[32] **STUDSAT**. https://directory.eoportal.org/presentations/7336/10002817.html, 2010. 82

[33] Angadi, C. and Manjiyani, Z. and Dixit, C. and Vigneswaran, K. and Avinash, G.S. and Raj Narendra, P. and Prasad, S. and Ramavaram, H. and Mamatha, R.M. and Karthik, G. and Arpan, H.V. and Sharath, A.H. and Sashi Kiran, P. and Visweswaran, K. **STUDSAT: India's first student Pico-satellite project**. In *Aerospace Conference, 2011 IEEE*, pages 1 –15, Mar 2011. 83

[34] Y. Aoki, R. Barza, F. Zeiger, B. Herbst, and K. Schilling. **The CubeSat Project at the University of Wuerzburg: The Mission and**

System Design. http://www.stec2005.space.aau.dk/getpdf.php?id=107. 84

[35] Marco Schmidt and bKlaus Schilling. **An extensible on-board data handling software platform for pico satellites**. *Acta Astronautica*, **63**(11-12):1299 – 1304, 2008. 84

[36] **UWE-2 (University of Würzburg Experimentalsatellit-2)**. https://directory.eoportal.org/get_announce.php?an_id=10001331. 84

[37] Huang, Yennun and Kintala, Chanda. **FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing**. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages xxii+685, Jun 1993. 87

[38] Wilfredo, Torres. **Software Fault Tolerance: A Tutorial**. Technical report, NASA, Oct 2000. 90

[39] Dorow, K. **Flexible fault tolerance in configurable middleware for embedded systems**. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 563 – 569, Nov 2003. 93

[40] Ganssle, Jack. **Great Watchdogs**. Technical report, The Ganssle Group, Jan 2004. 96

[41] The Internet Society. **Framework and four profiles: RTP, UDP, ESP, and uncompressed**. http://www.apps.ietf.org/rfc/rfc3095.html, 2001. 101

[42] Google. **PowerManager — Android Developers**. http://developer.android.com/reference/android/os/PowerManager.html, May 2011. 101