Reliable Software Updates for On-orbit CubeSat Satellites

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical Engineering by

Sean Fitzsimmons
June 2012

COMMITTEE MEMBERSHIP

TITLE:	Reliable Software Updates for On-orbit CubeSat Satellites
AUTHOR:	Sean Fitzsimmons
DATE SUBMITTED:	June 2012
COMMITTEE CHAIR:	Dr. John Bellardo, Assistant Professor
COMMITTEE MEMBER:	Dr. Fred DePiero, Professor
COMMITTEE MEMBER:	Dr. Christopher Lupo, Assistant Professor

Abstract

Reliable Software Updates for On-orbit CubeSat Satellites

Sean Fitzsimmons

CubeSat satellites have redefined the standard solution for conducting missions in space due to their unique form factor and cost. The harsh environment of space necessitates examining features that improve satellite robustness and ultimately extend lifetime, which is typical and vital for mission success. The CubeSat development team at Cal Poly, PolySat, has recently redefined its standard avionics platform to support more complex mission capabilities with this robustness in mind. A significant addition was the integration of the Linux operating system, which provides the flexibility to develop much more elaborate protection mechanisms within software, such as support for remote on-orbit software updates.

This thesis details the design and development of such a feature-set with critical software recovery and multiple-mission single-CubeSat functionality in mind. As a result, features that focus on software update usability, validation, system recovery, upset tolerance, and extensibility have been developed. These include backup Linux kernel and file system image availability, image validation prior to boot, and the use of multiple file system devices to protect against system upsets. Furthermore, each feature has been designed for usability on current and future missions.

To my friends, fellow colleagues, and especially my family, who have made this extremely fulfilling educational journey possible.

Acknowledgements

Much of this thesis would not have been conceivable without the platform hardware designer, Austin Williams, the motivator and visionary for Cal Poly's CubeSat missions, Jordi Puig-Suari, and the unending support, knowledge, and guidance from John Bellardo. I would also like to thank the current team, previous generation members, and all those who initially helped to form this program; this thesis would not have been feasible otherwise.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
- 1.1 CubeSat . . . 1
- 1.2 PolySat . . . 2
- 1.3 Previous and New Generation Avionics Systems . . . 3
- 1.4 Robust CubeSat Design Goal . . . 5
- 1.5 Thesis Scope . . . 6

2 Development Platform Architecture **9**
- 2.1 Platform Overview . . . 10
- 2.2 Memory Organization . . . 11
- 2.3 Boot Process . . . 12
- 2.4 Limitations and Risks . . . 14

3 New Avionics Platform Architecture **19**
- 3.1 Approach . . . 20
- 3.2 Goals . . . 20
- 3.3 System Requirements . . . 21
  - 3.3.1 Non-Functional Requirements . . . 22
  - 3.3.2 Functional Requirements . . . 23
- 3.4 Memory Organization . . . 25
  - 3.4.1 Primary Boot Device . . . 25

3.4.2 Platform Overview 26
3.4.3 Multiple File Systems 27
3.4.4 Secondary File System 29
3.4.5 Development Environment Overview 32
3.5 New Boot Process 32
3.5.1 U-Boot 32
3.5.2 PCM Support 33
3.5.3 Validation 34
3.6 System State Logging 35
3.6.1 Multiple Image Support 36
3.7 Software Updates 38
3.7.1 Remote Transfer 38
3.7.2 Final Validation 39
3.8 System Limitations 40

4 Major Subsystems **43**
4.1 System State Log 43
4.1.1 Overview 43
4.1.2 Requirements 44
4.1.3 Design 45
4.1.4 Bootstrap Usage 48
4.1.5 Linux Usage 51
4.1.6 Design Success 52
4.2 Secondary File System 55
4.2.1 Overview 55
4.2.2 Requirements 55
4.2.3 Design 56
4.2.4 Directory Listings 58
4.2.5 Error Conditions 60
4.2.6 Design Success 61
4.3 Final Update Application 64
4.3.1 Overview 64
4.3.2 Requirements 64

4.3.3 Software Update Validation 65
4.3.4 System Upset Tolerance 66
4.3.5 Design Success 68

5 System Results **71**
5.1 System Success 71
5.2 Additional Results 73

6 Related Works **75**
6.1 Ionizing Radiation Tolerant Non-Volatile Memory 76
6.1.1 Phase Change Memory Overview 76
6.1.2 Attractive Features 77
6.1.3 Evaluating Upset Susceptibility 77
6.2 Important Considerations for NAND Memory 78
6.2.1 Issues with NAND Flash 78
6.2.2 YAFFS2 Utilization 79
6.3 Log-based Rollback Techniques For Error Recovery 80
6.3.1 Distributed Systems and Parallel Applications 80
6.3.2 Checkpoint-based Rollback Recovery 80
6.3.3 Log-based Rollback Recovery 81

7 Conclusion **83**
7.1 System Success 83
7.2 Current Progress 84
7.2.1 Build Integration 84
7.2.2 Test Modes 85
7.3 Future Work 86
7.3.1 Command Structure and Organization 86
7.3.2 Kernel Log Entry 86
7.3.3 Memory Write Protection 87

References **89**

List of Figures

1.1 CP4 post-PPOD deployment 2
1.2 First Generation Avionics 3

2.1 AT91SAM9G20 Platform Overview 10
2.2 Potential NAND Memory Layout 12
2.3 Internal First-Stage Bootloader Flow Diagram 15
2.4 High-Level Boot Process Flow Diagram 16

3.1 Avionics Platform Overview 26
3.2 PCM and NAND Memory Content Overview 28
3.3 Avionics NAND Memory Content 31
3.4 Avionics Boot Process Overview 35
3.5 Avionics PCM Content 37

4.1 Log Entry Types 47
4.2 General Log Entry 48
4.3 Two Possible Log Structures 49
4.4 Startup Behavior using System State Logs 50
4.5 Log Cleaning Process Flow Diagram 53
4.6 Sample Directory 57
4.7 Sample Directory with Listings 59
4.8 Sample Directory Listing File Contents 60
4.9 Validation and Recovery Process Overview 62
4.10 Software Update Application Phase 67

LIST OF FIGURES

List of Tables

1.1 Avionics Design Comparison 4

3.1 Summary of Major System Goals 21
3.2 Summary of Non-Functional Requirements 23
3.3 Summary of Functional Requirements 24

4.1 State Log Tests and Results 54
4.2 Validation and Recovery Tests and Results 63
4.3 Final Update Tests and Results 70

5.1 Desired Goals and Status 72
5.2 System Requirements and Verification 73

LIST OF TABLES

1

Introduction

1.1 CubeSat

Depending on its mission, a spacecraft may range in physical size and cost drastically. Typically, satellites serve a multitude of purposes, such as providing interactive television entertainment, or providing communication services like broadband Internet support [1]. Developed in 1999 by aerospace engineering professor Jordi Puig-Suari from Cal Poly and professor Bob Twiggs from Stanford at the time [2], the CubeSat specification completely redefined the common spacecraft model in terms of physical size, affordability, and development time. This standard presented an extremely unique hands-on opportunity for academia to develop and operate their own satellites. Due to these factors, the use of CubeSats has become widespread and continues to grow in both the industrial and educational realms today.

The CubeSat is characterized as a pico-satellite with a volume of 1000cm^3, and mass of no more than 1kg [3]. Because of these restrictions, utilizing a CubeSat's volume efficiently requires some creativity. These satellites typically use the same core hardware platform, or the same avionics system with differing payloads to support various missions (e.g., additional payload hardware to support imaging capabilities). This allows developers to reuse previously designed systems on subsequent missions, or to share designs with other developers. Throughout years of designing for various missions, developers have realized that maximizing potential payload volume is crucial as missions become more complex.

In addition to defining this specification, Cal Poly has developed a standard launch-vehicle deployer for CubeSats, the Poly Pico-satellite Orbital Deployer, or P-POD. A single P-POD can house up to three 1U, or 10x10x10cm CubeSats for deployment, as well as other varieties of this form factor (e.g., one 3U). Typically, multiple P-PODs are integrated as a secondary payload onto a launch vehicle that contains a primary payload. Combined with the CubeSat standard, the P-POD provides an attractive opportunity for people to develop and operate satellites quickly and affordably.

1.2 PolySat

PolySat, Cal Poly's CubeSat development group, began researching and designing CubeSat spacecraft shortly after the specification was defined and continues to today. This team has grown knowledgable in the field over the past decade and continues to progress in redefining mission capabilities for Cal Poly CubeSats.

To date, PolySat has developed six pico-satellites, two of which are in orbit, CP3 and CP4, and the team continues to develop for new missions annually. PolySat's previous missions, CP1 through CP6, all employed essentially the same avionics system for radio frequency (RF) communication, power, and data handling. An image of PolySat's CP4 can be seen in Figure 1.1:

Figure 1.1: CP4 post-PPOD deployment - This image of CP4 was captured by Aerospace Corporation's AeroCube-2 CubeSat shortly after P-POD deployment.

In the past year, PolySat has designed a more robust, power efficient, and computationally-capable platform to support more complex and demanding mission capabilities. Comparisons of these two avionics systems are discussed in the next section.

1.3 Previous and New Generation Avionics Systems

Prior to redesigning the avionics system, PolySat utilized a hardware platform consisting of multiple electrical boards each managing various subsystems on the spacecraft. This system contained redundant hardware for RF communication, as well as relatively straight-forward custom firmware for all its microcontrollers. Each of these microcontrollers typically managed the spacecraft's RF, command and data-handling (C&DH), or payload subsystem functionalities. A block diagram overview of this avionics architecture can be seen in Figure 1.2.

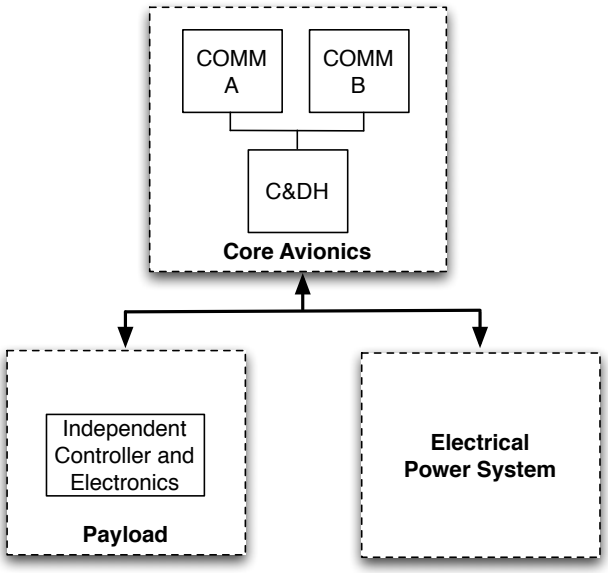

Figure 1.2: First Generation Avionics - This block diagram presents an overview of the original avionics architecture.

In summary, the C&DH is responsible for main operation of the spacecraft,

including remote command acceptance and response. The payload subsystem manages the data flow and control of specific scientific or experimental mission hardware. Although this avionics system has spaceflight heritage, it has encountered a variety of anomalies, and its use of limited volume is far from effective for currently desired payloads.

While developing the next generation avionics platform, both hardware and software redesign decisions were considered due to previously experienced limitations. First, the current hardware consumed almost a third of the CubeSat's available volume, which minimized payload potential. Secondly, the software was not developed in a modular fashion, thus making it inflexible when developing between missions. Additionally, as seen from results of the previous CPX missions [4], redundant microcontroller units further increased complexity and may potentially have caused on-orbit failures. This outweighs the benefit of what the duplicate hardware intended to provide: robustness.

To remedy these issues and also support future complex missions, the new avionics system volume was reduced considerably by integrating multiple subsystems together with a single high-performance and low-power microprocessor. As a result, the complexity incurred by utilizing redundant hardware was essentially removed. The avionics software architecture was redesigned to employ the use of the Linux operating system (OS) and to enable more flexible and extensible software development between missions. Lastly, this powerful microprocessor can potentially act as a payload controller in addition to supporting the C&DH and communication subsystems. This prevents the need to use an individual processor purely for payload control. A summary of the major differences between these avionics systems is shown in Table 1.1 [4].

Table 1.1: Avionics Design Comparison

Revision	Processor	Clock	Non-volatile Mem.	OS	Volume
1	3x PIC18	4 MHz	256 KB	Custom	0.25 L
2	AT91SAM9	400 MHz	528 MB	Linux	0.1 L

1.4 Robust CubeSat Design Goal

Hazards in the space environment can range from ionizing radiation to extreme hot and cold temperature exposure, and thus, examining features that protect the spacecraft to improve its robustness are important to mission success. In the new generation avionics, these solutions have exhibited upgrades from the previous design and now exist in both hardware and software. For example, a hardware watchdog used to passively detect improper software operation now includes a secondary long duration watchdog in case of failure. Additionally, the new avionics has incorporated a software architecture redesign, which provides the flexibility to develop more elaborate protection mechanisms that were infeasible in the past.

Since radiation exposure presents a serious problem for spacecrafts and can cause damaging effects commonly known as single-event latchups (SEL) or single-event upsets (SEU) [5], features to limit harm from its exposure have been considered. Either of these events can disrupt proper operation of the software and considering methods to reduce the risk of single or cascading system failures from such events is crucial. The redesigned software architecture now includes some solutions to handle potential radiation effects, such as a software watchdog process designed to detect basic system software anomalies and correct them. Another solution is the use of backup OS images in case of transient or permanent memory failure. The latter solution was developed as part of this thesis, and it is discussed in further detail later.

Despite risks imposed by the harsh space environment, human error will also limit mission success. This may occur due to an unforeseen circumstance not considered in the mission software development, or an inaccurate software model discovered during orbit. Currently, most flight software applications are installed a final time prior to orbit with hopes of only encountering limited (i.e., not mission jeopardizing) issues, if any, that were not discovered during integrated testing on the ground. However, this software development model is not entirely reliable and fortunately, this new platform facilitates the development of a solution to handle such potential errors. This solution is on-orbit software update functionality, which includes support for the Linux kernel, root file system, or both if necessary.

This feature will both improve the system's robustness and also provide a tremendously beneficial capability to increase mission potential: attempting to complete multiple missions rather than a single mission per developed CubeSat. Although mission development typically requires a fully tested and integrated system for a single payload experiment, this flexibility would allow multiple payloads or even a single primary payload with incomplete flight software to fly. This feature-set would only require minimal software functionality prior to flight, and final payload experiment software could be completed well after development of the CubeSat hardware.

1.5 Thesis Scope

The scope of this work includes the design and development of reliable remote on-orbit software update functionality. There are two main focuses, which include software update validation, and recovery from potentially non-functional or inoperable updates. The major goal is to extend the reliability and capabilities of PolySat's new generation avionics, both in terms of post-launch software error recovery and potential expansion of a mission's feature-set. Much of this implementation relies heavily on hardware or software architecture details of the avionics system, some of which were designed by other participating PolySat students. Their contributions and work are outlined below, as well as identified at the start of any chapter that may reference such contributions. Works related to this research have also been identified and discussed, as they have aided in the design.

This thesis contains six additional chapters, each of which is summarized below:

- Chapter 2 describes a development platform architecture used during initial avionics development, whose understanding was crucial for the design of a software update architecture. Investigation of the development platform architecture, which included obtaining information about the boot process and memory hierarchy options, was conducted by other team members including myself, Greg Manyak, and John M. Bellardo. The limitations and

risks addressed were part of an evaluation of the development board architecture performed solely as part of this thesis.

- Chapter 3 describes the avionics platform architecture as well as architecture for the software update feature-set, including validation, recovery, and an updated system boot process. The avionics platform hardware was designed by Austin Williams, which included device component selection and hardware testing to ultimately form a stable platform that could support software development. The software update architecture, which includes the integration and design of multiple software subsystems, was designed solely as part of this thesis.

- Chapter 4 details the design and implementation for critical software update subsystems or modules that are briefly discussed in Chapter 3. These three subsystems, including their design, implementation, and testing were completed solely as part of this thesis.

- Chapter 5 presents results from verification testing of the core design features, including performance measurements and requirements compliance.

- Chapter 6 evaluates works related to this research that were considered during the design phase.

- Chapter 7 concludes the thesis with a focus on the current development progress and potential future work.

2

Development Platform Architecture

While initially developing the new avionics hardware, a major amount of development support for the chosen microprocessor was obtained from a preexisting development board available from the manufacturer. This support included obtaining information about memory bus interconnects and attractive memory devices supported by the microprocessor. This information assisted in determining whether these devices should be included on the avionics. Ultimately, the new avionics platform was designed to contain several elements similar to the development platform, whose understanding was crucial to develop software update functionality. More specifically, these items include the memory device architecture and startup process, as they have implications on the design choices made in the software update system architecture.

Initial investigation of the memory device architecture, available options, and boot process on the development platform was conducted by Greg Manyak. Further investigation and evaluation of its implications on the avionics platform, as well as recommendations of how each memory device should be used on the avionics were conducted as part of this thesis. The final design decisions regarding hardware aspects for the avionics, such as specific memory component selection, were made by Austin Williams.

2.1 Platform Overview

The manufacturer of the newly chosen microprocessor, Atmel®, provides a convenient introductory development board option that was used extensively for initial avionics development. This system-on-a-chip, or SoC, the AT91SAM9G20, incorporates an ARM9 based architecture, specifically the ARM926EJ-S processor (v5), and it includes a variety of features, such as standard peripheral bus interfaces. The development unit also includes different potential primary and secondary non-volatile memory options. All of these options were heavily considered to be included on the new avionics due to their flexible use with the AT91SAM9G20, as well as individual tradeoffs. A block diagram presenting the critical elements of the AT91SAM9G20 platform architecture is presented in Figure 2.1 below [6]:

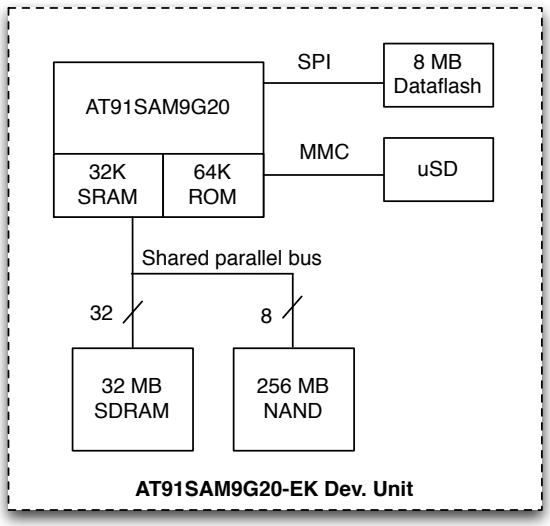

Figure 2.1: AT91SAM9G20 Platform Overview - This block diagram illustrates the various non-volatile, volatile, internal, and external memory devices and their interfaces to the avionics microprocessor.

An understanding of these memory options and how they can support various embedded applications is discussed.

2.2 Memory Organization

The number of external non-volatile memory devices available on the development platform allows for up to three primary methodologies for storing and booting Linux, as well as storing other desired data. These devices include NOR and NAND flash components, and a potential microSD card component. The physical differences and limitations between NAND and NOR flash components are important to consider, and these are described in a future section. The Atmel® NOR flash device is also known as Dataflash®, which is a specific technology name dedicated to external NOR flash devices included with the AT91 family of hardware [7].

On the AT91SAM9G20 development board, every external memory device has varying capacity and interfaces: 8MB Dataflash® with a serial peripheral interface (SPI), 256MB NAND flash with an 8-bit parallel interface, up to 32GB microSD (non-SDHC required to utilize as a boot device) with a MultiMediaCard interface (MMC), and 32MB volatile SDRAM with a standard parallel interface. Typically, one of these non-volatile external memory devices is utilized as the primary boot device, which allows for three boot device options. This boot device contains primary dependencies for Linux, as well as other startup dependencies. Any remaining non-volatile memory devices can be used for secondary or tertiary data storage.

The primary AT91 Linux support group as well as the microprocessor manual [6, 8] detail the variety of system memory and boot options available to the user, including a commonly recommended option used during early development. This option was to utilize NAND flash as the primary boot and secondary data storage device. In this configuration, the NAND flash contains all the primary Linux components, as well as other startup dependencies. A sample memory content mapping for such a configuration is shown in Figure 2.2 [8].

The primary components that make up Linux are commonly known as a system images, which are compressed files containing specific elements of the Linux OS. These main images include the kernel, which is actually the core of the OS, and secondly, what's known as the root file system. The kernel is a separate program that executes in volatile memory, and it is responsible for hardware control,

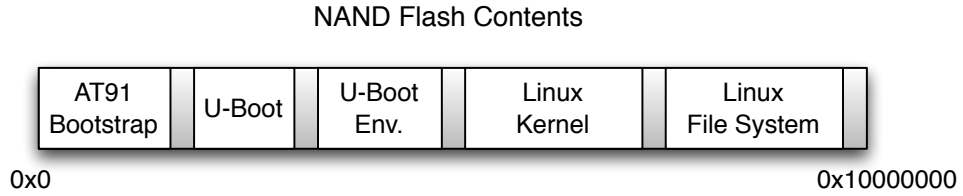

Figure 2.2: Potential NAND Memory Layout - This diagram illustrates a content mapping of NAND flash that could be used with the AT91SAM9G20.

system scheduling, and system I/O after full system startup. The root file system is actually the primary directory and file tree hierarchy that contains all the dependencies the system needs to properly run, such as other programs, libraries, and modules. The default file system utilized with this configuration is a journaling flash file system, or JFFS2 [9]. By maintaining these two primary components into separate image files, the Linux dependencies are essentially organized and simpler to program or flash onto the target hardware. For this NAND flash configuration, the kernel image size was roughly 1.5MB in size, and the initial root file system size was roughly 15MB.

In addition to these critical elements is the bootstrap program, which exists as a multi-stage program. More specifically, two of these stages are the AT91 bootstrap and U-Boot. These stages are all primary components necessary for startup and full system boot.

2.3 Boot Process

Bootloaders, or bootstrap programs are typically required with hardware that uses an operating system. They will execute as the hardware is initially powered and usually reside on a processor-internal ROM or similar device. Their primary responsibility is to perform low-level hardware initialization for devices such as memory or oscillators, which are necessary prior to loading an actual OS. In the case of the AT91SAM9G20, it contains an internal bootstrap program (a.k.a, RomBoot) on its 64KB ROM, which performs these tasks at a minimal level.

Utilizing the internal bootstrap is actually not required with the microprocessor and an external 16-bit flash memory device can be used instead. However, this requires the user to provide an independent firmware solution that will perform the necessary low-level configuration. This application must also be able to execute in place, which means it does not require content copying to RAM [10]. The option to choose between the processor's internal bootstrap or external bootstrap is hardware configurable using one of the microprocessor's external pins known as the Boot Memory Selection, or BMS pin. To avoid the inconvenience of having to develop a first-stage bootstrap solution for the development platform, the processor was configured to always boot using its preexisting internal bootstrap.

By design, this internal bootloader only acts as a first-stage program, which results in a very limited hardware configuration when done executing. This internal bootstrap can support multiple primary boot devices dynamically, and it will actually attempt to search and detect a suitable second-stage bootstrap program that can be loaded from other external memory devices. When the first-stage completes execution, control is handed to the second-stage bootstrap, which is ultimately responsible for loading the OS. By default, this second-stage bootstrap program for the development board is developed by Atmel® and known as AT91 Bootstrap.

On startup, the internal bootstrap immediately probes the SPI, parallel, and MMC buses for external memory devices that may contain a valid program executable (i.e., contains a valid ARM instruction code sequence). If one is discovered, it is copied into internal SRAM and system startup continues. Two independent 16KB banks of SRAM exist in the microprocessor which altogether may be used for the second-stage bootstrap since the internal first-stage actually executes in place via memory mapping.

After the first-stage completes scanning, if a valid bootstrap executable has not been detected or perhaps no external memory devices exist, an in-system programming utility also contained within the internal ROM is loaded. This is known as SAM-BA, and this executes and awaits any activity on the main microprocessor's USB device or debug RS232 serial port. This allows the system to be reprogrammed or perhaps troubleshooted in the event that the user may

not want to fully boot the system. A flow diagram depicting the overall behavior of the AT91 first-stage internal bootstrap is show in Figure 2.3 [6, 10].

Lastly, U-Boot is an optional third-stage bootstrap following the second-stage AT91 Bootstrap that manages other setup and passing control to the Linux OS when finished executing. U-Boot is known as a Universal Boot Loader, and it supports several other processors including many from the AT91 family. Although U-Boot's configuration, which mainly involves final steps in loading the OS, can be handled by the AT91 Bootstrap standalone, such an option was not readily available for the development board. Other information regarding Linux dependencies (e.g., kernel image address offset in NAND) that are necessary to boot exist in the U-Boot environment in memory.

A high-level diagram showing the overall flow of each of these stages in the boot process is shown in Figure 2.4 [6, 8]. Aside from the first and second stages, each stage is loaded into external SDRAM before executing, including the Linux OS.

Since multiple memory device configurations are supported by the AT91SAM9G20, the user can leverage this flexibility to design a specific startup sequence suited to a desirable application. In the case of the avionics, a few primary memory configurations were heavily considered due to their recommendations by online support [8]. This included potentially utilizing NAND flash as a primary boot and secondary data storage device. However, before finalizing the memory content layout for the avionics, a few common issues regarding these technologies were considered.

2.4 Limitations and Risks

NAND and NOR memory technology have implications on the overall functionality of the system that must be considered prior to their use. Unfortunately, they suffer from a couple disadvantages. These memories primarily differ in their internal memory cell arrangement and thus, their potential capacities and low-level interfaces differ significantly. NAND's cell structure is only limited to multiple-byte serial, rather than random single byte memory access and it can have greater density. The opposite is actually true for NOR memory.

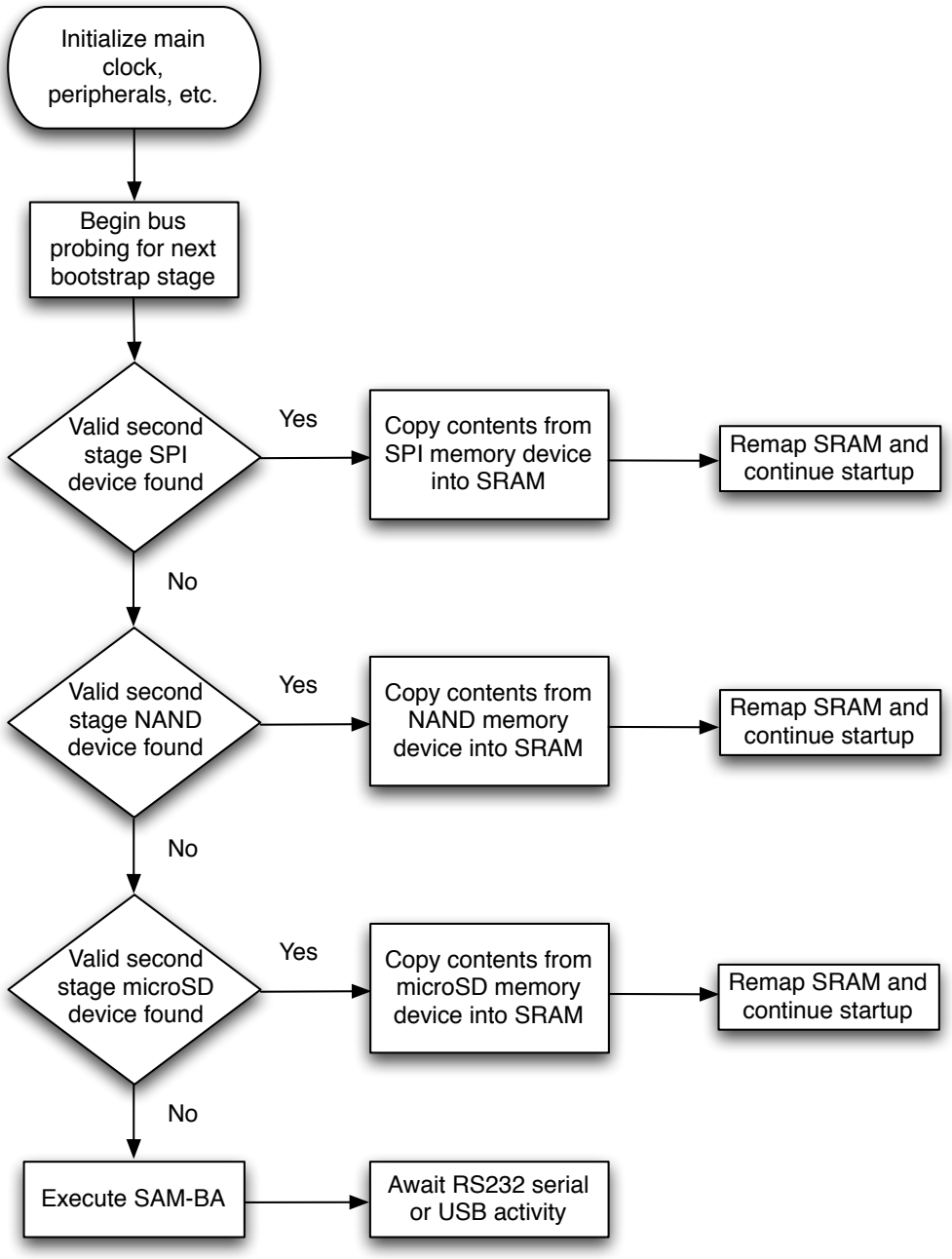

Figure 2.3: Internal First-Stage Bootloader Flow Diagram - This depicts the various buses probed by the first-stage bootloader application, which intends to load a second-stage bootstrap into internal SRAM, or load the SAM-BA in-system programming utility.

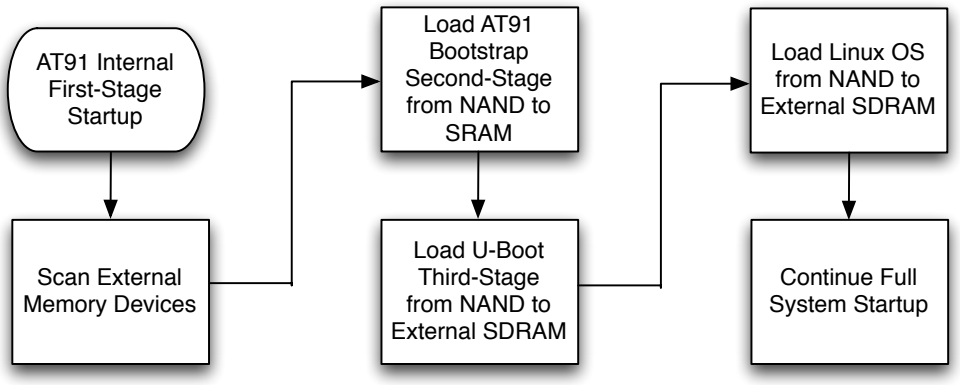

Figure 2.4: High-Level Boot Process Flow Diagram - This illustrates the various stages of the AT91 boot process.

Due to its larger capacity, NAND memory is organized into larger segments, such as pages, and these define the smallest multiple-byte read or write access portions (e.g., 1024 bytes) allowed from or to the device. Secondly, NAND's cell structure can degrade and wear-down over time due to frequent use, and this limitation must be managed to improve overall device lifetime. Managing this degradation is known as wear-leveling [11], which is a technique to handle and limit the occurrences of forming unusable memory blocks on the device. These issues are only briefly mentioned here but more detail regarding them are presented in the next chapter and related works Section 6.2.2.

The default development board architecture employs a single 256MB NAND flash device as a primary boot and secondary storage device. Given its capacity, this could be feasible for several missions with reasonable data storage requirements, but this is not an ideal memory layout. Since electronic components are susceptible to radiation induced upsets in space, relying on a single memory device to contain the primary Linux image components (i.e., Linux kernel and root file system images) for a mission is quite risky and unreliable. Additionally, this would result in all persistent memory-write activity occurring on a single device. This increases the risk of data corruption from overlapping writes, perhaps due to managing memory blocks, or failure of a memory protection mechanism within Linux. Moreover, this device could become a single point of failure for the mission

if it happens to fail entirely. Fortunately, the flexibility of the AT91SAM9G20 internal bootstrap allows for multiple memory device configurations. This has resulted in a memory hierarchy on the avionics that utilizes an independent primary boot memory device and independent secondary storage device.

Moving the critical Linux dependencies to a primary boot device does not completely solve the reliability problem since this device could also fail. However, such an event is less likely to occur on the avionics primary boot device due to its unique technology. In short, the primary boot device does not depend on standard electrical means to store bits as would a NAND device and thus, it becomes less susceptible to radiation induced upsets.

Lastly, another risk encountered by using multiple external memory devices is the increase in system complexity. By utilizing a new primary boot device, support for the device must be added to the existing AT91 Bootstrap and potentially U-Boot if not already existent. A more complicated boot process is then needed that may result in a higher potential for error occurrence.

One main advantage to multiple memory devices, however, is the separation of critical system components from a secondary storage system such as NAND. This will help to prevent potential data corruption on a single memory device. Secondly, there would no longer be a single memory device that could potentially become inoperable and end a mission completely. This memory hierarchy could tolerate a secondary storage device failure since it could now still fully startup with the primary Linux components intact on a different device. These advantages clearly outweigh the risk of the increased complexity that results from adding an independent primary boot device.

Many of these concerns, including complexity and failure modes, have been considered in the development of the software update architecture. Additionally, a firm understanding of the details regarding the existing development platform, such as boot process and memory hierarchy, was necessary to form a desirable avionics software update solution.

3

New Avionics Platform Architecture

After much consideration regarding potential memory architectures and boot methodologies for the avionics system, the platform was actually designed very similarly to its development unit predecessor. One key difference is that multiple external memory devices are utilized rather than a single NAND device. A primary memory boot device is used to store primary boot components, including Linux and its accompanying bootstrap. Since these platform architectures are so similar, understanding the development unit has aided in the design of software update features.

While considering software update functionality for this system, a few major concerns came to mind. These items include but are not limited to update validation, and potential recovery options if perhaps the update is inoperable but still used by the system. Another major consideration is a remote uplink or upload mechanism to support communication for an orbiting CubeSat. Since communication windows with the spacecraft may only last a few minutes, it may require multiple links, or 'passes' before a software update can be completely uplinked. Although important, this last consideration is only briefly discussed since multiple solutions to this problem are already available within Linux and the preexisting avionics software architecture.

At a high level, multiple modules were developed to address these concerns, one of which involves Linux image validation, either for pre-launch default system

images, or updated ones. A recovery process has also been designed to aid in system recovery if such functionality is ever necessary. The discussion of this avionics platform architecture focuses on a new memory hierarchy, boot process, and how each of the software update modules integrate within the platform. These software update modules, including system state logs, secondary file system, and final update application were developed as part of this thesis.

The choice to include multiple memory devices on the avionics unit was made collaboratively by Austin Williams and Greg Manyak. Their primary uses in comparison to the development platform architecture, including isolating a primary boot device, using multiple file systems, updating the boot procedure to support validation and recovery as a two stage process, were developed as part of this thesis.

3.1 Approach

The most desirable software update feature-set includes a reliable, robust, and extensible system and thus, a variety of design decisions were necessary to achieve these characteristics. Initially, a set of goals and requirements were established, which presented the need to develop independent major subsystems that are architected to support such functionality. Elaboration of these goals and requirements are discussed.

3.2 Goals

One main goal focuses on developing a reliable solution to update the avionics system software. When referring to the system software, this means that any critical Linux component, such as the kernel, should be updatable in addition to other system files. To facilitate this, validation and recovery mechanisms must exist.

Validation is necessary to ensure that the system can apply and use updates properly. If these updates cannot be validated, upsets on the system can result from attempting to use an update that does not contain valid data. Additionally, the system must be able to recover in case this event occurs. Software update

corruption may occur from other events, such as a bit-flip upset, so this validation should generally apply to all Linux dependencies that exist on the system, whether or not they are updates.

Since the space environment can present unpredictable resets within the system, the software update functionality should tolerate these occurrences to a reasonable degree. Minimally, any part of the software update process should not upset the system further if an unexpected reset does occur.

Another major goal is to create an extensible solution that can be employed on several future missions. Therefore, it only depends on a few major hardware requirements that will unlikely change between missions, and rather, the software has very minimal or no dependencies on any specific mission. A summary of these major goals can be found in Table 3.1.

Table 3.1: Summary of Major System Goals

Goal	Description
1	Update support for all major software components
2	Validation and any necessary recovery for all system updates
3	Reasonable tolerance of unexpected system upsets
4	Extensible solution for current and future missions

From these few major goals, a set of system requirements was generated.

3.3 System Requirements

The system requirements are organized into primary non-functional and functional requirements. The non-functional requirements refer to desired characteristics or attributes of the software update behavior as a whole, rather than what the actual system should be designed to do and perform. The latter are presented as functional requirements in a section directly following the non-functional requirements.

3.3.1 Non-Functional Requirements

The primary non-functional requirement relates directly to the avionics platform. This requirement states that the software updates should be fully compatible with the Linux distribution, software architecture, and hardware architecture on the avionics. The Linux distribution for the avionics is quite unique to the hardware, and the underlying hardware is also organized with a specific hierarchy and startup model. Unless the software update features are designed to function on the new avionics platform, it would not be feasible to remotely update it.

The next requirement states that the system should support a reusable and reliable remote uplink mechanism that can tolerate high latency. Since the communication link window is typically limited in terms of duration and number of occurrences per day, a solution to periodically collect transmitted data is required. This application or similar should validate this data and properly form it into an update when complete. It would be most beneficial if this solution could be reused on the avionics regardless of the mission that may want to support software updates.

An dual extension of the previous requirement is that first, the avionics must also contain communications hardware support in addition to a higher-level software application to manage the communication link. Secondly, there must be non-volatile memory storage available to temporarily contain software updates until they are complete and then applied to the system. Without these subsystems, which include a hardware transceiver to receive and transmit RF data packets, remote software updates to the avionics system would not be feasible.

Next, software updates for the avionics should have reasonably sized memory footprints to account for their uplink time and limited non-volatile storage. If updates were rather large in size, it may not be feasible to fit them onto an existing memory device on the system, and more time would be consumed when uplinking any updates to the system. Although a long latency software update may be tolerable, if the update is immediately critical to mission success, the avionics may not receive the update prior to a potential failure.

Furthermore, only minor changes, if any, should be required for subsequent missions utilizing this avionics platform to perform software updates. In other

words, the software update functionality should not only be usable on a single mission, but flexible enough to be used on future missions.

The last non-functional requirement states that the actual updates should have a specific format known by the system, whether that's on a per-file basis, or perhaps an entire compressed Linux image file. The software update system can thus expect a specific set of potential update formats, which is ultimately necessary to validate and apply the updates.

A summary of these non-functional system requirements can be found in Table 3.2:

Table 3.2: Summary of Non-Functional Requirements

Requirement	Description
1	Compatible with avionics software and hardware architectures
2	RF hardware support to receive and transmit packet data
3	Software support for high latency remote data transfer
4	Available non-volatile device for temporary update storage
5	Small memory footprint for software updates
6	Minor changes to support software updates for other missions
7	Specifically formatted updates

3.3.2 Functional Requirements

As for the functional requirements, the first states that the system should perform recovery in case of non-functional or corrupt software update use. Although software updates should be validated prior to the system applying them, they could still become inoperable perhaps due to transient memory failure (e.g., transfer of a complete software update between external memory devices). A system reset would most likely be required for a critical system update to take effect, such as updating a Linux component, so the avionics should recover from failure to startup with an unusable update.

Secondly, the software update system should perform low-overhead and robust validation of any software updates prior to their use. Although remotely transferred data typically utilizes protocols to perform validation on a per-packet

basis, it's appropriate to also validate the complete software update. This validation should not incur large delays on the system that may impact other components. In other words, other important subsystems may depend on fair use of the processor, and these validations should not impact their overall performance negatively.

The next functional requirement states that the software update system should tolerate unexpected global system upsets, such as unexpected reboots. If the software update application process perhaps does not complete prior to a system reset, this should not cause further errors upon system startup. Although this goes hand-in-hand with the first functional requirement, there may be other components aside from the update that are necessary to perform a software update (e.g., a request to apply the update). If any of these components become invalid due to a reset, the system should still allow for future updates and not upset the avionics on subsequent startup.

The last functional requirement states that the software update system should be able to apply the updates properly to the system when finally uplinked. If the software update functionality could not apply the software updates, the updates could not be used. A summary of these functional system requirements can be found in Table 3.3:

Table 3.3: Summary of Functional Requirements

Requirement	Description
1	System recovery for inoperable or corrupt software update use
2	Robust and low-overhead validation of software updates
3	Tolerance of unexpected system upsets during the update process
4	Application of software updates to the system

These sets of requirements have encouraged many of the architectural and design decisions for software update functionality.

3.4 Memory Organization

In comparison to the development platform architecture, the new avionics platform utilizes a memory device layout that is almost identical aside from an additional primary boot memory device. Important differences regarding actual internal memory content organization and the primary boot device technology are discussed.

3.4.1 Primary Boot Device

When deciding to use a separate primary boot device to store critical components, such as Linux, the goals were to isolate these dependencies from a secondary storage device and to reliably store them. Since the Linux images and bootstrap dependencies are vital to any mission's success, ensuring their validity on this device is essential. The main concern with choosing the proper memory device is considering its susceptibility to fluctuating radiation and temperature levels from space. If highly susceptible to upset occurrences, these critical dependencies could permanently corrupt and prevent proper system boot. Thus, a couple different solutions were explored to store these software images reliably.

The first solution involved investigating a programmable read-only memory (PROM) device, or one-time programmable read-only memory. These devices are essentially a sequence of hardcoded bits burned into silicon using fuse or anti-fuse technology to form internal electrical connectivity [12]. With this manufacturing process, the structural integrity of the circuitry is unlikely to change. Thus, they are essentially impervious to radiation induced state change like other common electrical-based memory. Upon continued research, it became impractical to consider this solution since it was expensive and not a widely available service by national manufacturers. It would have also significantly increased the avionics hardware complexity.

Next, the second and final solution was to utilize a low-cost and low-power phase-changing memory device (PCM). Although the hardware designer had already chosen to use this device on the avionics, it was not clear at the time how to use it most effectively. Further discussion of the chemical properties and the inherit robustness of this technology is presented in related works Section 6.1.1 at

the end of this paper. Due the overwhelming amount of research to commercialize this technology in a low-cost and practical form, it became feasible to integrate on the avionics platform. This device was chosen as a reliable option for storing critical data because its technology is inherently tolerant to space environment conditions.

3.4.2 Platform Overview

The avionics platform memory hierarchy has an external memory device layout containing a 16MB capacity PCM device that stores the system's primary Linux images and bootstrap components. Additionally, there are 128MB SDRAM, 512MB NAND flash, and up to 32GB MMC devices utilized. A block diagram reflecting this layout is shown in Figure 3.1, whose differences from the development platform architecture are highlighted.

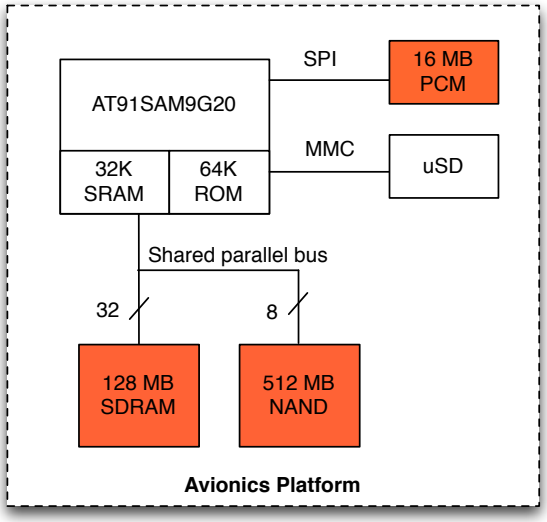

Figure 3.1: Avionics Platform Overview - This block diagram illustrates the various non-volatile, volatile, internal, and external memory devices and their interfaces to the avionics microprocessor. Differences from the development unit architecture are highlighted.

The SDRAM and NAND flash components only have minor differences from the development unit architecture, which are slight increases in memory capacity.

The avionics also supports a tertiary memory device, or uSD MMC interface for up to 32GB additional storage. For missions requiring large amounts of data acquisition, this option is readily available with a provided uSD card device.

On the development platform, the critical Linux components would be contained as two compressed kernel and root file system images in NAND flash. The total estimated memory footprint of these components is 16.5MB. In this configuration, the Linux kernel is copied to SDRAM at startup, and the root file system is simply mounted from NAND. Any file system change requests are always reflected due to the non-volatile nature of the NAND device. This startup and Linux usage has influenced a couple key design decisions of the software update architecture.

3.4.3 Multiple File Systems

On the development unit, individual memory footprints are approximately 1.5MB compressed for the kernel image, and 15MB compressed for the root file system. However, storing these Linux images similarly in the 16MB capacity PCM would not be feasible given this memory footprint. Thus, the decision to separate the larger root file system image into two independent file system images was made.

The first, the primary root file system, contains Linux dependencies that are always required for system startup and that are necessary for mission critical components. This primary file system is stored on the PCM device, and the file system type is JFFS2. The secondary file system is designed only to contain non-critical, convenient system files that should not be necessary to complete a mission. This secondary file system consists mainly of larger file system components and is stored on the NAND device as JFFS2. A sample of this memory content configuration is shown in Figure 3.2.

With the primary root file system stored on PCM, Linux has been configured to make use of this file system by copying its contents directly to SDRAM prior to using it. One key difference between this and mounting a file system directly from NAND is that any changes made to the file system are no longer persistent. In other words, changes made during a Linux session are reflected only until next system startup since the entire file system state is stored in SDRAM. Beforehand,

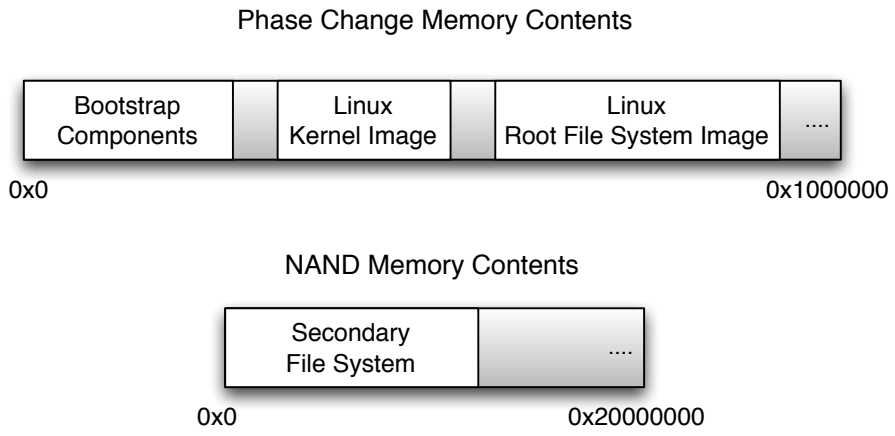

Figure 3.2: PCM and NAND Memory Content Overview - This diagram shows a sample memory content layout with Linux dependencies stored on the PCM and secondary file system stored on NAND.

the entire file system was directly mounted from the NAND device, which means its state is always stored in a non-volatile fashion.

A couple primary advantages are presented by utilizing the images this way. One of these advantages is simpler and less overhead image validation. This is due to the fact that primary file system validation does not have to be recalculated, or tracked every time there is a file system change. If the primary file system was contained within NAND, all changes to the file system would have to be tracked in order to validate the file system on a subsequent reboot. This is not only a more complex validation process, but performing it continuously can induce more unnecessary overhead in the system.

Another key advantage is that reduced memory footprints for Linux updates has resulted. More specifically, each of these file systems can now be independently updated rather than as a single large file system image. If the primary file system was stored only on NAND, smaller updates such as single files or directories could be directly uplinked rather than entire images. This behavior may seem desirable, but this requires file system changes to be actively tracked on the system. Unfortunately, this behavior is complex and undesirable.

Overall, software updates for the Linux kernel and primary root file system consist entirely of new images that can be easily validated with a single checksum.

However, in order to support validation and use of the secondary file system, a new process is necessary since any changes to the file system would persist by default. To validate and update this file system more simply, this default persistent behavior was removed, and the secondary file system is mounted read-only on startup. This file system can then be mounted with write capability only temporarily to apply an update to it. This actually makes validating changes to the secondary file system much simpler, since each update would only require changing the file system a single time, rather than continuously tracking file system changes.

3.4.4 Secondary File System

The secondary file system stored within NAND flash on the system has been designed only to contain extra dependencies rather than those necessary for proper Linux operation. Additionally, since this file system is stored on NAND flash, it can support a much larger memory footprint than one contained on PCM. The primary root file system stored on PCM has an approximate 3MB memory footprint with this dual file system setup, and the secondary file system has an approximate 25MB memory footprint.

However, this additional file system requires its own validation and potential recovery steps because it should not be used by the system if any files happen to become corrupt. In summary, this validation process is performed prior to mounting the file system and consists of validating checksums that are stored for each individual file or directory. When an update is applied to this file system, a new checksum for the updated file is generated and saved.

Before deciding to validate the file system in this manner, another option was considered. This option involved using soft-links to refer the entire secondary file system to another location in NAND, which is also known as a soft-linked file system. Soft-links are special files in Linux whose data refers to a relative or absolute path elsewhere on the system [13]. As a result, any changes requested to the secondary file system would have been reflected elsewhere in a temporary location rather than at the default paths of the secondary file system. These

changes would then have to be collected sometime, not necessarily tracked continuously, and applied to the file system in bulk. This method of validation was not used since the alternative method of mounting read-only and making single file system changes was systematically simpler.

The decision to store multiple partitions of this file system on NAND was also made to support potential recovery and multiple secondary file system versions. Five partitions of this file system exist with a max size of 32MB to allow for expanding the secondary file system size. This number of partitions was chosen to support two primary configurations. The first configuration would be two uniquely versioned, or updatable partitions, each with a recovery partition, and the second would be four uniquely versioned partitions. The first option allows for up to two unique secondary file system versions to exist, whereas the second option allows for four. Each of these options would leave room for a last partition to act as an ultimate fallback partition that cannot be updated. In order to recover a partition if it cannot be validated, another partition of the same version must exist. These are the most likely configurations foreseen that may be desired for current and future missions. If a circumstance arises where additional partitions may be needed, support for them could be added with minimal effort.

Initially, each of these file systems is a copy of the other. The only time when these are not identical is during a software update procedure when one of these is updated with new files. Thus, the other partitions initially act as a backup in case unexpected behavior results when attempting to update the currently used partition.

The rest of the capacity available in NAND is allocated as a general data partition, and this is always in read-write mode. It exists to contain any desired set of data, including data gathered from the mission and temporary storage of software updates as they are uploaded. Since uploading these updates will require numerous communication links, such a partition is necessary. The amount of storage available in this partition, roughly 350MB, will also allow for simultaneous software updates, such as simultaneous root file system and kernel image updates (i.e., uplinking one before the other is actually complete). This feature may be useful depending on mission requirements.

This data partition actually employs Yet Another Flash File System, or YAFFS2. Due to issues regarding potential NAND memory cell, or block degradation and limited read and write cycles, the decision to use this file system was made. This data partition will more than likely experience the most amount of read and write activity while storing mission data and temporary software updates. This file system is designed to efficiently utilize the NAND memory cells to extend their lifetime, and to handle any degraded block occurrences. More information regarding this file system can be found in related works Section 6.2.2.

A memory content layout of NAND to reflect these details is presented in Figure 3.3:

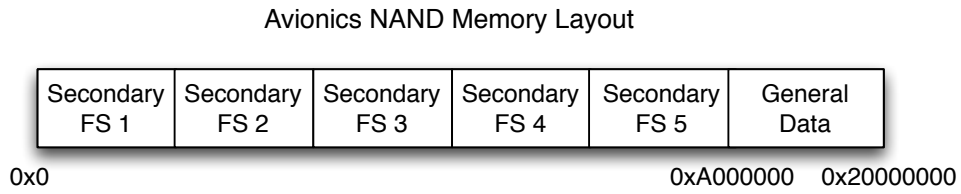

Figure 3.3: Avionics NAND Memory Content - This diagram shows the actual memory content layout of NAND on the avionics system.

By utilizing the NAND flash in this form, two primary operating modes have been established. These are known as degraded and non-degraded mode for the avionics. This NAND device does not actually have to be functional for the system to properly run given that components for Linux are stored on another device. Thus, the avionics can tolerate a NAND flash failure while still operating at a minimal, or degraded level. The non-degraded mode means both the primary boot device and secondary NAND device are functioning properly.

Fortunately, creating this multiple-image Linux environment was accomplishable by slightly tweaking an existing tool. This tool has been used throughout development of the new avionics, and it has been used to generate the development environment and Linux dependencies for this platform.

3.4.5 Development Environment Overview

The primary tool used to develop for the AT91SAM9G20 and create an environment on a host machine for cross-compilation and Linux image generation is known as buildroot [14]. In short, it supports multiple architectures and it presents a simple configuration interface for any user to begin embedded Linux development, such as on the AT91 family of hardware. Much of the dynamic configuration and build process is handled by a variety of scripts and standard Makefiles common in a Linux build environment, which were slightly modified to generate three separate primary Linux images. Buildroot will also generate a variety of dependencies, such as everything contained in a cross-compilation toolchain, expected root file system, kernel source, and extended file system for the desired target. In this case, the desired target is the AT91SAM9G20. Additionally, the target file systems are easily customizable just by modifying, adding, or removing any files on the host machine that resemble the final content of these file systems. These changes are then reflected after simply rebuilding buildroot.

3.5 New Boot Process

To further support this new memory content layout and software update functionality, an updated boot process has been put in place.

3.5.1 U-Boot

On the development unit, AT91 bootstrap's default behavior acts as a second-stage in the boot process and loads U-Boot into SDRAM. However, since the memory footprint of U-Boot and its accompanying environment were unnecessary, they were removed from the boot process. This was done to free additional memory in the PCM and to remove the overhead of using a third-stage in the boot process.

To remove U-Boot, the Linux kernel configuration that was originally supported by U-Boot needed to be added to the AT91 bootstrap. Fortunately, the AT91 bootstrap source is provided as freeware by the manufacturer, which allows for customizing it for the avionics. It's important to note the limited internal 32K

SRAM available for the AT91 bootstrap, because if this program size limitation is exceeded, the bootstrap binary cannot fully execute within the microprocessor. Fortunately, this limitation did not prevent the necessary additions that would achieve the desired boot behavior. A bit of guidance available from [15] was used to support loading Linux, which contains valuable information regarding necessary steps to boot Linux on ARM architectures.

The AT91 bootstrap sets up the required Linux tags and command-line parameters for the kernel to startup properly on the avionics. For example, one of these tags includes specifying whether an initial ram-disk for the kernel is included on an external memory device, or as part of the Linux kernel image. Either option is allowed in Linux, and the former is currently the case for the avionics.

An initial ram-disk is used as a default file system by the kernel during system startup. This file system is special since it contains dependencies that are necessary for the kernel to start important services and to complete additional system configuration. This initial ram-disk is used prior to mounting the actual primary root file system. In the case of the avionics, the initial ram-disk is identical to the primary root file-system image, so a separate initial ram-disk is not necessary to store on the PCM. This is a completely valid configuration for Linux, although it is not common to have identical initial ram-disk and root file system images.

3.5.2 PCM Support

To support interfacing with the PCM device on startup, additions to the AT91 bootstrap were necessary. Since a standard SPI interface driver already existed within the AT91 bootstrap for Dataflash®, adding support for the PCM was not difficult. The only requirements were to add a specific command-set to support the PCM, such as for reading and writing from and to the PCM device.

The internal first-stage bootstrap stored within the AT91SAM9G20 needed to be compatible with the PCM at a minimal level. This is due to the fact that the second-stage AT91 bootstrap is stored in the PCM and must be read by the first-stage. Since the first-stage bootstrap cannot be changed because it's located on an internal ROM, the basic read functionality performed by the first-stage needed to be compatible with the PCM device command-set. Fortunately, basic

read operations with SPI-based flash devices are common, including those of the PCM, so the necessary compatibility existed.

3.5.3 Validation

A major subsystem of the software update architecture is the validation process. This exists in two forms, one of which is Linux system image validation, and the other, secondary file system validation. To facilitate updating either of these components, validation is performed at different steps during the startup process.

First, validation of system images occurs while initially booting using a robust checksum algorithm and computing a checksum on each image during the bootstrap phase. By doing this at startup, system image updates can also be easily validated since they are copied to the same primary boot device before being used by the system. Performing validation at startup will detect invalid images on the PCM and prevent the system from potentially booting using invalid images. This event could be caused by invalid read operations from the PCM or a corrupt image being saved to the PCM. Although all updates transferred to the avionics will be verified, transient memory corruption can occur while applying any update from temporary NAND storage.

Fortunately, a low overhead and reliable algorithm was chosen to achieve validation, which is the MD5 cryptographic hash. This generates a 128-bit checksum regardless of the input data size and produces drastically different results with even a single bit difference. To employ this checksum, the bootstrap was customized to compute checksums on each of the Linux images before loading them into SDRAM. These checksums, image sizes, and memory offsets are compiled as part of the bootstrap each time a new set of images is programmed into the PCM.

Secondly, a validation process for the secondary file system was designed that would be executed after Linux starts. This process would run prior to mounting the secondary file system from NAND. As previously mentioned, two solutions were considered that would perform with low-overhead. The final solution that was simpler and reliable maintains an MD5 checksum for each individual file and

directory within the file system. It scans the file system hierarchy recursively ensuring each file is valid by computing its corresponding checksum and comparing.

This file system is mounted read-only such that unanticipated changes do not occur without updating any checksums for the requested file system change. To apply new updates to this file system after they are uplinked, a checksum for the corresponding change is calculated and the file system is remounted read-write to apply the change. After applying and validating the update, the file system is again remounted as read-only to prevent any further changes. If any of these steps happen to fail and recovery is necessary, other copies of the partition are used to attempt to recover any damaged or missing files.

A flow diagram showing the overall behavior of this boot process can be seen in Figure 3.4.

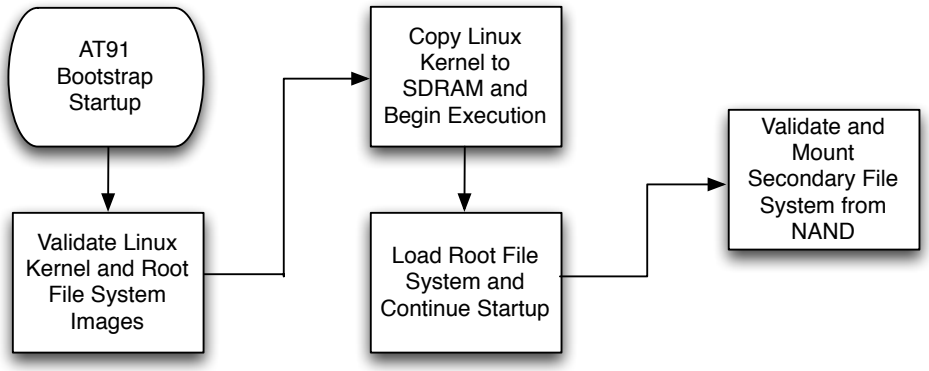

Figure 3.4: Avionics Boot Process Overview - This diagram illustrates an overview of the avionics boot process, including validation of Linux images and secondary file system.

3.6 System State Logging

Another major subsystem designed to support software update functionality is an overall system state storing mechanism. This state would be contained as a set of simple messages to facilitate things such as software update application requests, or boot attempt failure and success messages. In order to recover from a

potential boot failure, having knowledge such as the number of boot attempts and successes is critical recovery info. To essentially unapply a software update, the system must be aware that the update is non-functional. This could be discerned simply by knowing how many times the system was able to boot or not boot successfully using an update. Additionally, with this information stored as a set of messages, it would be simple to request a software update to be applied to the system, since this could be done by logging a single message.

This system state information exists in the form of a statically sized log stored in PCM. A linked-list data structure is employed in the implementation of the log to enable memory wrap-around support and cleanup of old message entries. Logs on many systems act similarly by wrapping in memory after reaching max capacity. Without performing this way, the log would have to stop collecting data until it could be cleaned by the system, and this behavior is typically not desirable.

Access to this state information would need to be available from different parts of the system, such as the bootstrap and Linux OS. Since it is stored in PCM, it can already be accessed by the bootstrap. However, support for this device did not already exist within the Linux kernel, but it was entirely feasible to add. The existing SPI device kernel driver was modified similarly to the bootstrap to support the PCM command-set. After performing the necessary modifications, the PCM could be accessed from user-space in Linux.

3.6.1 Multiple Image Support

To ultimately upload software image updates and store them in the PCM, functionality to store multiple copies of Linux images was employed. Otherwise, only a single image set (i.e., kernel and root file system) can be used at a time and an updated image set must replace the original set. This approach can be fatal to the system if an original working image set is potentially replaced with an inoperable one. This would also allow for the system to recover and fallback to a preexisting image set if a new set of images cannot be used.

Currently, there are two sets of kernel and root file system images, which are initially copies of each other until a software update replaces one of these sets.

Backup images are only used in the event of failure, or exceeding a max number of boot attempts with a specific set of images. By default, one initial kernel and root file system set is always made available for recovery, and it is not allowed to be modified. This ensures that there is always a known working set of images for recovery.

Since the PCM has a max capacity of 16 MB data storage, all of it is utilized effectively to leave storage for future software updates and to have backup Linux images available in case recovery is needed. Two copies of the system state log, which are initially identical to one other, are also stored in PCM in case the data in one happens to become corrupt due to unexpected upsets. These two logs are designed to contain the exact same sets of data and thus, they are resynchronized if any desynchronization is detected.

Figure 3.5 shows the final content memory layout of the PCM on the avionics.

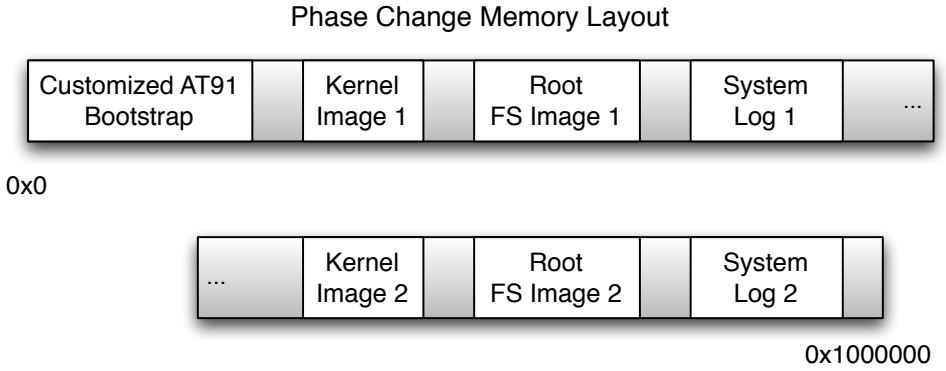

Figure 3.5: Avionics PCM Content - This shows the typical content organization of each item contained within the PCM device.

Intentional gaps of memory have been placed between each image, log, or bootstrap pieces in the PCM. This was done to reduce the likelihood of overlapping these components when perhaps a software image update is written. If this event were to occur, the overwritten data would not be usable. This extra precaution is simple and could prevent a major error from occurring, which may not even be recoverable.

3.7 Software Updates

Overall, individual steps of the validation and recovery process have been described. To fully perform a system update, it must be remotely transferred to NAND and then copied to its corresponding destination. Potential solutions for remote transfer are discussed, as well as the final validation process.

3.7.1 Remote Transfer

Supporting reliable software update transfer via RF involves several features. The majority of these include verification of individual data packets as they are sent and received between the spacecraft and ground. Additionally, these packets may be received out of order, lost or dropped and need to be re-requested, and eventually, they all must be merged into a single update file. Fortunately, utilizing Linux on the avionics provides preexisting solutions to handle this. These include standard network data transfer utilities that have been developed and tested by several other users, such as a standard file-transfer protocol, or *ftp* [16, 17, 18]. These solutions are inherently more reliable and robust than one that could be developed as part of this thesis.

Ftp achieves reliable data transfer by employing a standard networking protocol known as transmission control protocol, or TCP [19]. Although this incurs additional overhead for a satellite link, it does suffice as data transfer solution. A custom *ftp* utility is currently being developed by another PolySat student to improve data transfer performance for the link between the avionics system and the ground. This will eventually be used in place of standard *ftp*. Other utilities such as *scp* [20] can be used, which essentially uses the same protocol as *ftp* but instead with an encrypted and secure data connection.

Typical communication data rates on past PolySat missions have ranged from 1200 to 9600 baud. Although baud rate is actually the signal modulation rate for a digitally modulated transmission, the effective data rate is approximately the same given the encodings usually used for communication (e.g., non-return-to-zero, or NRZ). An estimated time for remote transfer of a 1.5MB software update at 9600 bits per second with protocol overhead, including packet loss and packet latency, would be roughly seven to ten days. This highly depends on other

factors of the communication link not considered here, such as ground station elements and orbit, but this is a reasonable estimate derived from communication experience on past satellites.

This estimation also assumes only a single ground station is available to remotely transfer a software update, which may not be the case. Several solutions are independently or collaboratively being developed to network additional ground stations across the globe to support satellite operations, such as GENSO [21]. These solutions may be widely used in the near future, which would effectively increase the overall communication link time for satellites. The resulting advantage is that software updates or other data could be transferred more quickly, which may be critical to mission success depending on the nature of the update.

3.7.2 Final Validation

The last major subsystem for software update functionality is final validation and application of the software update to the system. Final validation must be performed prior to and after transferring the completed update or unintended upsets may be introduced into the system. The primary cause of such an upset would be transient memory corruption, and thus, the data may not be written correctly to its destination or even to temporary storage.

To help prevent such an upset, the software update's MD5 checksum is transferred to the avionics prior to actually updating the system. This MD5 can be used in comparison with the transferred data and decide whether to proceed with applying it to the system. After receiving and validating this final checksum, the software update is either copied to PCM if it's intended to be a Linux update, or copied to the secondary file system if intended to be a file update. Lastly, a boot request message is placed in the system state logs so the bootstrap may load the proper image from PCM on subsequent startup.

3.8 System Limitations

Although the overall system supports all desired functionality, a few primary limitations of the software update architecture exist that should be briefly addressed.

Multiple images are supported by the software update architecture, but only one additional set. If storing more than one update for Linux was desired so they can both be readily available and used, this is currently not possible. Depending on the mission requirements, this may or may not be an ideal configuration. This is simply a limitation of the PCM capacity, and it would not be difficult to support additional sets of images in PCM with a larger capacity device. An undesirable solution to this problem could be to store multiple image updates on NAND flash and continue to apply them to the PCM when desired. However, this would require writing a new set of images each time to the PCM to reflect a new update, and this could increase the likelihood for data corruption on the device.

Secondly, there are multiple copies of each critical component contained in the PCM except for the AT91 bootstrap. Unfortunately, this is a limitation of the internal first-stage bootstrap, which is hardcoded to search for a valid second-stage bootstrap from physical address zero on all external memory devices. This is why the second-stage bootstrap starts at address zero in the PCM, but also the reason why a copy cannot be stored directly after it. If a copy of this bootstrap was located directly after the original, it would not be usable and could not be detected by the first-stage bootstrap.

The worst-case failure that could occur from this is corruption of the bootstrap, thus preventing the system from ever properly booting. One potential solution considered was to contain a copy of this second-stage bootstrap on NAND flash. Since the first-stage internal bootstrap scans multiple external memory devices at startup, the thought was that a corrupt bootstrap in PCM would be skipped and the copy on NAND flash could be used. However, this is not the case, since the first-stage internal bootstrap will still attempt to use the one provided in PCM if any remnants of it exist. To actually remove all data associated with the second-stage bootstrap, it needs to be completely removed from PCM with a sequence of erase commands. The resulting corrupted or damaged second-stage

bootstrap due to a radiation upset or overlapping memory write would most likely not resemble that of a second stage that's been completely erased (i.e., remnants of it will most likely always exist after corruption).

Because the PCM is inherently robust against radiation induced upsets, permanent corruption of the bootstrap is unlikely to occur from radiation. However, this does not prevent another subsystem from accidentally overwriting its contents. A solution to this potential scenario is to utilize the memory-protect feature available in the PCM device. This feature allows for any region of memory to be write protected if commanded, and thus, any accidental writes that would result in overwriting the bootstrap could be prevented. An implementation to support write-protecting this memory space has not yet been developed but could be an extension to this thesis.

4

Major Subsystems

The three major systems that play key roles for overall software update functionality are described in further detail. The overall design and independent verification for each subsystem are presented. The design decisions, implementation, and testing details regarding these subsystems were a major part of the work that encompasses this thesis.

4.1 System State Log

4.1.1 Overview

The system state log is a major subsystem utilized on the avionics, which is designed to support validation and recovery of critical system components. This mainly includes the Linux kernel and root file system images, but it also encompasses maintaining an overall boot state for the system. This system boot knowledge is necessary in order to decide when recovery of the system is necessary, and to request application of Linux image updates. To support system image validation, this log must also store other important Linux image and secondary file system data, such as image lengths and associated MD5s in order to validate and load images at startup.

This subsystem acts as a message-passing log that is contained within the PCM and has a static size of 128KB. More information regarding common log structures and protocols is presented in related works section 6.3. There are two

copies of the system state log, where one should be used in case the other happens to reflect an invalid state or contains improper data. Data is stored in the log in the form of separate entries or messages. These entries are designed to be collected in subsequent or appending fashion and removed or cleaned when no longer needed by the system.

4.1.2 Requirements

There are a few primary requirements regarding information that the system state log should maintain, and how it should be used.

In order for the avionics system to recover from potential error, some type of information to discern when the system should recover is necessary. In other words, the system may fail to boot by attempting to use a corrupt Linux image, or by encountering an unexpected reset prior to full startup. However, unless these failed boot attempts are tracked and acknowledged by the system, recovery from these scenarios are not possible. Storing this critical information in a reliable device such as the PCM makes the most sense, and by doing this, these logged messages can be viewed at the bootstrap or Linux OS level.

To allow for software updates to be applied to the avionics, some type of message or request to apply an update is necessary. Since all system updates must be applied on subsequent reboot of the system, the system state log should handle supporting this type of request. This would enable functionality for a desired update to be used on a subsequent system startup.

Lastly, the system state log should support storing a series of subsequent or chained messages that can be simply appended or removed when desired. This structure allows for messages to simply be scanned, added, or removed from the log as necessary. If the data was perhaps organized such that specific sections of memory were designated for storing only certain message types, the process to scan, add, and remove entries would be less flexible. With a standard log structure, any log entry type can precede the next and vice versa.

4.1.3 Design

Four types of message or log entry types have been designed for the log to support its desired functionality. The first is a boot attempt message, which simply denotes an attempt to boot using a specific set of Linux images and a secondary file system. This message type allows the system to determine when recovery is needed by simply counting boot attempts and checking whether the max number of attempts for a specific image version has been exceeded. The max number of boots attempts allowed per Linux image or secondary file system is five. Multiple versions of these system images can exist, and thus, the versions of each image used are marked accordingly in a boot attempt.

The second entry type is a boot status message, which is used to signify a successful boot for a specific Linux image or secondary file system. This boot status signifies that one of these components successfully loaded during startup. This boot message does not signify that all three components necessary for startup were successful, as the system could not discern the proper boot state if this were the case. In other words, the boot status success message must represent independent successful startup for each image type in order to determine which image or secondary file system may be corrupt or unusable. When either of these Linux components is successfully loaded, a boot status success message is placed in the log with the component's corresponding version number.

The third entry type is a file table entry, which is designed to contain information to support different versions of the Linux images and secondary root file system for software updates. Without these, software updates would not be possible. There are three file table entry types, which are Linux kernel, root file system, and secondary file system tables. Each table contains specific information for one of these dependencies, such as MD5, memory address in PCM, and so on to support multiple versions. Each file table can support information for up to seven versions of a Linux image or secondary file system. These tables are updated during a software update to reflect the new contents in memory, such as the content in PCM after a software update is applied.

The last entry type is a boot request, which is used to request that the system load a specific Linux image set and secondary file system on subsequent startup.

Since there are version numbers associated with each image and secondary file system partition, these are specified in a boot request. The bootstrap scans this boot request on the next system startup and responds by attempting to startup using the requested images. An overview of these entry types and their contents is shown below in Figure 4.1.

The boot request entry is very similar to a boot attempt, except that associated file table information regarding each image is not specified. This is not necessary since the bootstrap will just attempt to use the first encountered image descriptions from each file table that match the desired versions in a boot request entry. However, in a boot attempt, since the same version of multiple images can exist, specific information contained in the file table for the image used during startup must be noted.

The file table entries mostly contain information regarding the Linux kernel and root file system images, but only some of these fields are leveraged for the secondary file system partitions. Fields not specifically utilized in a secondary file system table are the MD5 and length fields, since one specific MD5 cannot apply to an entire secondary file system, and storing its static partition size or length is not necessary because it's already known. Compatibility modes for each image and secondary file system are also defined to ensure incompatible image combinations are not used on startup. These modes exist as a set of three (one for the Linux kernel, one of the root file system, and one for the secondary file system) minimum version numbers that are compatible with a specific image. For example, a kernel image can have a minimum version of the root file system and a minimum version of the secondary file system with which it's compatible.

A log entry has a general form that can be designated as one of these specific entries when needed. Each log entry is the same size, 256 bytes, which allows for up to 512 entries to exist per log. This size was chosen to accommodate the largest entry type and some margin in case extra fields are added in the future. The general form of a log entry is shown below in Figure 4.2.

The entry contents depend on the type of entry, whose specific fields are described in Figure 4.1. The entry MD5 is used to validate each entry prior to being used by the bootstrap. The sequence number for each entry acts as an entry counter for all entries contained in the log, which is incremented for each

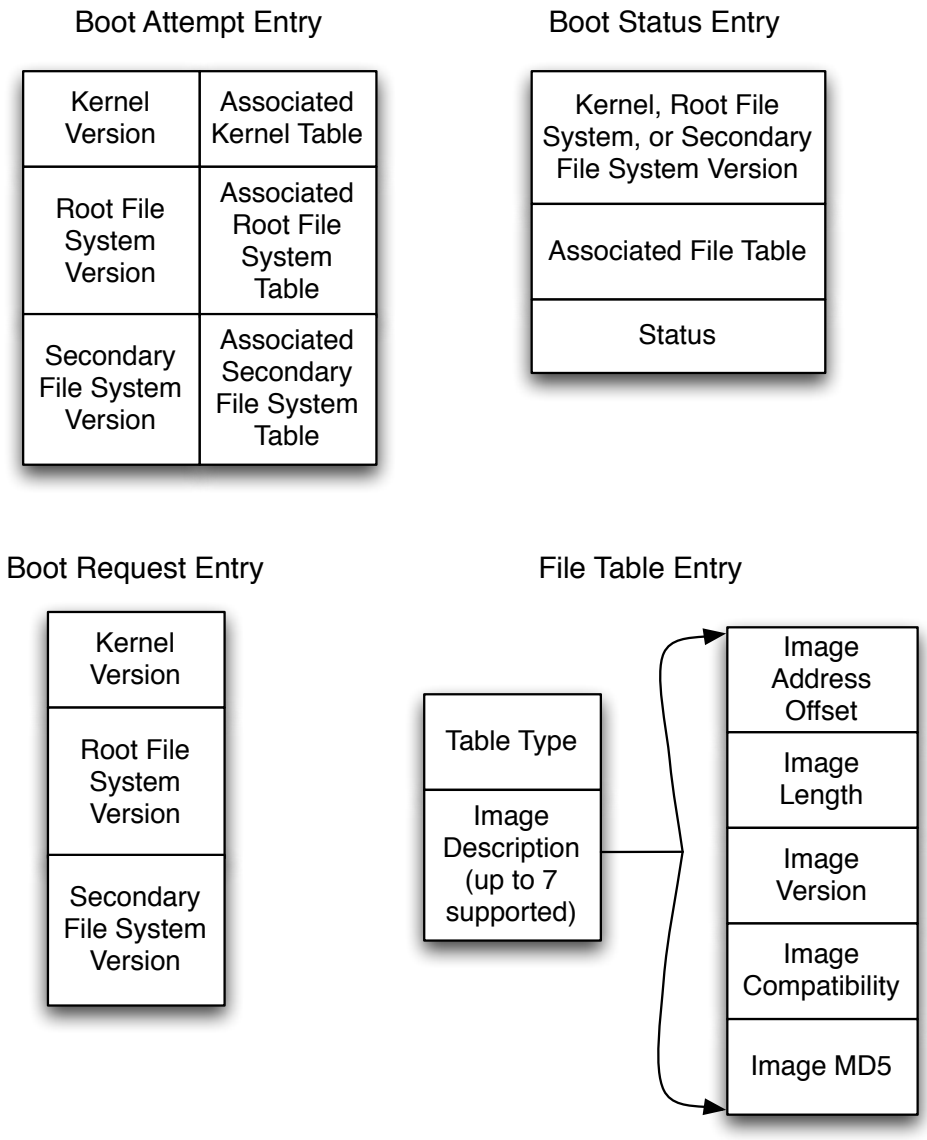

Figure 4.1: Log Entry Types - This figure overviews each log entry type and their associated fields.

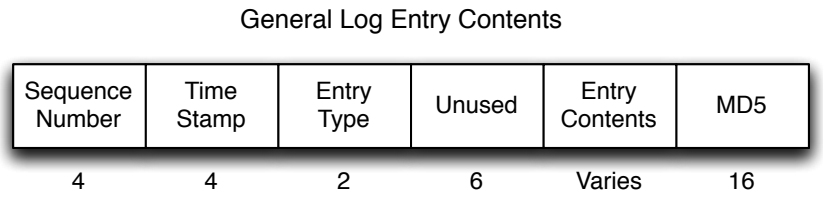

Figure 4.2: General Log Entry - This figure shows a single general log entry, and additional fields and sizes that are included in every entry.

new entry appended. New entries are only added to the end of the log, and the end is identified based on the max sequence number that exists in the log. Since the entries are organized by ascending sequence number, it's easy to determine the starting and ending entries for each log.

The log entries do not actually have to begin at the memory address starting point for each log, and they do not have to end at the designated memory address stopping point (i.e., the log memory bounds in PCM). When the log entries are scanned, a linked-list data structure is used to support this organization. Thus, the head of the entry list is wherever the lowest sequence number entry is located in memory. The tail is located wherever the highest sequence number entry is located in memory. This allows the log structure to wrap in memory such as a wrap-around buffer if needed. Two sample log structures representing two distinct possible entry organizations are shown in Figure 4.3.

4.1.4 Bootstrap Usage

In order to support using software image updates and validating them on startup, the AT91 bootstrap was further customized to scan each log for the current boot state. If any images or secondary file systems have attempted to boot and failed at least five times, the corresponding component is no longer considered valid. Thus, it is skipped until a corresponding boot status success message is placed into the log sometime in the future.

If a boot request message is encountered while scanning these logs, the requested Linux image and secondary file system versions are loaded during startup.

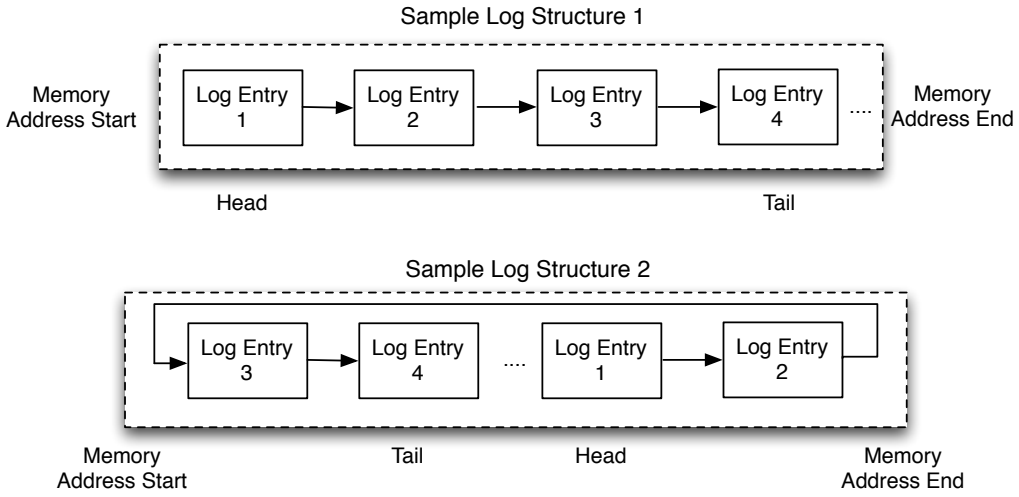

Figure 4.3: Two Possible Log Structures - This figure demonstrates two possible log structures that may exist using a linked-list data structure.

A boot attempt is written to the log as the last step prior to loading the kernel image.

Since multiple versions of the same log entry may exist, the bootstrap only utilizes the latest version of each log entry type. The latest versions will reflect the most current system state, and the latest version of a specific log entry is identified by the highest sequence number among a given set of entries of that specific type. However, all boot attempt entries are considered during startup since these are counted to ensure the number of boot attempts for a specific image set and secondary file system has not been exceeded. An overall flow diagram representing the startup behavior using these system state logs is shown in Figure 4.4.

The compatibility modes defined for each Linux image and secondary file system must be recognized, as any boot request should specify versions that are compatible. If this is not the case, the bootstrap will continue to iterate through each file table until a valid combination of these main components can be used. This new combination is chosen to resemble, by version number, the originally requested image set as much as possible (i.e., by starting at the requested version numbers and decrementing).

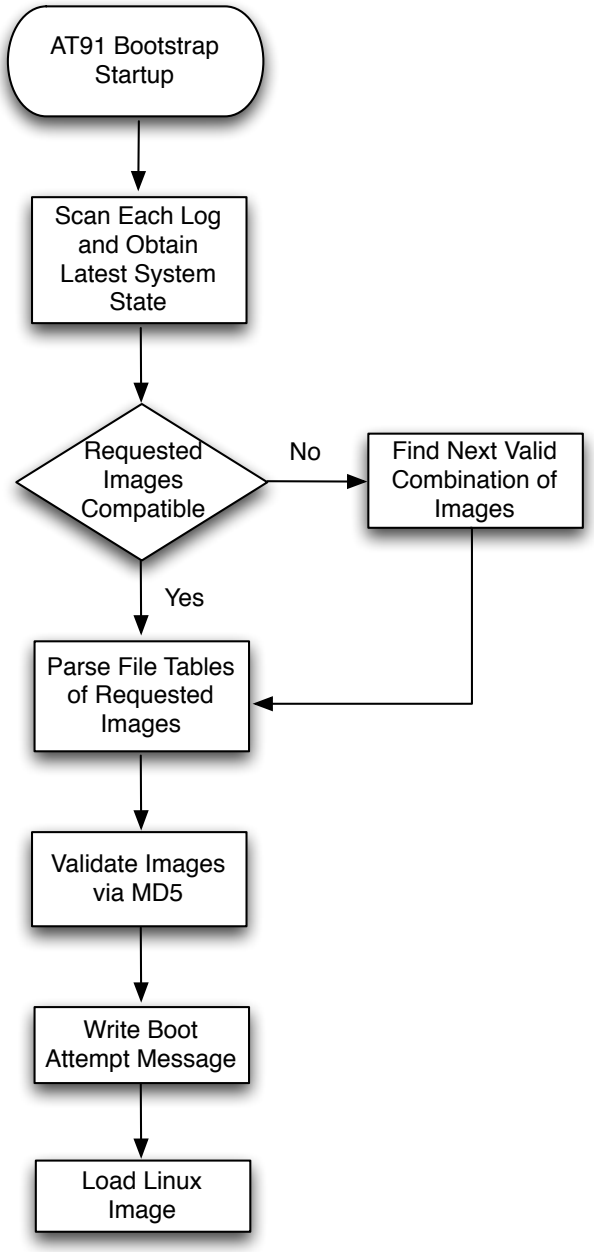

Figure 4.4: Startup Behavior using System State Logs - This flow diagram illustrates the startup behavior of the AT91 bootstrap while utilizing the system state logs.

4.1.5 Linux Usage

The system state logs are also used at the Linux level. More specifically, this is done with a user-space process to write boot status success messages, and to clean the log. Although the default SPI driver within the Linux kernel did not support the PCM, it was modified to include support for the PCM's command-set.

At startup, two boot status success entries are written for the current kernel and root file system images to signify their proper startup. These should actually be done independently rather than in unison, which is currently the case. However, kernel space support for writing a corresponding boot status success message would need to be added, and this development should be an extension to this thesis.

After writing the necessary boot status messages, the log is cleaned to remove any unnecessary entries that are no longer needed. The only entries that need to be preserved in the log are the latest versions of each entry type. For example, if multiple boot attempts were made for a specific set of Linux images, only the latest boot attempt entry needs to be preserved after the system fully boots. Additionally, if either log happens to become desynchronized from the other due to a system upset or invalid write command to the PCM, they need to be resynchronized. This resynchronization process, if needed, is also performed while cleaning the logs.

A user-space process was written to perform this cleaning as a final step at startup. In short, the process obtains the latest state from both system logs. Then, it deletes any entries that are no longer needed based on their respective sequence numbers (i.e., only entries with the highest sequence number for each type are preserved). This procedure is performed for both logs.

One thing to consider while cleaning is fragmentation of the logs. This means log entries should not be erased such that the resulting log has unused sections of memory between the head and tail entries. This is similar to fragmentation of a hard disk drive in a desktop machine that may occur from frequently deleting and writing new files. A fragmented log would result in inefficient storing of log entries, since it's structured to always append new entries. Thus, unused memory

for new entries would exist within the log itself rather than only after the tail entry.

The cleaning process is designed to detect any fragmentation that would result before cleaning any entries. If fragmentation would result from cleaning the log, all entries that should be preserved are copied to the end of the log before any entries marked for deletion are actually erased. After copying any necessary entries to the end, the entries marked for deletion are erased. An overall flow diagram representing the cleaning process is shown in Figure 4.5.

If the logs are not synchronized, the entries to preserve are copied to the end of the log prior to cleaning regardless of any detected fragmentation. The entries marked for preservation are the latest versions of each type of entry from both logs (i.e., the union of all entries from both logs). This is a simpler repair scheme than attempting to determine which specific entries are desynchronized from either log and then attempting to repair the entries individually.

4.1.6 Design Success

Overall, the system state logs and associated subprocesses such as the cleaner have been implemented to function with desired behavior.

A critical feature of the logs is enabling system recovery if it's necessary to perform at system startup. By maintaining an overall system boot state, this desired behavior is achieved. The boot attempts can be used to determine when a set of Linux images or secondary file system are not behaving properly and thus, a different version of each component can be used instead.

The other critical feature allows for application of software updates, which necessitates two distinct log entries. First, an associated file table is updated to provide necessary information for booting with a specific Linux image or secondary file system. Next, a boot request entry is written to request that the AT91 bootstrap attempt to load the desired versions of the Linux components and secondary file system on subsequent startup.

Lastly, the structure of the log is maintained as a series of subsequent entries that are ultimately appended to one another. This has resulted in the flexibility to scan, append, and clean the logs in a simple fashion.

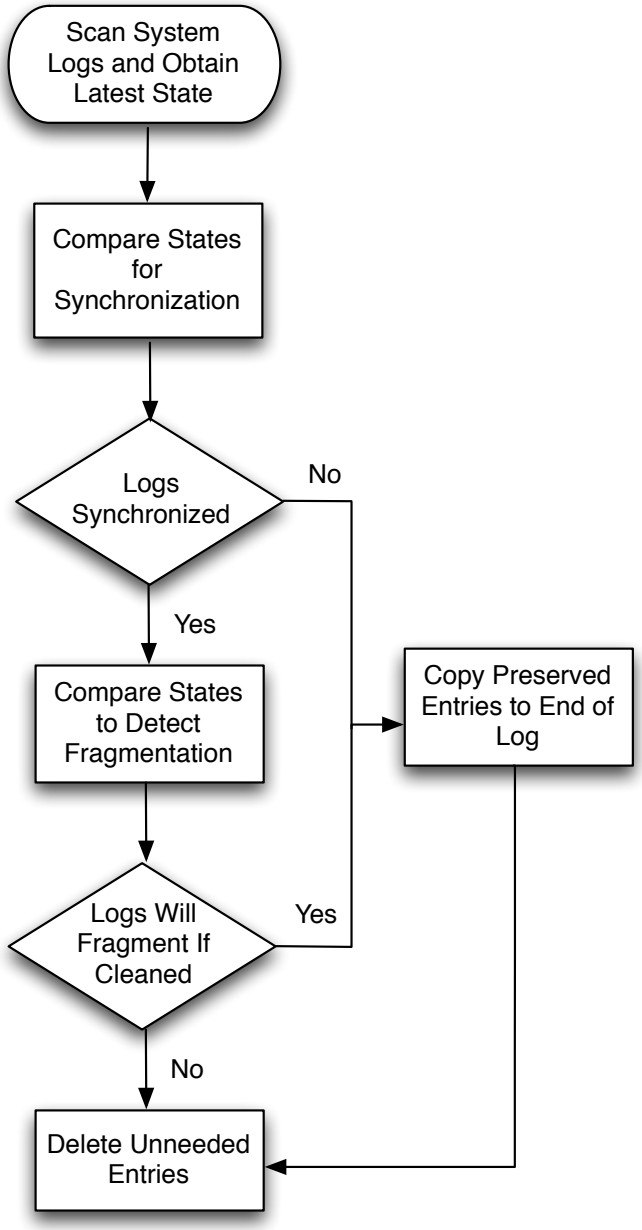

Figure 4.5: Log Cleaning Process Flow Diagram - This diagram shows the behavior of the cleaning process that's executed at system startup.

To test these primary log features, various small test cases were first conducted followed by tests that encompass functionality of all of these elements. The latter test cases are discussed. Recovery from using a requested image set on the system was conducted by manually rebooting multiple times until the max number of boot attempts was exceeded. On final startup, the bootstrap considered these images invalid and ultimately loaded fallback images. The logs were then cleaned and boot success messages were placed into the log to signify successful startup. On subsequent startup, the original images were once again considered valid and properly loaded.

The majority of testing for applying software updates was completed with the final validation and application process, which is detailed as the last subsystem in this chapter. However, to initially test functionality for file table entries and boot requests, a small utility was used to manipulate boot request and file table entries to ultimately request different boot combinations. Tests such as compatibility detection between various image sets and versions, and the ability to store multiple versions within file table entries were conducted.

Lastly, the cleaning process was tested for basic cleaning cases, and also tested to verify that system state log corruption would not result from unexpected resets while cleaning. If an entry is copied or erased during an intermittent reset, this entry will most likely be corrupt on subsequent startup. In such a case, the corrupt entry was either meant to be deleted, or it's a copy of another, so the system state is actually unaffected. The cleaning procedure is conducted in a way such that the overall system state is not lost while the procedure takes place, whether or not a reset occurs.

A summary of these tests and their results are provided in Table 4.1.

Table 4.1: State Log Tests and Results

Primary Test	Result
File table and boot request compatibility	Proper system startup
Multiversion data stored in file tables	Verified using debug output
Recovery from boot request image use	Recovered with fallback images
Desynchronized and fragmented cleaning	Proper cleanup of logs
Upset tolerance while cleaning	Successful log cleanup on next startup

4.2 Secondary File System

4.2.1 Overview

The secondary file system is used to contain non-critical components that are not required for full Linux operation. Its main purpose is to store all convenient Linux dependencies that are not necessary to store within the primary root file system. This both decreases the software image update footprint for the primary root file system, and also results in a simpler validation process for the Linux images.

This file system is stored entirely on NAND and exists as five separate partitions that may be used for performing software updates and recovery. Only four of these partitions may be updated to a newer version, which results in an updated file hierarchy. One partition is not modifiable so that it can be used as a fallback partition in case additional recovery is necessary. Recovery of the file system is only possible if a second partition of the same version exists, since the file system hierarchy state between the two need to be identical. If such a version does not exist for recovery or recovery with a specific partition fails, the fallback partition is ultimately mounted for use.

To use this file system reliably, a validation and recovery procedure was developed in the form of a Linux user-space process. This process runs after Linux startup and attempts to validate the secondary file system by comparing individually stored MD5 checksums for each file. This is done prior to initially mounting the file system so whether it's the default secondary file system or an updated one, they're always validated. If recovery is necessary due to a detected invalid file, a partition of the same version is also mounted and the corresponding file is copied for repair. If no recovery partition is available or the recovery fails, the fallback partition is mounted.

4.2.2 Requirements

In order to reliably use this file system and any updates applied to it, it should be validated and a recovery mechanism needs to exist. These are the two primary

requirements for this process, which are necessary to prevent the use of invalid updates and to recover if such an event does occur.

This process is required to use standard file system calls and services available from Linux in order to perform efficient validation and recovery of these files. This will also ensure that this process can be used in other Linux distributions if necessary, as well as future missions. Other standard system utilities should be used and executed for the recovery procedure, such as *rsync* to copy files between partitions, rather than manually copying file contents during recovery. These preexisting Linux utilities are more reliable and efficient for performing simple operations and should be used when possible.

This process should support all necessary Linux file types that may exist in the secondary file system. Currently, only regular files, directories, and symbolic links exist within this file system. These are the only file types supported by the validation and recovery process, although support for others in the future may be added.

4.2.3 Design

To prevent continuously tracking file system changes for validation, the secondary file system is mounted read-only so that it cannot be dynamically modified. If the partition was mounted read-write, a process or service would always need to be executing to detect any file system changes and to constantly recalculate validation or checksum data for the files. A simpler solution is to perform these steps on a specific partition only when a request to update that file system is made.

During a file system update, the secondary file system to be updated is mounted read-write to apply the desired update. Updates cannot be applied to the currently used partition since it may be in use by a Linux process or service. The current secondary file system is only ever mounted read-write during a recovery procedure, which would occur at initial Linux startup. After any recovery procedure is performed, the secondary file system is remounted read-only.

To validate the file system prior to mounting it at startup, an MD5 checksum is calculated for each file and compared to a known MD5 for the file. If the MD5

checksums do not match, recovery is attempted for this specific file. The MD5s of the entire file system hierarchy are pre-calculated using another utility that's executed on the entire file system prior to it being flashed to NAND.

For regular files, the corresponding MD5 checksums are calculated on the file's contents. For symbolic links, the MD5 checksum is calculated on the file path referenced by the link. This does not ensure that the link is not dangling or lost, which means the destination file being referenced by the link does not actually exist. However, the file that the link references will also be validated, and any dangling links will be detected and repaired.

Lastly, directories are unique and usually contain references to several other files contained within the directory. This is also known as a content or entry listing. For example, the data for a directory will contain a listing of files that it references, including path and other attributes of each file. A sample directory referencing three files is shown in Figure 4.6.

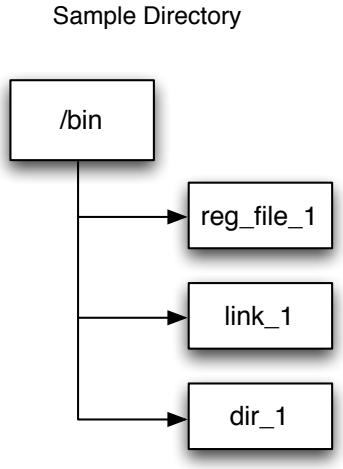

Figure 4.6: Sample Directory - This diagram shows a single directory structure with three file entries that would be contained in its directory listing.

To ensure the listing of a directory is valid on the file system, a simple checksum cannot be used. Instead, a listing file that contains the proper listing contents for each directory must be saved within the file system for validation. These directory listings are stored as hidden files in the same file system. Although only

the paths and modes for each file need to be contained within a listing to validate a directory, MD5s for each file within the directory are also recorded. This prevents the need to use another mechanism for storing the MD5 information for regular files and soft links. This information is ultimately needed to validate these files.

To support multiple versions of the secondary file system, the current version of each partition is contained in a file table log entry. This file table entry is parsed to identify other partitions that may have the same version that can be used for recovery.

Lastly, to maintain the overall state of the system, a boot success entry is saved to both system state logs once a secondary file system is mounted. This is done by the validation and recovery process as a last step prior to completing execution.

4.2.4 Directory Listings

A few more details regarding directory listings should be addressed, since these contain all critical validation information for the secondary file system.

Validation of any directory listing file must be performed before using the MD5 information stored within the actual listing. To solve this problem, a directory's corresponding listing is stored one directory above its designated directory, and an MD5 for this file is also maintained. The directory listings for the root or 'top' directories of the file system hierarchy are actually stored in the primary root file system image. A sample structure of these files for a couple directories is shown in Figure 4.7.

Each directory listing name is prefixed with *.listing.* followed by its associated directory name. For example, the listing *.listing.bin* refers to a directory listing for the directory *bin*. The directory listing and its associated directory are assumed to exist in the same directory. In the given example, two directory listings are presented, one which contains information for the entries contained in */bin*, and the second which contains information the entries contained in */bin/dir_1*.

The necessary data to store in the listing to validate a directory includes any file names referenced by the directory, and their modes. A mode of a file on

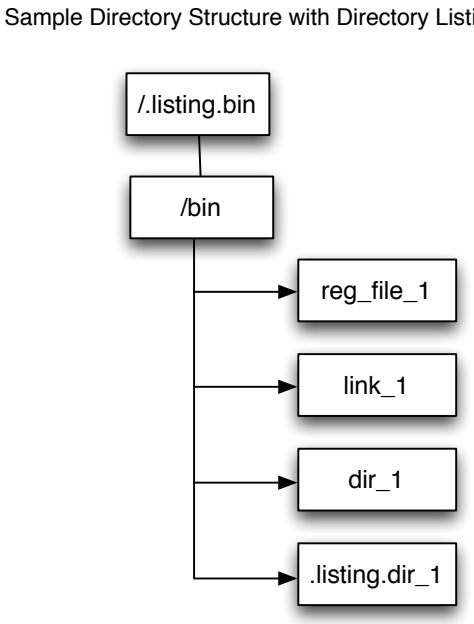

Figure 4.7: Sample Directory with Listings - This diagram shows a simple directory structure with listing files that would contain directory listing contents.

most Linux file systems consists of a permission state that defines who can and cannot access the file. In addition, the actual directory mode corresponding to a directory listing should also be saved within the listing. To further make use of this directory listing file, MD5s for regular files and soft-links are stored for their validation.

The original solution to this problem was to preserve the directory listing and associated MD5s for regular files and soft-links as separate entities. However, using a single file to contain both sets of information is a simpler solution and does not have any drawbacks from the original solution.

The mode for the directory is the first set of data in the listing file, but all data following is associated with the entries contained within the directory. This information is semicolon delimited and consists of three main parts. The first part is the file's mode, followed by the file name, and lastly its MD5. Any information regarding a subdirectory entry would be contained in its associated listing file, which should exist in the same directory. A directory listing sample that contains

data for two general file entries is shown in Figure 4.8.

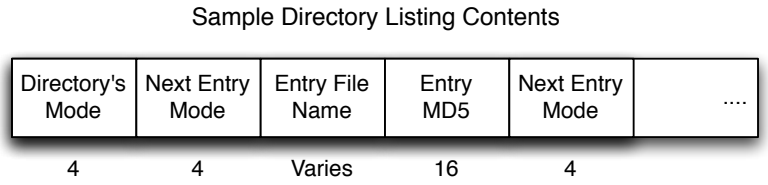

Figure 4.8: Sample Directory Listing File Contents - This diagram illustrates the content structure within the directory listing file. The size in bytes for each field is also shown.

Each of these fields is semicolon delimited to simplify parsing. One initial concern was the file system naming allowed for JFFS2, since the directory listings depend on a delimiter. JFFS2 in fact does not allow semicolons in its file names, although it may be allowed on other standard Linux file systems. Additionally, a static field size of 16 bytes is always assumed for the MD5 so that parsing it does not depend on the delimiter.

To validate the file system, a recursive tree traversal is executed starting at the root of the secondary file system. Before traversing any subdirectories, their associated directory listings are scanned and validated by comparing each file's mode, name, and MD5 for correctness. This process is repeated until all directories within the secondary file system have been traversed.

4.2.5 Error Conditions

If any errors are encountered during traversal of the secondary file system, recovery is attempted if possible. However, some errors are not recoverable, such as some encountered from Linux file system calls being used by the validation process. During traversal of the directory tree, an overall error state is kept to use for future recovery.

Errors for which recovery is necessary include invalid comparison between any file mode, name, or associated MD5 for a file. If one or more of these fields do not match, the file is flagged for recovery. If either of these errors occurs on a

regular file or soft-link, it can simply be copied from another partition of the same version for recovery.

A special case is presented when an error condition is detected on a directory listing. In this event, the associated directory contents are not scanned until the listing is recovered. If perhaps any of these errors occurs on an entire directory, an attempt to copy the entire directory contents from a recovery partition is made.

Prior to copying any of these files for recovery from a recovery partition, they are validated on their corresponding file system. These files are actually copied between partitions using a standard Linux utility known as *rsync*, which is designed to preserve the file's mode and timestamps when copied. Once all flagged files are recovered, they are again validated on the current partition. If after recovery the file system is still invalid, another attempt at recovery is made using the next available recovery partition. This process is repeated until all available recovery partitions are exhausted. If full recovery is not achieved, the fallback secondary file system partition is ultimately mounted.

Lastly, if an unexpected system reset occurs during any of these procedures, validation and recovery would resume on subsequent system startup. If a reset occurs prior to the full validation of a secondary file system, an attempt to validate it again on reboot will be made. This validation process is executed every time Linux starts until a secondary file system can be properly mounted. If a reset occurs during the recovery process, an additional recovery attempt is performed on next startup, regardless of whether the past recovery attempt completed.

An overall flow diagram illustrating the behavior of the validation and recovery process is shown in Figure 4.9.

4.2.6 Design Success

This process has been successfully implemented and tested to meet the desired requirements. Although a variety of test cases were conducted for validation, much of the testing process is similar, and this is summarized.

The primary requirements to validate and recover the secondary file system have been met by first testing functionality for each individually, and then together. The Linux process was executed and its recovery state was output for a

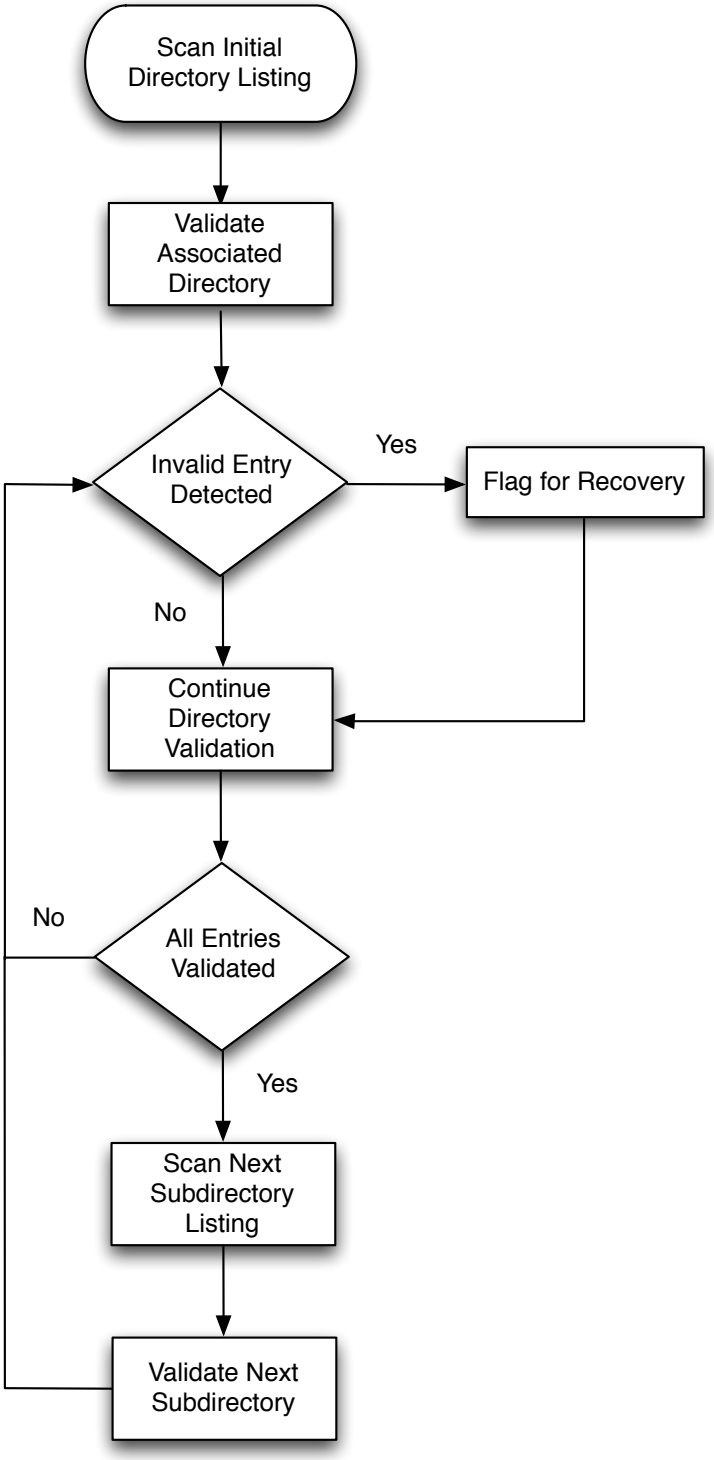

Figure 4.9: Validation and Recovery Process Overview - This flow diagram shows the overall behavior and primary subtasks of the validation and recovery process.

small file system hierarchy to verify invalid files were not detected. This same test was then conducted on a much larger file hierarchy that would actually represent the secondary file system, and eventually conducted on the secondary file system itself. In these cases, the file systems were properly validated but recover attempts were not made.

Initially, a few cases not considered were discovered during testing, and the process was remedied to account for these issues. These include multiple soft-link file dependencies, such as a link that points further down into the file system, or links that depend on other links in the system.

To verify the recovery scheme would actually recover, an invalid file was intentionally placed into the file system and valid recovery partitions were defined. The Linux process was run to attempt to replace the file from an available recovery partition. When complete, the originally invalid file was able to be revalidated, even after reboot. Lastly, recovery partitions were made unavailable when needed for recovery. As a result, the system would ultimately mount the fallback partition as expected.

An overview of the primary test cases and their results are presented in Table 4.2.

Table 4.2: Validation and Recovery Tests and Results

Primary Test	Result
Validate 10-15 entry size file system	Properly validated
Validate 800-900 entry size file system	Properly validated
Recover single file	Recovered and revalidated
Recover multiple files	Recovered and revalidated
Recovery partition unavailable	Fallback partition mounted

This process was implemented using a variety of standard I/O system calls available within Linux and additional libraries. Because of this, the validation portion of the process may be run on another host machine architecture if desired, but with a couple minor tweaks (e.g., default starting path for file system to validate may be different on another machine).

To meet the final requirement, all potential file types that may be encountered in the secondary file system are fully supported by this process. This currently includes directories, soft-links, and regular files.

4.3 Final Update Application

4.3.1 Overview

Before any updates are applied to the system, a final validation process is executed. This simply validates the full update file prior to copying it to its destination memory device. Only three update types are possible: kernel image, root file system image, or secondary file system updates. The validation and update application steps are performed using a process that is always executing on the system. The final validation and software update application sequence begins when the process is commanded remotely via RF. In order to support receiving and transmitting commands remotely, this application was integrated with a preexisting software architecture that was designed by Greg Manyak and implemented by several past students.

4.3.2 Requirements

The final validation process is rather simple and only a couple primary requirements exist. First, this process must be able to receive a valid command from the avionics RF system. The data contained with this command should consist of a file path referring to a software update file to apply, its type, a version number, an offset number, and MD5 of the update file. The secondary file system is only limited to applying updates one file at a time, and thus, must be commanded for each subsequent update desired. Additionally, a command to update the secondary file system must include a destination path to signify where to apply the update, since this can be applied anywhere in the file system.

The second requirement states that this recovery process must update the system state logs, or directory listings to reflect any applied updates. This includes updating associated file tables for any Linux image as well as for secondary

file system updates. Additionally, a boot request entry must be written to the system state logs such that a subsequent system reboot will attempt to use the applied update. This boot request must only be written if the software update is properly validated after being copied to its destination.

4.3.3 Software Update Validation

When the software update process receives a command to apply a system update, it will use the provided MD5 and compare it to one calculated on the desired update file. If the update does not exist or the update file cannot be validated, the command is not accepted and a corresponding error code is generated by the process. If the desired update file is validated, it will be copied to its destination depending on the update type.

For Linux images, this destination is the second available image-set slot in PCM, and for a secondary file system update, this is any destination path provided with the command. The destination path for a secondary file system update may or may not be a file that already exists on the file system. In case the file to update already exists on the file system, the preexisting file is simply overwritten when the update is validated and applied. Since these updates cannot be reflected immediately for either the primary Linux images or the secondary file system, a subsequent reboot is required to apply the software update.

After a Linux image is copied to the PCM, the image is validated once again by reading it from the PCM. This process is done to account for and correct any transient memory disruptions that may have ultimately corrupted the update image. Once the image is fully validated this way, its associated file table in the system state log must be updated. Since these file tables are parsed by the bootstrap upon startup, it is necessary to update them to reflect the existing system state. The latest file table entry associated with the software update is scanned, and information regarding the update image is inserted into the file table. This information includes version, memory address offset, length, and MD5. All of these fields are critical information necessary for the bootstrap to properly load the system image on subsequent system startup. Lastly, a boot

request is placed within the system state logs to request the system to use this update on next startup.

After a secondary file system update is applied, the file is validated after being copied to its designated partition in NAND. This is done similarly to a Linux image update to detect any potential memory corruption that may occur. Once the update is validated, the corresponding secondary file table entry is updated similarly to how the Linux image file tables are updated. However, the MD5 and length fields are disregarded since they are not relevant to the secondary file system. The next important step is only necessary for secondary file system updates, which requires the directory listing that contains the new update file to also be updated. This is performed by determining whether the software update is a new or preexisting file, and updating the directory listing entry accordingly. Lastly, a boot request is placed in the system state logs to request the system to mount this file system on next startup.

An overall flow diagram representing this process's behavior is shown in Figure 4.10:

A couple important considerations should be acknowledged prior to any user requesting the system to apply an update. The version number that's provided with a software update should not conflict with any preexisting version numbers for the Linux images or secondary file system partitions. In other words, system images or secondary file system partitions with the same version can exist, but any update applied to the system should reflect this properly. If two partitions are marked as the same version but contain different file tree hierarchies, this desynchronization could result in undesirable error occurrences.

4.3.4 System Upset Tolerance

To ensure the update process does not cause additional upsets in the system, its specific order of steps have been designed to prevent further upsets. In summary, these steps include copying the software update, final validation, updating a file table, and writing a boot request in this respective order. During any part of the update process, if a step happens to fail or prematurely exit due to a system reset, the system will not try to use the update.

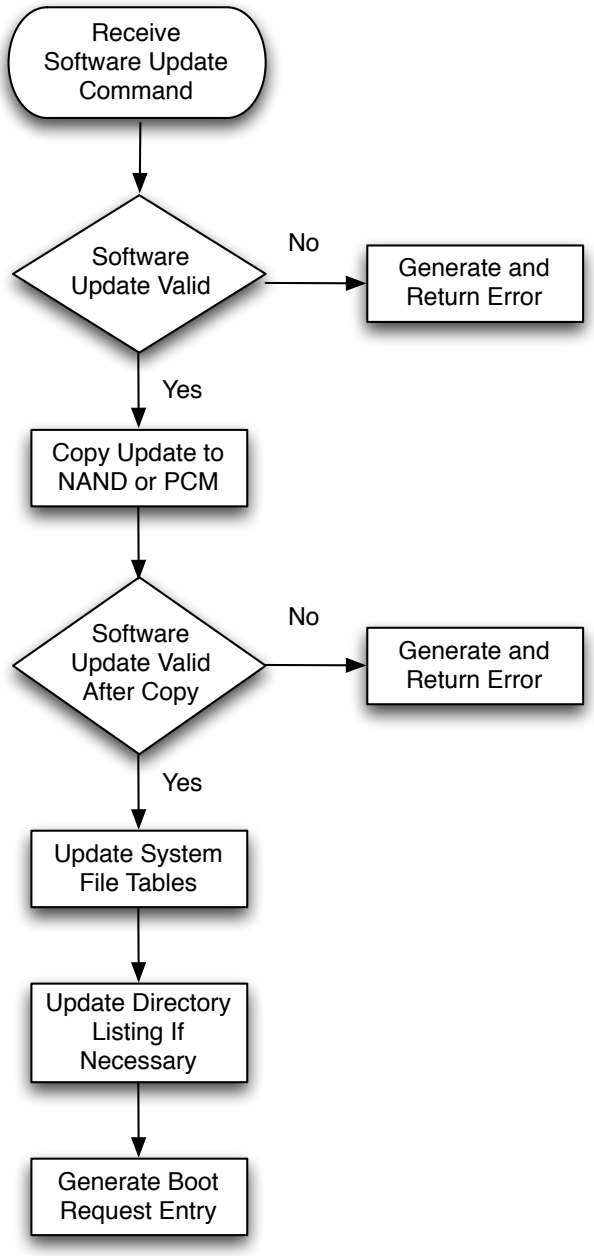

Figure 4.10: Software Update Application Phase - This diagram illustrates the validation and application steps performed by the final update Linux process.

All of these items depend on each other uniquely such that they must all exist in proper form to apply an update. More specifically, updated files tables must properly reflect the state of the Linux images or secondary file systems before a boot request can be used. Lastly, the software update content contained in PCM or NAND flash must be valid before it can be used, and this step is performed before any file tables are updated. Thus, if any part is missing that another depends on, the software update is not applied.

4.3.5 Design Success

This subsystem has been successfully implemented and tested to meet the desired requirements. A Linux process has been written to perform all necessary subtasks, some of which have been implemented with preexisting solutions that already exist in the avionics software architecture.

One main component is the ability to send and receive commands that should be transmitted over RF to enable remote commanding. Fortunately, being able to receive and send commands on the spacecraft was simply added to the process. The standard software architecture on the avionics actually uses Unix sockets to perform interprocess communication (IPC), which has also been integrated for use with RF communication.

The second requirement states that this subsystem must perform necessary file table and boot request entry updates as a final step to apply an update. This functionality has been successfully implemented in the process, and the updated file table and boot request entries are inserted into the system state logs to apply a system update. Much of the information necessary to update these entries is included with the update request command parameters, such as desired version type, and associated MD5. However, preexisting boot request and file table entries are taken into account since their preexisting state must also be reflected in new entries (i.e., preexisting entry information may need to be merged with new entries).

A couple primary testing techniques have been used to validate this subsystem. To test remote communication capability of this process, commands to apply software updates were sent using a preexisting command client. This

client was developed for other testing on the avionics and has sufficed for similar communication testing. Overall, simple commands were first tested for receive functionality and for commanding software updates. Commands to update either a Linux image or secondary root file system were successfully received, and any error codes that were generated by the process were successfully retransmitted.

To test software update application functionality, the process was commanded to perform all three types of potential software updates independently. One of each type of update was transferred to temporary NAND storage, and then copied to their respective destination devices as the updates were applied.

To ensure individual Linux images were applied properly, the expected kernel and root file system image versions were viewed upon system startup. The overall system log state is printed at startup, and this capability was leveraged for some of this testing. Initially, the first boot cycle used specific versions of these images for startup, and these were noted. After a system image was updated, the expected version number change was verified on subsequent system startup. Additionally, the overall state of the file table entries and boot requests were viewed for verification. An extremely similar setup was used to test secondary file system updates with single file updates.

Lastly, manual resets of the avionics platform were conducted during different phases of the update process to simulate unexpected system resets. This was done to ensure the system would not create additional upsets upon subsequent startup if the software update did not succeed. Resets were performed intermittently during the final validation, file table, and boot request entry update phases. Upon system reset, no unexpected error occurrences were observable and the system was still able to boot using the original secondary file system partition or Linux images. The software update process could still run successfully to apply an update even after experiencing an intermittent system reset.

An overview of the primary test cases and their results are presented in Table 4.3.

Table 4.3: Final Update Tests and Results

Primary Test	Result
Remote command transaction	Received and executed
Error code generation	Generated by invalid command or other error
File table entries updated	Validated on subsequent reboot
Boot requests updated	Validated on subsequent reboot
Reset at different phases	No further upsets encountered

5

System Results

Each major subsystem has been individually tested and verified, and this chapter briefly discusses the overall system design success and compliance with original system requirements. The system was designed to provide software update functionality for critical components of the avionics system, and to conduct this process reliably by considering validation and recovery. This system has achieved the desired functional behavior and met system requirements.

5.1 System Success

After each subsystem was independently tested, they were integrated as a whole unit to perform system state logging, recovery from invalid software updates, validation, and the application of software updates. Each of these subsystems currently work in unison and have been tested together to achieve overall desired software update functionality.

This system has resulted in two main processes that execute at startup. One is responsible for validation and recovery of the secondary file system as well as writing boot success messages. The other is responsible purely for cleaning and synchronization of the system state logs. Additionally, a third process is executed on the avionics for final application of the software updates when a remote request is provided.

Each major goal of the system has been realized, and a summary of this is provided in Table 5.1.

Table 5.1: Desired Goals and Status

System Goal	Status
Support for major component software updates	Achieved
Validation and recovery of software updates	Achieved
Tolerance of system upsets	Achieved
Extensible solution for avionics	Achieved

This system does not only apply to a single mission, but it is tightly integrated with the current software and hardware architecture of the avionics. Since it is anticipated that these architectures will not vary much for future missions, this software update system can and should be used on future missions.

To some degree, tolerance of system upsets has been achieved. The overall system is designed to handle potential resets that may occur during the update process and recover if necessary. A concern during system design was handling memory corruption of any element in PCM, including the system state logs and Linux images. To handle these cases, these images are validated on startup and recovery is performed as necessary with backup images or backup secondary file system partitions. Additionally, if an upset occurs that corrupts data in either system state log, whether it's a radiation induced bit flip or a reset occurs while log entries are being written, the other log copy can be used to resynchronize the two. Overall, bit flip corruption is tolerated for any region of memory in the PCM except for the bootstrap region, or the log regions if both happen to become corrupt.

Unfortunately, these scenarios are unrecoverable. For example, if the system can not boot for some reason using any set of images within the PCM or because the bootstrap region is corrupt, this would result in a mission failure. In this case, a software update could not be applied to attempt to recover the system. Additionally, if both system state logs are corrupted such that they cannot be synchronized, this may result in undesirable behavior, or an inaccurate system state representation at startup.

Lastly, all system requirements have been met, and these are summarized in Table 5.2. Each requirement was met and verified in different ways. The right-

hand column presents summarized justifications for how each system requirement was met.

Table 5.2: System Requirements and Verification

System Requirement	Justification
Compatible with overall avionics	Solution developed for avionics
Available RF support hardware	Functional remote file transfer
High latency remote data transfer	Preexisting solutions for support
Temporary update storage	NAND flash utilization
Small memory footprint updates	Average 1-2 MB footprint
Minor changes between missions	Only a couple foreseen
Specific formats	Supports compressed images or individual files
System recovery	Verified with system testing
Robust and low overhead validation	MD5 checksum and system performance
Tolerance of unexpected upsets	Verified with system testing
Application of software updates	Verified with system testing

5.2 Additional Results

A couple additional performance tests were conducted to measure the overhead incurred from the new startup, validation, and recovery procedures. The most costly procedure is the validation and recovery of the secondary file system. After a series of time tests using the Linux *time* utility, an average wall-clock time of 20 seconds is required for validation and single file recovery. These tests were conducted with an empty microprocessor cache to emulate system startup.

Although this performance is not ideal, the system still recovers and validates properly, which is the most important result. Currently, no planned missions depend on a quicker validation of the secondary file system. An extension to this validation procedure has been considered, and it has been partially developed. This feature is priority-based file validation, which is performed by simply including a file that contains entry names of files that are validated in a priority fashion

before any others. If any files need quicker validation for a future mission, this feature can be fully developed.

Secondly, an average time to clean the system state logs was obtained to estimate the overhead of a worst-case log clean procedure. Depending on the number of entries to delete, the cleaning process may require several more or less seconds than the results provided. The test was conducted with cases of deleting 5 and 15 entries. The average wall-clock time to complete the cleaning of these entry sets was approximately 2 to 5 seconds. Although these results are useful to note, the cleaning process may only ever need to clean 1 to 2 entries at a time because they are cleaned at every system startup. Thus, the performance overhead from cleaning and synchronizing the system state logs may only ever require milliseconds of time.

6

Related Works

During the design and development phase of this thesis, a variety of works were evaluated for guidance. A series of papers that influenced major design decisions are summarized and detailed descriptions of their contributions to this thesis are presented. A summary of each design decision is presented below:

1. To store software updates and images reliably, a few decisions were made regarding the different memory components and in what fashion they're stored. Eventually, the decision to place critical data on a non-volatile phase-change memory device was made to limit the potential radiation effects that may occur and upset the system. Discussion of the research investigating the benefits of using phase-change memory in a space environment, specifically its inherent radiation tolerance, are presented.

2. NAND memory technology has a variety of attributes that must be considered to use it effectively and retain its lifetime. These attributes exist due to its manufacturing process and internal structure. More detail about this internal structure and a preexisting solution used on the avionics to help utilize this technology efficiently are discussed.

3. The system state log maintained provides capabilities necessary for software updates, such as recovery in case of update failure, and software update application requests. It provides common features found in many other log-based implementations for other applications, such as checkpointing and

rollback techniques for error recovery. The system state log used for software updates was designed with the guidance of these works.

6.1 Ionizing Radiation Tolerant Non-Volatile Memory

Since doses of ionizing radiation can cause undesirable memory upsets and potentially incorrect software behavior, mechanisms to assist in limiting or preventing drastic consequence from these circumstances should be employed. The platform hardware designer initially chose to utilize a structural phase-change non-volatile memory in an unknown fashion, but additional research showed that storing the most critical data (i.e., kernel and root file system images) on this device would potentially result in more desirable behavior on the avionics. For example, if the kernel and root file system become corrupted on-orbit, the system would fail to reboot or not boot at all and the planned mission could not continue. Storing this data on such a device should help to prevent such occurrences.

These papers [22, 23, 24] discuss the underlying technology behind phase-changing, or chalcogenide memory cells (PCM) and their effectiveness to tolerate the space environment.

6.1.1 Phase Change Memory Overview

PCM technology utilizes chalcogenide based alloys within memory cells by representing bit states with two different structural phases of a specific alloy, rather than with traditional electrical charge. These two unique structural forms, known as amorphous and polycrystalline states, exhibit different but identifiable optical and resistive properties, thus allowing for representation of standard electrical bit states. Adjusting the alloy physically requires thermal activation, which is achieved by applying different levels of electrical potential to heat the material at varying periods of time. Ultimately, these cells require no extra energy to maintain, but only to set various structural phases. This creates an inherently non-volatile and radiation tolerant memory device [24].

As rare as it seems, this technology has actually been commercially used in CD and DVD disk production for several years. When writing data to a disk, a high-power laser thermally forms differently phased memory cells whose states are then read with a low-power laser by pointing and measuring different reflections produced from each cell. Fortunately, continued developments have improved this technology and made it a suitable device for this avionics platform.

6.1.2 Attractive Features

Although this memory behaves differently from standard non-volatile semiconductor memory (e.g., EEPROM, flash), there are a number of features in addition to invulnerability to ionizing radiation that make it an attractive standard memory alternative for space applications. This technology also has reduced susceptibility to harsh thermal environments while maintaining low manufacturing costs, low power consumption, and achieving DRAM speeds [22]. It also utilizes a standard electrical bus interface and practical physical size for simple integration into this platform. These features form a strong argument for establishing this unique memory permanently onto this platform. Furthermore, experimentation to help measure its actual robustness to radiation exposure has been conducted and summarized results are provided.

6.1.3 Evaluating Upset Susceptibility

Despite existing models of expected radiation doses within certain spacecraft orbits, the actual absorbed radiation by electronic components will vary due to factors such as the current solar cycle, and exact proximity to the Earth's poles [25]. Additionally, depending on the microelectronic device, upsets can occur with exposure ranges between 1000 rad(Si) and 10 Mrad [26]. Thus, to accurately evaluate electronic component tolerance with radiation levels similar to those in space is difficult. However, a couple experiments conducted in these papers measure susceptibility to upset or latchup events with comparable radiation exposure [23, 24].

For both experiments, variations in successfully reading bits (i.e., reading back expected bit states after writing) occurred with increased irradiation of different

PCM devices. One experiment tested with a 64 Kbit memory array while the other with a 4 Mbit using exposure levels between 0 and 30 Mrad. However, the occurrence of incorrect reads due to these upsets was low at high irradiation levels and did not affect the actual memory cell contents since further subsequent reads were successful.

Based on these results, utilizing a PCM device on the avionics suits as an ideal application. Since the kernel and file system images, including potential backups, are read only during boot from the PCM, the number of expected read operations performed should be minimal. Utilizing the PCM in this fashion will help to prevent critical system memory corruption that may occur and recovery from potential read failures during boot, both of which are crucial for mission success.

6.2 Important Considerations for NAND Memory

Flash memory has become the standard for embedded system non-volatile memory use due to several attractive features, such as low-power consumption and low cost. Two distinct types exist, NAND and NOR, which in short, differ in their memory cell arrangement. Each device has a cell arrangement that corresponds similarly (at the transistor level) to how NOR or NAND logic gates are architected [11], hence their naming. This creates fundamental interfacing differences between both types and necessitates considerations of how each should be employed. These papers [27, 28] discuss the limitations of NAND technology that were considered prior to its use on the avionics, and offers existing solutions designed to extend device lifetime and improve reliability.

6.2.1 Issues with NAND Flash

Despite the number of advantages, some disadvantages exist that should be recognized. The serially structured memory cells within a NAND device facilitate high storage density, but this means data cannot be accessed randomly like in RAM devices. Thus, a serial method to read or write data in large subsets known as

pages must be used instead of other common addressing schemes that access data in a truly random (i.e., byte-wise) fashion. This introduces extra overhead when reading from or writing to a NAND device, and may reduce efficiency depending on the circumstance (e.g., only a subset of the data is needed from a single page read). This serial behavior does not significantly degrade performance for the avionics since the average number of read and write operations would be minimal for applications on current missions. This should still be noted since this may differ for future missions.

Another issue is anticipated page failure caused from imperfect manufacturing processes, and inherit limited lifetime of the memory. Typically, sections of the memory known as blocks have a limited number of write cycles before completely deteriorating, and some blocks are expected to be damaged directly after manufacturing. To avoid using these blocks, the manufacturer will test and "mark" any as reference so they are not used. To limit producing bad blocks and to detect and avoid newly formed bad blocks, software solutions have been developed and proposed in these papers. Since integration of such a solution allows for a subset of tolerable memory failures and improved robustness, these options were explored. Fortunately, these solutions happen to exist in the form of file systems, several of which Linux already supports.

6.2.2 YAFFS2 Utilization

Yet Another Flash File System (YAFFS) was originally developed as the first optimized NAND file system for Linux, and it is also used by the popular Android mobile software stack [27]. YAFFS2 was established soon after to support additional features for current NAND flash devices, and it eventually became an addition to the avionics Linux build.

The features provided from this file system make it an ideal candidate for improving reliability of a NAND device. The design of this file system attempts to prolong the total lifetime of the memory by utilizing dynamic wear leveling techniques. These techniques uniformly distribute several write operations among multiple, instead of single blocks [28]. Ultimately, this limits the number of write cycles performed on individual blocks such that they degrade slower over

time. YAFFS2 also employs bad block management methods that perform write verification and remapping of data to new blocks in the event of write operation or entire block failure [27]. Such features should provide extreme benefit to the system, especially in the form of a robust and well-tested file system.

6.3 Log-based Rollback Techniques For Error Recovery

Log-based and checkpoint-based rollback recovery protocols have been a prime subject of research for some time due to their core issues and the solutions developed for them. Log-based or checkpoint-based rollback recovery is used to rewind the system state to a previous known or preserved working state if a failure is encountered. Among other systems, distributed systems make use of rollback recovery techniques for error recovery similar to the log-based technique used in this thesis. The papers [29, 30, 31] were evaluated for guidance during its design.

6.3.1 Distributed Systems and Parallel Applications

A distributed system typically uses multiple nodes to act as a cluster to perform extreme large-scale computations. For several applications, this is a more efficient solution than one that could be developed serially [30]. However, unless these nodes can handle errors gracefully by perhaps recovering to a previous state, the distributed system will eventually degrade in its performance and overall service as nodes encounter errors. One of the most common fault tolerant techniques used for these parallel applications is checkpoint-based rollback recovery.

6.3.2 Checkpoint-based Rollback Recovery

Generally, the checkpoint-based rollback recovery protocol periodically store the snapshots of a system's application or multi-application state in order to be restored if ever necessary. One major problem to consider is known as the domino effect, where under some recovery scenarios, rollback may be propagated so deep

that all previous computational work completed by one or multiple nodes is lost [31].

There are three primary types of checkpoint-based rollback recovery protocols, two of which are not vulnerable to the domino effect. The first type of checkpoint-based protocol is known as an uncoordinated checkpointing scheme. This essentially means any node or application executing on a node decides when to save its local state into 'stable' memory. It's assumed that a reliable memory source exists on which this state can be stored, which is categorized as a 'stable' device. This type of protocol is actually less complex than others, but does suffer from the disadvantage of the domino effect.

Coordinated checkpointing is the second type of protocol where all nodes or applications must organize and synchronize their individual checkpoints such that a single consistent application checkpoint is generated and saved [30]. In this scenario, generating a checkpoint induces more system overhead and it's usually more complex. However, this significantly reduces the complexity of restoring a preserved state since only a single checkpoint needs to be evaluated rather than multiple. More importantly, this checkpointing scheme does not suffer from the domino effect.

The last type is quasi-asynchronous, or communication-induced checkpointing, which is a hybrid of the previous two. This scheme does not require global coordinated checkpointing and nodes can perform checkpoints leisurely, but some applications or nodes can be forced or required to perform a checkpoint when necessary. Forcing checkpoints results in the avoidance of the domino effect, but this protocol is the most complex of the three.

6.3.3 Log-based Rollback Recovery

Log-based rollback recovery is an extension of checkpoint-based, except that it involves message recording as an additional mechanism to save and restore state. This rollback recovery technique is extremely similar to that of the system state log solution designed in this thesis. One primary advantage is that this protocol will result in a more recent state recovery during rollback than checkpoint-based recovery. This behavior is usually desired because during recovery, losing the least

amount of application state from error is more beneficial. In a distributed system, all messages communicated among all nodes during checkpoint-based recovery are now saved with a log-based rollback recovery system. Thus, their most recent states prior to any error can easily be recalled.

This recovery technique depends on a piecewise deterministic model, which assumes the system can be modeled as a sequence of deterministic states that begin with the execution of a non-deterministic event [29]. A deterministic event could be storing system state log entries at different startup points on the avionics, while a non-deterministic event would be an unexpected system reset. Thus, the system has complete knowledge of non-deterministic events that would transition the system to a subsequent state if such an event occurred. For example, a non-deterministic reset on the avionics prior to saving boot status success messages during Linux startup will result in a retry to boot using the same Linux images. This assumes the number of boot attempts has not been exceeded, since this is required in order for a boot retry.

There are two primary types of log-based recovery protocols, one of which is used in the system state logs. The first is known as a pessimistic or synchronized logging technique, which records each event before the event actually takes place on the system. This simplifies both the recovery and cleaning process for the system state log. With this scheme, the system state does not depend on any non-deterministic events that may not be able to be reproduced during recovery. In other words, the state is only dependent on the deterministic events that are logged, such as boot attempts, successes, and requests. Additionally, this only requires that the most recent messages be contained in the log to reflect the current system state. Older messages, or those with lower sequence numbers, can simply be discarded for the cleaning process.

The last primary type is optimistic or asynchronous, which is more ambitious and partially stores its state in volatile memory, as well as non-volatile. However, this has the drawback of potentially losing critical state if the volatile memory is recycled due to a system upset. Recovery is still possible, but most likely to a less recent state since multiple rollbacks may be necessary. A pessimistic rollback recovery type has been integrated into the system state log due to its overall robustness, and simplified recovery and cleaning processes.

7

Conclusion

7.1 System Success

The design and implementation of the software update feature-set for this platform has been an exceptional learning experience and an overall success. Each major subsystem has been developed to form a software update architecture that achieves the desired system goals.

Software updates can now be remotely requested and performed on any major part of the avionics system, including the Linux kernel, primary root file system, and secondary file system. This software update feature-set has also been developed to support current and future missions, which is an invaluable characteristic. Meeting these two primary goals provides the flexibility to support software update functionality for various components on multiple missions, which may differ depending on mission requirements and desired functionality. Validation is also performed prior to any update being used, and recovery options are available if perhaps the update becomes inoperable. Validation and recovery of critical system elements, including Linux, is now always performed at startup to improve the overall fault tolerance from potential system upsets. Lastly, each major subsystem was designed to tolerate unexpected system upsets and does not induce further upsets if such an event does occur.

Although there are many components that form this complex system, it's an extremely valuable and desirable solution to have for the avionics. In future mission planning and mission considerations, the design process may drastically

change given the flexibility now available with this system. For example, if additional physical volume is available for a mission, a secondary payload could be designed. This payload would not have to be related to the primary payload and should not be mission critical, as the goal is not to deviate from the primary mission. However, if this secondary payload is simple enough to design while staying within the desired timeline and budget for a mission, it could be developed, tested, and flown. Development of the supporting software for this subsystem could be completed perhaps after developing the primary mission functionality. The key advantage is that this development could take place before or after the satellite makes it to orbit.

The ability to update system software will most likely be leveraged more extensively to remedy issues in software applications that are detected post-launch. Depending on the complexity of the mission, a variety of scenarios will not be tested on the ground simply because there are so many. Additionally, there are always errors encountered in space given the characteristics of the environment. Hopefully, any encountered errors that may result are recoverable and can be fixed by a software update. Recovery will most likely be necessary for several future missions, whether the encountered errors are minor or major.

7.2 Current Progress

This feature-set has not yet flown on any missions, but all upcoming missions should have this functionality enabled. All major subsystems, including system state logging, validation, and recovery are functional and have been tested on the avionics platform. A few convenient but minor additions are still missing that should be added.

7.2.1 Build Integration

All of these features have yet to be integrated into the overall PolySat software build system. As they were being developed, a build solution more suited for individual development was used. Although this does not make it infeasible to use on the avionics system, the necessary setup to currently integrate is almost entirely

manual. Most of the preexisting build process, including generating Linux images and bootstrap binaries is automated, and integration of the software update features should be done in a similar fashion.

More specifically, generation of the secondary file system directory listings is done manually, as well as placing initial file table entries for both image sets into the system state logs. Although the latter cannot easily be performed without system startup, a small program to write these entries only once could be added to Linux startup.

Secondly, only one partition of the secondary file system is flashed to NAND by an auto-generated script when being programmed. This approach is used for quicker development and testing on the avionics, but ultimately, a separate flight script should be generated that flashes all five partitions. It would be an unfortunate mistake to forget to flash all NAND partitions prior to conducting a mission.

7.2.2 Test Modes

Developers frequently use the avionics platform to test many other software applications or hardware subsystems. The validation and recovery process used for the secondary file system will always execute by default once upon avionics startup, and this may not be necessary depending on what a developer may be testing. Thus, a test mode should be defined that can enable and disable this feature in case it becomes an inconvenience while testing other features.

This may also be true for log-based actions for the system state, such as cleaning entries. However, this procedure incurs much less overhead than secondary file system validation, and it should not be an issue.

Although test modes are convenient, adding this functionality should be done carefully. The developer should be aware of them and not always have them enabled. Otherwise, testing this feature-set as an integrated unit with other test applications may present future issues. It is better to always perform such integrated testing early in case any issues may arise.

7.3 Future Work

A variety of features to use this functionality in flight, as well as to enhance its reliability can be added as an extension to this work. Three primary components that would be immediately beneficial are described.

7.3.1 Command Structure and Organization

In order to test this system, a simple command set was developed for the final validation and application phase for software updates. However, this command set is probably not the most efficient and it could be designed better.

For example, the data transferred for the command to apply a kernel image update included the full path name of the file. This is not the most efficient way to request that an image update be applied to the system over RF. Thus, a software update application command-set should definitely be redesigned for overhead efficiency.

Secondly, organization of the update files in NAND could be better defined in order to make the feature-set more user-friendly. For testing, a naming scheme only known by myself was used to simply delineate between kernel and root file system update versions, as well as their temporary location paths in NAND. This should be predefined for all future missions and known by other developers than just myself.

7.3.2 Kernel Log Entry

Currently, the boot status success entries for a specific kernel and root file system image set are logged simultaneously at startup. These entries are written using a user-space process, which means that it depends on the root file system mounting. However, this forces a dependency between the two entries in that both images must be valid in order for their individual success entry to be written. The system state log and boot process are actually flexible enough to attempt booting with perhaps a different root file system, or different kernel image if only one of them happened to be invalid. With this dependency, the number of potentially valid boot combinations for a set of images is reduced.

To enable entry writing within the kernel, this must be done in kernel space while also creating access to PCM. This kernel space functionality could easily use the preexisting SPI device driver, but specific components regarding system state log parsing and similar must be ported to kernel space.

7.3.3 Memory Write Protection

This is probably the most important component necessary to add to the system. As previously mentioned, without a valid bootstrap, the system will not boot and it cannot recover. Although nothing should be accessing the PCM device aside from the couple user-space processes at startup, one may unintentionally attempt to write to the wrong address, and this may be fatal.

This is a simple extra precautionary measure that will probably prove extremely valuable sometime in the future. The easiest solution is to add the series of region write-protection commands to the PCM from the host memory flashing application. The PCM is flashed using an executable on a host machine that is responsible for placing all Linux dependencies and bootstrap onto on PCM. This host application already has an existing interface to the PCM and thus, it would most likely be a minor addition in order to support memory protection behavior.

References

[1] Maral, G. and Bousquet, M. and Sun, Z. *Satellite Communications Systems: Systems, Techniques and Technology.* John Wiley & Sons, fifth edition, 2009. 1

[2] Puig-Suari, J. and Turner, C. and Twiggs, R.J. **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

[3] 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

[4] Manyak, G. and Bellardo, J. M. **PolySat's Next Generation Avionics Design**. In *Proceedings of the 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, SMC-IT '11, pages 69–76, Washington, DC, USA, 2011. IEEE Computer Society. 4

[5] Mehlitz, P. and Penix, J. **Expecting the Unexpected- Radiation Hardened Software**. 5

[6] Atmel. *AT91SAM9G20 Preliminary.* Atmel Corporation, rev. 6384d edition, May 2009. 10, 11, 14

[7] Atmel. *Using Atmel's Dataflash.* Atmel Corporation, rev. 0842d edition, 2002. 11

[8] Various contributing authors for online support group. **Linux & Open Source related information for AT91 Smart ARM Microcontrollers, http://www.at91.com/linux4sam/bin/view/Linux4SAM/GettingStarted**, 2012. 11, 14

[9] Woodhouse, D. **JFFS : The Journalling Flash File System**. Technical report, Red Hat, Inc. 12

[10] Boyer, F., Support and Training Group Engineer. **AT91SAM Boot Strategies Application Deployment**. Technical report. 13, 14

[11] Eureka Technology Inc. **NAND Flash FAQ**. Technical report, Eureka Technology Inc., Memory and other Microelectronics Design. 16, 78

[12] Brown, S. and Rose, J. **FPGA and CPLD architectures: a tutorial**. *Design Test of Computers, IEEE*, **13**(2):42 –57, summer 1996. 25

[13] Finnie, S., et al. **Hard Links and Symbolic Links, http://www.linuxclues.com/articles/17.htm**, 2008. 29

[14] Various contributing authors and developers for open-source embedded Linux tool. **Buildroot : Making Embedded Linux Easy, http://buildroot.uclibc.org/**, 2012. 32

[15] Sanders, V. and et al. **Booting ARM Linux, http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html**, 2004. 33

[16] Various contributing authors and developers for the FreeBSD project. **FreeBSD General Commands Manual - FTP, Internet file transfer program, http://www.freebsd.org/cgi/man.cgi?query=ftp**, 2008. 38

[17] Postel, J. and Reynolds, J. **RFC 959, File Transfer Protocol, http://www.ietf.org/rfc/rfc959.txt**, 1985. 38

[18] Hethmon, P. **RFC 3659, Extensions to File Transfer Protocol, http://www.ietf.org/rfc/rfc3659.txt**, 2007. 38

[19] Various contributing authors. **TCP, Transmission Control Protocol, http://www.networksorcery.com/enp/protocol/tcp.htm**, 2012. 38

[20] Rinne, T. and Tatu, Y. and et al. **FreeBSD General Commands Manual - SCP, Secure copy program, http://www.freebsd.org/cgi/man.cgi?query=scp**, 2010. 38

[21] Various contributors to the GENSO project. **Global Educational Network for Satellite Operations, GENSO, http://www.genso.org**, 2012. 39

[22] Tyson, S. and Wicker, G. and Lowrey, T. and Hudgens, S. and Hunt, K. **Nonvolatile, high density, high performance phase-change memory**. In *Aerospace Conference Proceedings, 2000 IEEE*, **5**, pages 385 –390 vol.5, 2000. 76, 77

[23] Gasperin, A. and Wrachien, N. and Cester, A. and Paccagnella, A. and Ottogalli, F. and Corda, U. and Fuochi, P. and Lavalle, M. **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, sept. 2007. 76, 77

[24] Maimon, J. and Hunt, K. and Rodgers, J. and Burcin, L. and Knowles, K. **Radiation Hardened Phase Change Chalcogenide Memory: Progress and Plans**. 76, 77

[25] National Aeronautics and Lyndon B. Johnson Space Center Space Administration. **Understanding Space Radiation**. Technical report, National Aeronautics and Space Administration, Lyndon B. Johnson Space Center, October 2002. 77

[26] Stassinopoulos, E.G. and Raymond, J.P. **The Space Radiation Environment for Electronics**. *Proceedings of the IEEE*, **76**(11):1423 – 1442, nov 1988. 77

[27] Maker, F. and Yu-Hsuan, C. **A Survey on Android vs. Linux**. Technical report, Departments of Electrical and Computer Engineering and Computer Science, UC Davis. 78, 79, 80

[28] Tei-Wei, K. and Yuan-Hao, C. and Po-Chun, H. and Che-Wei, C. **Special Issues in Flash**. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 821 –826, nov. 2008. 78, 79

[29] Wang, Y.M. and Huang, Y. and Fuchs, W.K. **Progressive retry for software error recovery in distributed systems**. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 138 –144, june 1993. 80, 82

[30] Treaster, M. **A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems**. *CoRR*, **abs/cs/0501002**, 2005. 80, 81

[31] Elnozahy, E. N. (Mootaz) and Alvisi, L. and Wang, Y. and Johnson, D. **A survey of rollback-recovery protocols in message-passing systems**. *ACM Comput. Surv.*, **34**(3):375–408, September 2002. 80, 81