ROM Boot Module Specification

James E. Francis

MIT 64-30110.2201

Rev. E

July 2, 1993

Contents

Figures

1 Introduction

1.1 Scope

This document describes the components that make up the ROM Boot Software Module (BOOT). This module consists of a procedure and a set of functions that the Experiment Data System (EDS) uses to load the Flight Software.

This document does not describe the overall design of the EDS, nor does it describe the operating details of each component in this module. It assumes that the reader is extremely familiar with the Flight Software Global Design, with the "C" programming language and with the architecture of the EDS.

This document should be of primary interest to Flight Software Engineering, Flight Hardware Engineering, Ground Software Engineering and Software Quality Assurance.

1.2 References

ISBN 1-55512-055-5 1987	"80286 and 80287 Programmer's Reference Manual"
HARRIS Microprocessor Products, 1992	"82C237 CMOS High Performance Programmable DMA Controller".
MIT 64-02006 Rev. D	"XTE EDS Hardware Functional Description and Requirements"
MIT 64-30102 Rev. E.......................	"Flight Software Global Design"
MIT 64-03007 Rev. C	"SDS to EDS Interface Control Document"

1.3 Module Perspective

The following diagram is a copy of the Level 1 data flow diagram from the **Flight Software Global Design**, Section 4.1.2. This module's scope is reflected by the shaded areas:

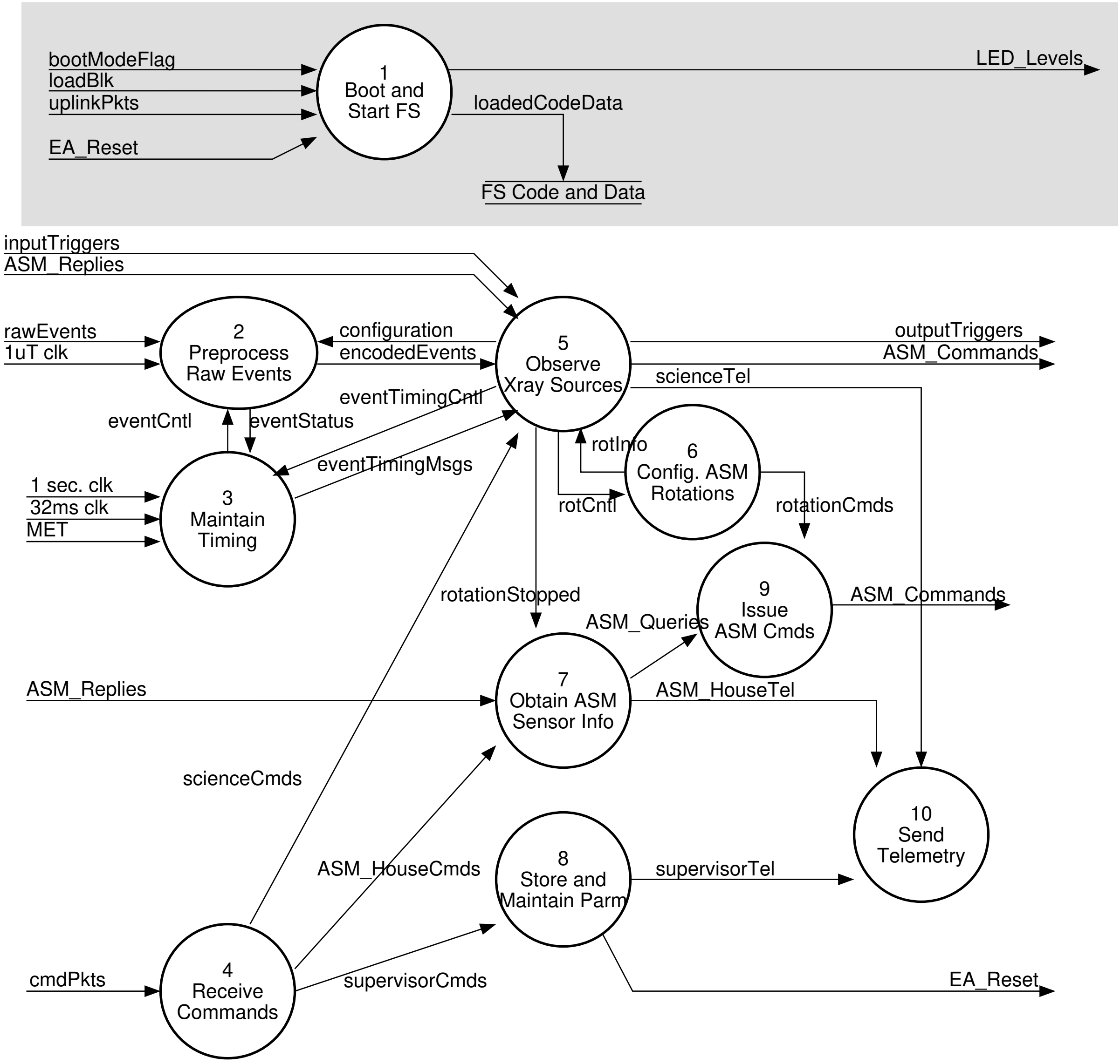

Figure 1 Module Perspective

1.4 Document Overview

Sections 1 and 2 provide an overview of the ROM Boot Module, summarizing the purpose, services, context, interfaces and basic operation of the module. Sections 3 - 5 provide the details of the hardware and command interfaces to the module, the system environment when the boot module is invoked, and the system environment when the boot module passes control to the loaded code. Section 6 describes some of the implementation details of the module. Section 7 describes which requirements are met by the module. Section 8 describes what types of tests need to be performed to ensure that the implementation of the module is operating properly. Appendix A describes the probability of Single Event Upsets while the ROM Boot Module is running.

1.5 Acronyms

Please refer to the **Flight Software Global Design Specification**, Section 1.5, for a list of acronyms and abbreviations. In addition to these acronyms, this module defines and uses the following:

LED .... Light Emitting Diode. On the ground breadboard hardware, the LED signals are wired to physical LED's. On flight hardware, however, the levels of these signals are included in the discrete telemetery sent to the SDS by the hardware, and subsequently to the ground.

2 Overview

2.1 Purpose

The purpose of the ROM Boot Module is to load and execute code from either the ROM Repository or from the DMA Controller's uplink channel. The ROM Boot Module is stand-alone, in that it does not require the services of any other module in the Flight Software.

2.2 Services Provided

The ROM Boot Module provides the following services:

1) ROM Repository Boot .... Load code and data from the ROM Repository, and start execution of the loaded code.
2) Uplink Boot .... Load code and data from the command interface, and start execution of the loaded code.

2.3 Module Context

The following illustrates the ROM Boot Module with respect to the EDS system:

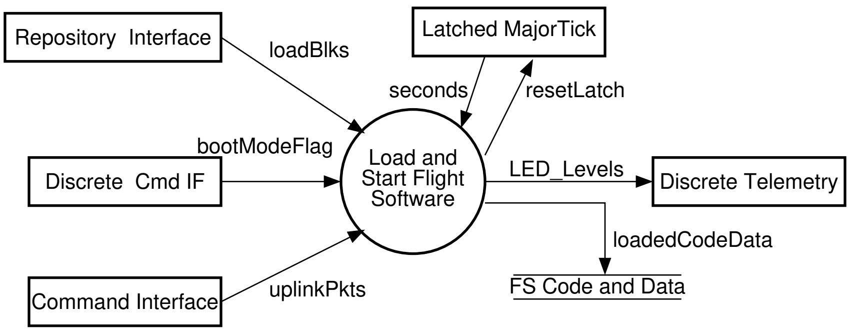

Figure 2 ROM Boot Module Context Diagram

The ROM Boot Module directly manipulates the hardware, and does not rely on any other software modules. It uses the Discrete Command Interface hardware to select which type of boot to perform. If a repository boot is requested, it loads and executes code from the ROM/Patch Repository, using repository load blocks (loadBlks). If a given access to the repository requires re-synchronization, the ROM Boot Module uses the latched MajorTick pulse signal to measure the passage of time (seconds, resetLatch). If an uplink boot is requested, it reads, stores and executes one or more uplink packets (uplinkPkts) using the DMA's uplink channel. During the boot, this module indicates its progress by setting LED values, which are read by the SDS once every 8 seconds, and are telemetered to the ground in the SDS's discrete telemetry blocks.

2.4 Theory of Operation

The ROM Boot procedure resides in each EA's boot ROM, and is only entered after a CPU reset. It is responsible for loading and executing code from either the ROM repository or from the uplink channel. The loaded and executed code is then responsible for all further system startup requirements. The uplinked code packet or ROM code load block must conform to a predetermined format. To keep this module as simple as possible, the ROM Boot procedure does not produce any downlink packet telemetry, but indicates its progress via the LED levels. It sets the LED's to values which indicate the operation that is about to be attempted. If and when the boot hangs or waits, these values tell the ground which operation is stuck.

Upon reset, the ROM Boot module tests the state of the uplink boot flag, which can be set and cleared by the ground using discrete hardware commands (the hardware clears this flag when the System Manager is powered up). If the flag is clear, it reads from repository until it obtains a synchronization word, after which it copies a single load block from the repository and executes it. Each synchronization attempt is separated by at least 2 seconds to ensure that a successful access occurs as close to the start of the arbitration window as possible. If the uplink boot flag is set, it loads one or more uplink packets using the DMA's uplink channel, stores the code contained in the packets, and executes the loaded code.

Please read the **XTE-EDS Hardware Functional Description and Requirements**, "BREGS and 286 Peripherals - Repository" section for a description of the Repository Hardware. The ROM Boot Module uses synchronization words and delays between retries to accomodate the restrictions imposed by the hardware.

2.5 Interface Summary

The ROM Boot code cannot rely on the presence of the main Flight Software. Therefore, it must access the hardware directly. The ROM Boot Module uses the following hardware components:

80286 CPU .... Initializes certain registers and disables CPU interrupts.

82C237 DMA Controller	Used to transfer uplink boot packets from the SDS Interface.
CPU Status Register	Contains Uplink Boot Flag and Uplink Interrupt Latch Contains MajorTick latch
CPU Control Register	Used to set LED levels
CPU Pulse Register	Resets Uplink Interrupt Latch Resets MajorTick and MinorTick latches
ROM Repository Register	This is where ROM/Patch load blocks are read from.

2.6 Data Structures and Constants Summary

The ROM Boot Module defines the format of the startup code load blocks and uplink packets.

It also defines the LED values set during boot. These values indicate what the system is doing during the boot.

2.7 Interrupt Summary

This module disables interrupts, via the CPU, and leaves them disabled. Since this module does not program the 82C59 Interrupt Controller, it is up to the loaded code to program the controller, install vectors, and enable interrupts, if and when it needs interrupts.

(NOTE: Although the CPU interrupt is disabled by hardware upon reset, this module will also actively disable it. This is in case the module was entered by some means other than a hardware reset.)

2.8 Function List Summary

This module does not provide any functions for general use. This module is entered automatically, and only after a CPU reset.

3 Data Structures and Constants

3.1 Hardware

This module accesses all hardware registers directly. Please refer to the **XTE EDS Hardware Functional Description and Requirements** for a detailed description of the EA specific hardware, to the **82C237 CMOS High Performance Programmable DMA Controller data sheet** for a description of the DMA controller hardware, and to the **80286 and 80287 Programmer's Reference Manual** for a description of the 80286 CPU.

3.1.1 CPU Status Register (16-bits, 0x0a0000 flat address- read only)

The Boot Module uses the *Boot_Uplk* bit (bit 8) in the CPU Status Register to determine whether to boot from the ROM Repository, or to boot from the DMA's uplink channel. If the bit is clear, the module boots from the ROM Repository. If the bit is set, then it boots from the DMA's uplink channel. The ground is responsible for setting or clearing this bit, using discrete hardware commands, depending on the desired type of boot. If this bit is set, the ground is responsible for sending one or more uplink boot commands (see Sections 3.2.1, 3.2.2 and 3.2.3).

This module uses the *UpInt* bit (bit 3) in the CPU Status Register to determine when a complete uplink packet has been transferred during an Uplink Boot. This bit is reset using the CPU Pulse Register (see Section 3.1.3).

This module uses the *Maj_Lat* bit (bit 2) in the CPU Status Register to determine when a MajorTick pulse has occurred when re-synchronizing to the start of the ROM Repository. This bit is reset using the CPU Pulse Register (see Section 3.1.3).

3.1.2 CPU Control Register (16-bits, 0x090000 flat address- read/write)

This module sets and clears *LED* bits 0-2 (register bits 3-5) of the CPU Control Register to indicate the current stage of the boot. This provides the ground with some simple diagnostic information in case the system hangs or stops at a given stage during boot.

3.1.3 CPU Pulse Register (16-bits, 0x0a0000 flat address - write only)

The Boot Module uses the *Rst_UpInt* bit (bit 3) of the CPU Pulse Register to reset the Uplink Interrupt latch, contained in the CPU Status Register (see Section 3.1.1).

The Boot Module uses the *Rst_MajMin* bit (bit 0) of the CPU Pulse Register to reset the MajorTick latch, contained in the CPU Status Register (see Section 3.1.1). This bit also has the side-effect of resetting the MinorTick interrupt latch (*Min_Lat*, bit 0) in the CPU Status Register. The Boot Module does not use this Status Register bit or interrupt.

3.1.4 Repository (16-bits, 0x0b0000 flat address, read only)

The ROM Boot Module accesses the ROM/PRAM Repository via a 16-bit register. The address of this register is aliased across the entire 64Kbyte segment (i.e. addresses 0x0b0000 through 0x0bfffe address the same physical register). This register supplies each word from the System Manger ROM Repository and PRAM. The words are supplied in sequence with each read from the register. The ROM Boot Module repeatedly reads from this register when performing a boot from Repository. The entire load must take under 2 seconds and each load attempt must be separated by at least 2 seconds. Please read the **XTE-EDS Hardware Functional Description and Requirements**, "BREGS and 286 Peripherals: Repository" section for a complete description of this register.

NOTE: This module waits for at least 4 MajorTick pulses (see Section 3.3.17) to ensure that at least 3 seconds elapse before retrying loads from the repository. This handles any possible frequency or phase differences between the System Manager's timeout clock and the SDS supplied MajorTick clock.

3.1.5 DMA Controller (16-bit addr, 8-bit registers, 0x0c0000 - read/write)

When performing an uplink boot, this module programs the DMA controller directly. This module uses DMA channel 3 for uplink packet transfers. The EA hardware maps all of the DMA registers to word addresses, and therefore, the DMA register addresses always increment by 2. The following lists all of the register addresses (in flat address format) used by the ROM Boot Module:

Base Address Register, Chan. 3 .... (0x0c000c r/w) .... Destination Word Address
.... (byte address divided by 2)
Base Word Count, Chan. 3 .... (0x0c000e r/w) .... Word transfer length
.... (byte transfer length divided by 2)
Status Register .... (0x0c0010 r) .... Read Status Register
Command Register .... (0x0c0010 w) .... Write Command Register
Request Register .... (0x0c0012 w) .... Write Request Register
Single Bit Mask Register .... (0x0c0014 w) .... Write Single Mask Register Bit
Mode Register................................... (0x0c0016 w) .... Write Mode Register

Byte Pointer Flip-Flop Control .... (0x0c0018 w) .... Clear Byte Pointer Flip-Flop
Temporary Register .... (0x0c001a r) .... Read Temporary Register
Reset DMA Controller...................... (0x0c001a w) .... Master Clear
Clear all channel masks .... (0x0c001c w) .... Clear Mask Register
All mask register bits .... (0x0c001e w) .... Write All Mask Register Bits

Please refer to the **82C237 CMOS High Performance Programmable DMA Controller data sheet data sheet** for details on the register contents, and on how to use the DMA Controller.

3.2 Data Structures

3.2.1 Uplink Packet Format (uplinkPkt)

When the ROM Boot module boots from uplink, it receives the code to load in the form of CCSDS command packets. The ground is responsible for forming and ensuring in-sequence delivery of each packet. Each packet consists of a CCSDS header, a secondary header, followed by uplink boot load blocks. All packets contain an even number of bytes, which is less than or equal to 64 bytes. The first packet received consists of an Initial Startup Code Load Block. Subsequent packets consist of Startup Code Load Blocks. The formats of the Initial Startup Code Load Block, and Startup Code Load Blocks are described in Sections 3.2.2 and 3.2.3. All 16-bit words are delivered to the EA in little-endian format (least significant byte stored first). The following describes the format of the packet headers preceeding the load blocks:

Format

```
[v,v,v,t,h,a,a,a,a,a,a,a,a,a,a,a] ............ CCSDS Header Word 0
                                               v ................................ Version = 000
                                               t ................................ Type = Telecommand = 1
                                               h ................................ Secondary header present = 1
                                               a ................................ Application id
[g,g,s,s,s,s,s,s,s,s,s,s,s,s,s,s] .............. CCSDS Header Word 1
                                               g ................................ Segment Flag = None = 11
                                               s ................................ Sequence Number
[z,z,z,z,z,z,z,z,z,z,l,l,l,l,l,l] ............... CCSDS Header Word 2
                                               z ................................ Top 10 bits of Packet Length = 0
                                               l ................................ Lower 6 bits of Packet Length
[r,f,f,f,f,f,f,f,c,c,c,c,c,c,c,c] ............... Secondary Header Word 0
                                               r ................................ Reserved
                                               f ................................ Function Code
                                               c ................................ Checksum
```

Description

The ROM Boot Module ignores all fields in the CCSDS header except the packet length. This length is the size, in bytes, of the entire packet, minus the size of the primary CCSDS header (6), minus 1. The ground must ensure that the length of the application data (Startup Code Load Block) is *(packet length* - 1), and is always an even number of bytes. Since the maximum size of a command packet is 64 bytes, the top 10 bits of the Packet Length Field are always zero.

3.2.2 Initial Startup Code Load Block (initialStartupCodeBlk)

Both ROM Repository Boot and Uplink Boot load an initialStartupCodeBlk to determine how much code and data to load, and where to load it.

When performing an Uplink Boot, the ground must ensure that an initialStartupCodeBlk is contained in the application data area of the first uplink packet. The code and data for a given load may span more than one uplink packet. The application data of the first packet will contain the initialStartupCodeBlk (length, load address) for the entire load. The ground must then ensure that subsequent packets contain only the code (no address or length). The ground must ensure that such a series of packets must be transmitted in ascending order. Once "total length" bytes of code have been read, the boot procedure jumps to the original starting load address.

When performing a ROM Repository boot, the entire load is contained in the initialStartupCodeBlk, and is contiguous. Within the ROM, the initialStartupCodeBlk immediately follows the ROM Repository synchronization word (see Section 3.3.18).

Format

[total length] .... Total Load Length (4-bytes, little-endian)
[load address] .... Load/Execution Address (4-bytes, little-endian)
[code] .... Code to load.

Description

The startup code load block's purpose is to allow the ROM Boot procedure to load code and execute it. The fields have the following interpretations:

total length .... This is the total number of bytes to load.
During uplink boot, the system will read and load packets, into contiguous addresses, until this number of bytes have been loaded. During uplink boot, the ground must ensure that this length is even.
During ROM Boot, this is number of bytes to read from the ROM Repository.

load address .... This is the address at which to load and execute the code.
For uplink boot, the ground must ensure that this address is word aligned and that the total load from load address to load address + total length lies between BOOT_UP_PKT_ADDR + BOOT_UP_PKTLEN and BOOT_STACK_ADDR (see Sections 3.3.24, 3.3.25 and 3.3.8).

code .... The code to load. When performing an uplink boot, the length of this portion of the code is determined using length derived from the CCSDS header (see Section 3.2.1), minus the size of *total_length* and *load_address* (8 bytes).

3.2.3 Startup Code Load Blocks (startupCodeBlk)

Format

[code] .... Code to load.

Description

During uplink boot, packets are read and stored until *total_length* (see 3.2.2) bytes of code have been loaded. The field is interpreted as follows:

[code] .... Code to be loaded next to the previous packet's loaded code. The length of this portion of the code is determined from the CCSDS header (see Section 3.2.1).

3.2.4 ROM Repository Layout

The ROM Repository must start with the synchronization word (see Section 3.3.18) followed by a single initialStartupCodeBlk (see Section 3.2.2). The synchronization word allows the ROM Boot Module to find the start of the ROM Repository, and the initialStartupCodeBlk tells the Boot Module how much code and data to load and where to load it.

3.3 Constants

Some of the following addresses may not appear directly in the source code. Instead, their values may be used as inputs to the linking-locator used to build the final ROM Boot Image.

3.3.1 BOOT_RESET_VECTOR_ADDR (0x0ffff0, flat-address)

This is the address of the 80286 Reset Vector. Whenever the CPU resets, control is passed to this memory location. This address is in the Boot ROM address space.

3.3.2 BOOT_ROM_ADDR (0x0f8000, flat-address)

This is the lowest address populated by the Boot ROM accessable by the Flight Software. Addresses 0x0f000 - 0x0f7fff appear identical to addresses 0x0f8000 - 0x0fffff (see **XTE EDS Hardware Functional Description and Requirements**, Diagnostic Hardware Section).

3.3.3 BOOT_ROM_SIZE (0x08000 bytes)

This is the maximum size of the Boot ROM, as seen by the Flight Software.

3.3.4 BOOT_RAM_ADDR (0x000000, flat address)

This is the starting address of RAM.

3.3.5 BOOT_RAM_SIZE (0x040000 bytes)

This is the maximum size of RAM.

3.3.6 BOOT_INT_VECTOR_ADDR (0x000000, flat address)

This is the starting address of the interrupt vector table. (NOTE: this is the same address as the start of RAM (BOOT_RAM_ADDR). The ROM Boot Module fills the interrupt vector table with pointers to the reset vector early during its initialization procedures.

3.3.7 BOOT_INT_VECTOR_SIZE (0x0400 bytes)

This is the size of the interrupt vector table.

3.3.8 BOOT_STACK_ADDR (0x03ff00, flat address)

This is the base address of the stack space used by the ROM Boot Module.

3.3.9 BOOT_STACK_SIZE (0x0100 bytes)

This is maximum number of bytes that the ROM Boot Module uses for its stack. Since, on the 80286, the stack grows downward, the initial stack pointer is set to BOOT_STACK_ADDR + BOOT_STACK_SIZE (stack pointer is always word aligned). When "pushing" objects on the stack, the 80286 always pre-decrements the stack pointer. When "poping" objects off the stack, the pointers is always post-incremented. Since nothing is initially on the stack, the stack pointer points to the word immediately following the end of the stack.

3.3.10 BOOT_CNTL_LED_MASK (0x07)

This mask is used to select the lower three bits of a word to be written as the LED values to the CPU Control Register (see Section 3.1.2). This is used in conjunction with the left shift value described in Section 3.3.11.

3.3.11 BOOT_CNTL_LED_SHFT (3)

A masked LED value (see Section 3.3.10) is shifted left by this number of bits before being written into the CPU Control Register (see Section 3.1.2).

3.3.12 BOOT_STAT_MAJ (0x4)

This mask selects the MajorTick latch bit within the CPU Status Register (see Section 3.1.1).

3.3.13 BOOT_STAT_UPINT (0x8)

This mask selects the Uplink Interrupt bit within the CPU Status Register (see Section 3.1.1).

3.3.14 BOOT_STAT_BOOT_UPLK (0x100)

This mask selects the Boot from Uplink bit within the CPU Status Register (see Section 3.1.1).

3.3.15 BOOT_PULS_RST_MAJMIN (0x1)

Writing this bit to the CPU Pulse Register resets the MajorTick and MinorTick latches in the CPU Status Register (see Sections 3.1.3 and 3.1.1).

3.3.16 BOOT_PULS_RST_UPINT (0x8)

Writing this bit to the CPU Pulse Register resets the Uplink Interrupt latch in the CPU Status Register (see Sections 3.1.3 and 3.1.1).

3.3.17 BOOT_SYNC_DELAY (4)

This is the number of times the ROM Boot Module must wait for a MajorTick pulse, after failing to read the synchronization word, to ensure that the next attempt starts at the beginning of a grant access window (a value of 3 guarantees 2 to 3 seconds of delay. A value 4 handles any possible phase delay between the internal repository clock and the MajorTick clock).

3.3.18 BOOT_SYNC_WORD (0x04f525f4d, little-endian)

This 32-bit word is stored at the start of the ROM Repository Image, and is used by the ROM Boot Module to locate the start of the initialStartupLoadBlk (see Section 3.2.2). Since the repository is accessed using two reads, this constant must be broken down into a lower 16-bit value (0x05f4d) and upper 16-bit value (0x04f52), whose symbolic names are "BOOT_SYNC_WORDL" and "BOOT_SYNC_WORDH", respectively. NOTE: This value is the reverse of the ASCII character sequence "ROM_" and shall not occur anywhere except at the start of the loaded ROM Repository Image.

3.3.19 BOOT_MAX_WORD_CNT (0x7ff8)

This is the maximum number of 16-bit words that can be safely copied without crossing a segment boundary, assuming the "offset" portions of the addresses have been minimized (i.e. are less than 16).

3.3.20 BOOT_DMA_CMD (0x008)

This value is written to the DMA’s Command Register when performing uplink boot. This command value selects the following:

Bit 0 .... 0 .... Memory-to-Memory Disable
Bit 1 .... 0 .... Bit 0 == 0, ignored by DMA
Bit 2 .... 0 .... Controller Enable
Bit 3 .... 1 .... Compressed Timing
Bit 4 .... 0 .... Fixed Priority
Bit 5 .... 0 .... Bit 3 == 1, ignored by DMA
Bit 6 .... 0 .... DREQ Sense Active High
Bit 7 .... 0 .... DACK Sense Active Low

3.3.21 BOOT_DMA_MODE (0x047)

This value is written to the DMA's Mode Register when performing uplink boot. This command value selects the following:

Bits [1,0] .... 11 .... Select Channel 3
Bits [3,2] .... 01 .... Write to Memory Transfer
Bit 4 .... 0 .... Auto-initialize Disable
Bit 5 .... 0 .... Address Increment Select
Bit [7,6].. 01 .... Single Mode Select

3.3.22 BOOT_DMA_SNGL_ENABLE (0x003)

This value is written to the DMA's Single Mask Register to enable the Uplink Channel when performing uplink boot.

3.3.23 BOOT_DMA_SNGL_DISABLE (0x007)

This value is written to the DMA's Single Mask Register to disable the Uplink Channel when performing uplink boot.

3.3.24 BOOT_UP_PKT_ADDR (0x000400, flat address)

When performing an uplink boot, the ROM Boot module uses the DMA to transfer all uplink boot packets to this address. Once a packet is transferred, the code in the packet is copied to the appropriate address (the ROM Boot Module assumes that each packet is separated by at least 31.25ms).

3.3.25 BOOT_UP_PKTLEN (64 bytes)

This defines the maximum size of a command packet, in bytes. Other packet component sizes are derived from this value and the size of the various structure definitions (see Section 3.2). Some intermediate values may be used to form packet array sizes, but are not listed in this document.

3.3.26 BOOT_UP_ADJUST_LEN (7 bytes)

This is the value to add to the CCSDS Header Length field to get the total byte length of an uplink packet.

3.3.27 BOOT_MODE_REP

This symbol is used internally by the ROM Boot Module to indicate that the system should boot from the ROM Repository.

3.3.28 BOOT_MODE_UPLINK

This symbol is used internally by the ROM Boot Module to indicate that the system should boot from the Uplink Channel.

3.3.29 BOOT_LED_RESET (0x06)

The ROM Boot Module sets the *LED* bits to this value when it gets control after a CPU reset.

3.3.30 BOOT_LED_REP_WAIT (0x05)

The ROM Boot Module sets the *LED* bits to this value immediately before attempting to load code from the ROM Repository.

3.3.31 BOOT_LED_REP_STORE (0x04)

The ROM Boot Module sets the *LED* bits to this value after reading the synchronization word and header from the ROM Repository, and before loading the code and data contained in the repository's load block.

3.3.32 BOOT_LED_REP_EXEC (0x03)

The ROM Boot Module sets the *LED* bits to this value immediately before attempting to execute code loaded from the ROM Repository.

3.3.33 BOOT_LED_UP_WAIT (0x02)

The ROM Boot Module sets the *LED* bits to this value immediately before attempting to read the first uplink packet during an Uplink Boot.

3.3.34 BOOT_LED_UP_STORE (0x01)

The ROM Boot Module sets the *LED* bits to this value immediately before storing the code received from the first uplink packet during an Uplink Boot and remains in effect while waiting for and storing subsequent packets.

3.3.35 BOOT_LED_UP_EXEC (0x00)

The ROM Boot Module sets the *LED* bits to this value immediately before attempting to execute code loaded from uplink packets during an Uplink Boot.

3.3.36 FS_LED_RUNNING (0x07)

The Flight Software, or uplinked code, should set the *LED* bits to this value to indicate to the ground that the system has successfully booted.

4 CPU Reset Environment

4.1 CPU Registers

When the 80286 is reset, it comes up in Real Mode. The following registers will have the the corresponding values:

FLAGS 0x00002
MSW 0x0FFF0
IP 0x0FFF0
CS Selector 0x0F000
DS Selector 0x00000
ES Selector 0x00000
IDT Base 0x00000

Please refer to the **80286 and 80287 Programmer's Reference Manual** for a description of these registers and their bit fields.

5 Startup Code Environment

5.1 Hardware Devices

When the Boot Module executes the loaded startup code, all hardware devices are in an unknown state. It is up to the startup code to initialize all devices that it requires.

5.2 80286 Register Values

When the Boot Module executes the loaded startup code, most of the registers are in an unknown state. The startup module must initialize all registers that it requires. The exceptions are the code segment and instruction pointer, which point to the first location of the loaded code.

5.3 CPU Stack

The ROM Boot uses a very small, private stack. The startup code should assume that the stack segment and stack pointer are invalid, and should immediately set them to desired values.

5.4 Interrupts

When the Boot Module executes the loaded startup code, CPU Interrupts are left disabled. The 82C59 interrupt controller, however, is in an unknown state. Although the ROM Boot Module fills the interrupt vector table with pointers to the reset vector, the startup code should assume that the interrupt vector table is in an unknown state. Before enabling interrupts, the startup code must set the interrupt vectors and program the 82C59 Interrupt Controller.

6 Detailed Design Considerations

6.1 Data Flow

The following illustrates the overall data flow within the ROM Boot Module:

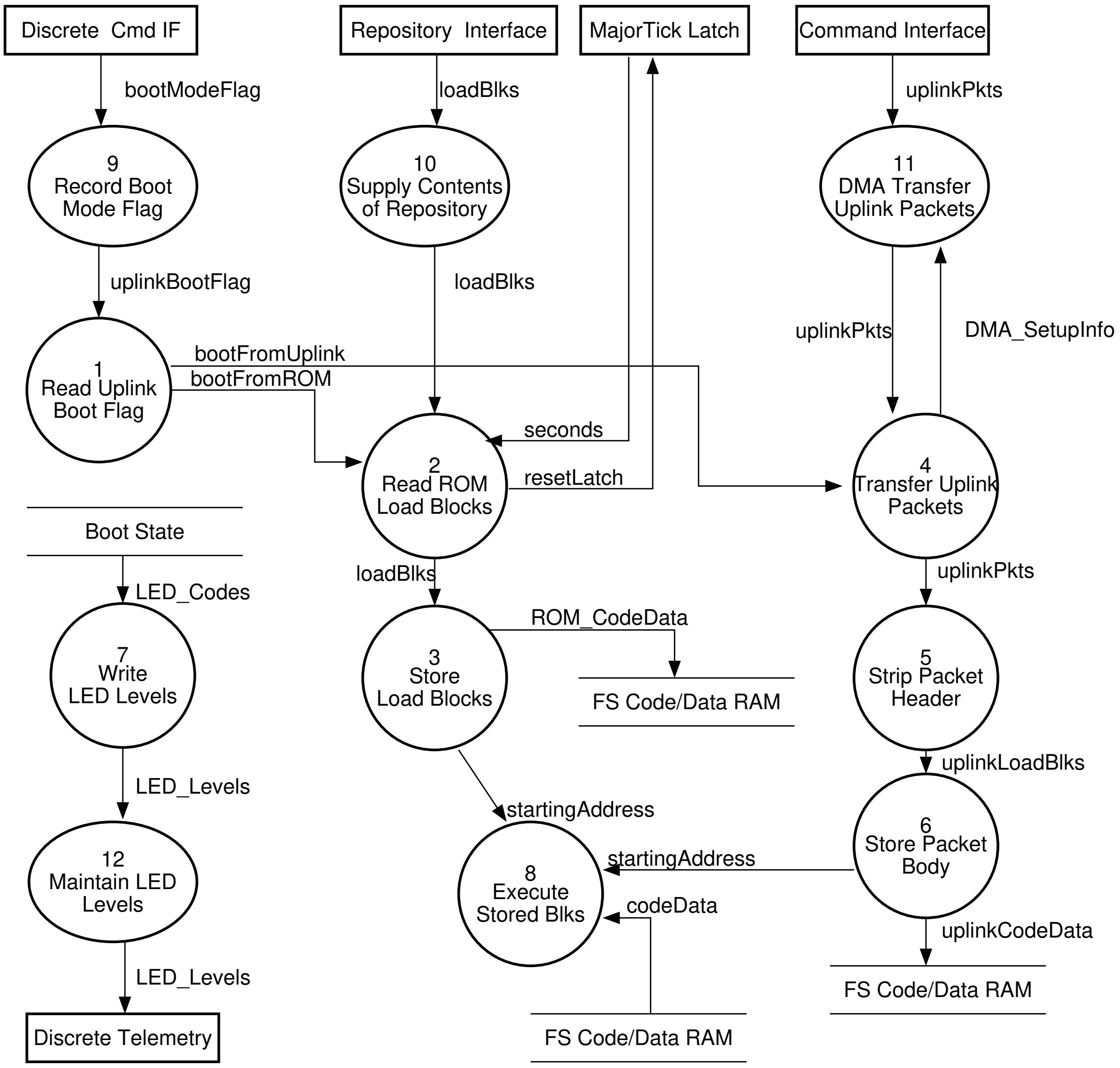

Figure 3 ROM Boot Module Data Flow

Upon reset, the CPU starts executing its ROM Boot. The ROM Boot module decides whether to load code and data from the repository or from the Uplink DMA channel using a flag provided in the CPU Status Register (bubbles 1 and 9). This flag is set and cleared by the ground using discrete commands to the EA hardware.

If the flag indicates a repository boot, the ROM Boot module then searches (not shown) for the ROM Repository synchronization word (see Section 3.3.18), waiting at least 2 second between each attempt, and loads the subsequent code and data from the ROM/Patch Repository interface register (bubbles 2, 3, and 10). This operation must take less than 2 seconds (see **XTE EDS Hardware Functional Description and Requirements**, "Back End Core : Repository"). Once the code is loaded and stored, the ROM Boot module starts executing the loaded code (bubble 8).

If the flag indicates an Uplink Boot, the ROM Boot module programs the DMA controller to transfer command packets into the EA's memory (bubbles 4, 5, 6 and 11). Once all of the boot packets have been transferred, the ROM Boot module starts executing the loaded code (bubble 8).

During each stage of the boot, the system indicates its current stage by setting the LED bits in the CPU Control register to predetermined values (bubbles 7 and 12). The EDS hardware periodically transfers (once every 8 seconds) these bits to the SDS, to be telemetered to the ground as part of the EDS's housekeeping data.

6.2 Control Considerations

6.2.1 Execution States

As the ROM Boot Module performs its operations, it sets the *LED* levels to certain values, to indicate what operation is about to be performed. If, at any point in the boot procedure, the system hangs or waits for an extended period of time (> SDS discrete telemetry polling time), the ground can determine the state of the boot by reading the *LED* values from the discrete telemetry packets sent by the SDS. The following describes the major states during boot, and the *LED* values associated with each state are indicated by the "BOOT_LED_" symbol prefix:

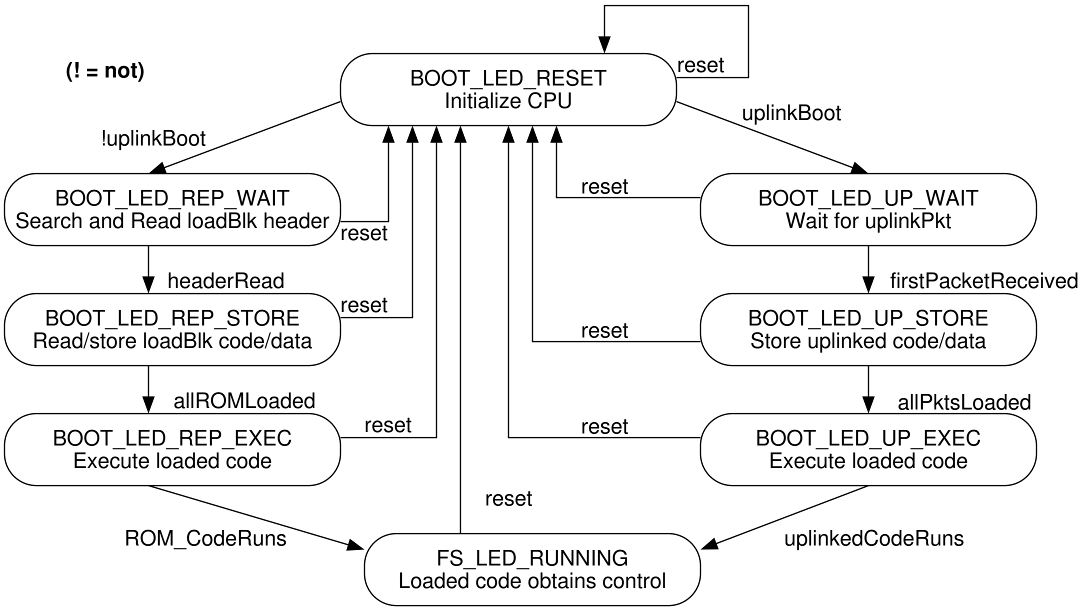

Figure 4 ROM Boot State Diagram

Refer to Section 3.3 for the descriptions and values of BOOT_LED_RESET, BOOT_LED_REP_WAIT, etc.

6.2.2 Potential Pitfalls

The following lists potential problems and pitfalls that may be encountered by the ROM Boot Module:

6.2.2.1 Resets while reading Repository

If one EA continually resets and requests access to the Repository, it can delay the worst case reset-to-grant times of the other EA's in the subgroup. If two or more EA's continually reset, they can conspire to prevent any other EA in the subgroup from obtaining access to the Repository.

If an EA resets in the middle of a Repository load, the ROM Boot Module waits for at least 2 MajorTick's before re-reading and testing the sychronization words at the start of the Repository. The Repository hardware, however, uses a separate 1 second clock to time the 2 second grant window. If the Repository clock is sufficiently slower than the MajorTick clock, and the ROM Boot detects that it is reading from the middle of the repository, it is possible that the ROM Boot will not wait long enough to ensure that the grant window has expired. This will cause the ROM Boot to fail to read the synchronization words and try again. This module attempts to minimize this possibility by waiting for at least 3 MajorTicks to have elapsed (i.e. 4 pulses, see Section 3.3.17).

6.2.2.2 Resets while obtaining Uplink Boot Pkts

If a reset occurs during an uplink boot, while the module is reading the uplink boot packets, the hardware will silently discard the packet. In this situation, the loaded code and data will contain a hole, and will probably crash when executed. As a result, the ground should avoid sending resets during an uplink boot. Since the Boot Module is running from ROM, the probability of a reset due to a Single Event Upset is extremely low (see Appendix A.1).

6.2.2.3 Uplink Boot during Cmd Delivery

If a Flight Software command packet is erroneously delivered to an EA while it is performing an Uplink Boot, the EA will treat the packets as uplink boot packets. The ground must ensure that only uplink boot packets are delivered to the EA when performing an Uplink Boot.

6.3 Module Structure

This section describes the structure of the functions within the ROM Boot Module. The following illustrates the structure of these components:

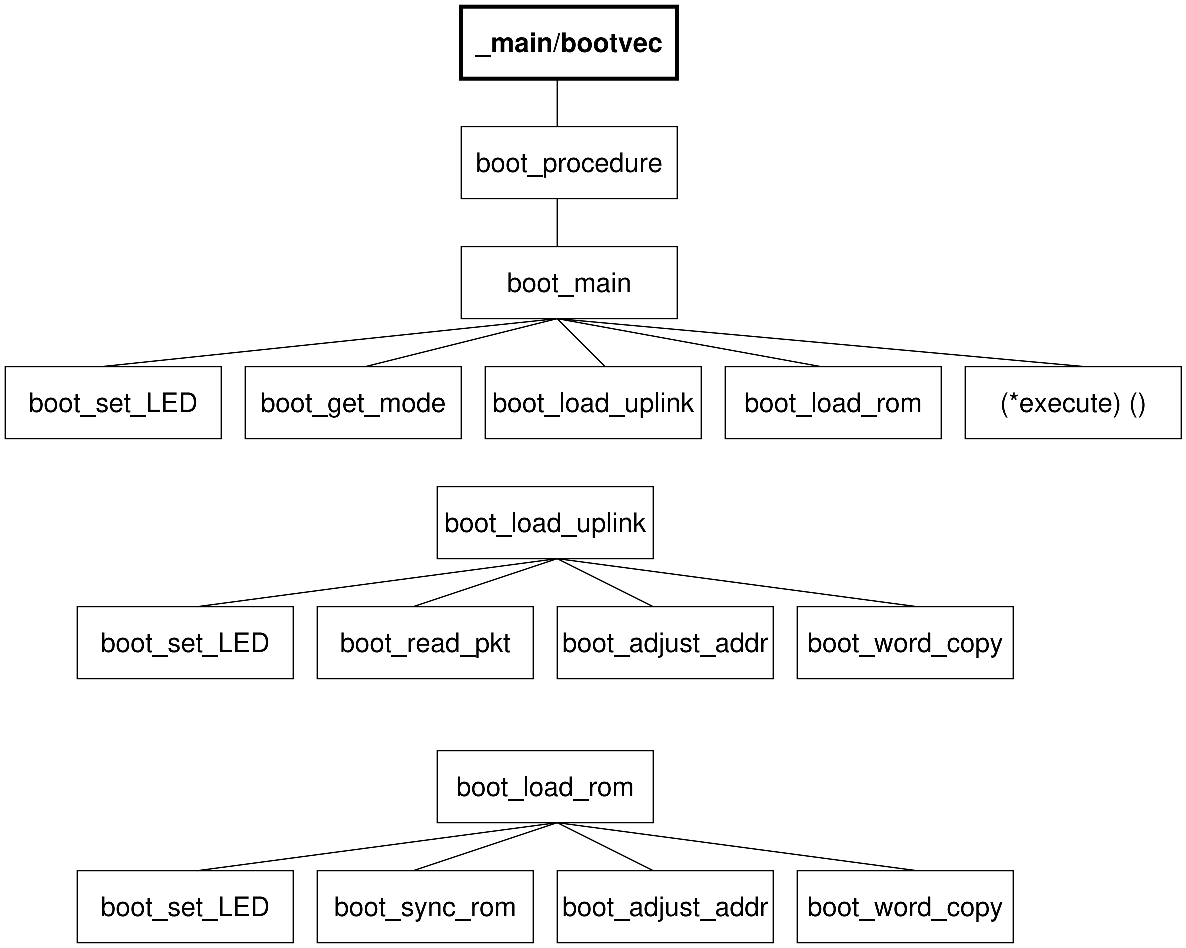

Figure 5 ROM Boot Module Structure

The following lists each of the functions shown in Figure 5. The details of each function's operation and arguments are described in the ROM Boot Module's detailed design document.

_main/bootvec .... Reset Vector Jump Instruction.
boot_procedure .... Initialize Interrupt Vector Table and 80286 Registers
boot_main .... Load and Execute code from Repository or Uplink
boot_load_uplink .... Load code and data from Uplink Packets
boot_load_rom .... Load code and data from ROM/PRAM Repository
boot_set_LED .... Set CPU Control Register LED Bits
boot_get_mode .... Select Boot from Uplink or Boot from Repository

7 Requirements Traceability

7.1 Science Requirements

This module does not address any specific Science Requirements, beyond providing the ability to load and start the Flight Software.

7.2 Interface Requirements

This module satisfies the **XTE EDS Hardware Functional Description and Requirements** interface requirements, specified by the following sections:

<u>Back End (286 CPU) Core :</u>
Backend Memory Map .... Boot ROM and RAM addresses
CPU Control Register .... Used to set LED levels
CPU Status Register .... Used to obtain Uplink Boot Flag
Used to detect end of a single Uplink Transfer
Used to detect MajorTick pulses
CPU Pulse Register .... Used to clear Uplink Interrupt latch
Used to clear MajorTick/MinorTick latches
Repository .... Used to load from ROM/PRAM
DMA Controller .... Used to load from Uplink

This module satisfies the **SDS to EDS Interface Control Document** interface requirements, specified by the following sections:

<u>Command Packet Descriptions</u>
CCSDS Command Primary Header Format ...... Defines Command Packet Header formats

8 Testing Considerations

8.1 Unit Testing

To perform unit tests on this module, the following equipment is required:

a. An Event Analyzer equipped with the Intermetrics ROM Monitor

b. A workstation running the Intermetrics debugger

c. An RS-232 serial cable between the ROM Monitor's serial port and the workstation

d. A logic analyzer

To unit test this module, the tester must link and locate this module's functions to some EA RAM memory location. Using a resident monitor/debugger, the tester should first verify the operation of all functions internal to this module, and then verify the interactions between all functions internal to the module. These tests should verify all branches, calls, and legal and illegal parameters. In addition to these tests, the following items need to be verified:

a. Ensure that once the ROM Repository read has started, the entire repository (128K of ROM + 64K bytes of PRAM) is read within 0.5 seconds (about 400Kwords/second). Since RAM accesses incur 0 memory wait states, but ROM accesses incur 2 memory wait states, this ensures that, once the module is run from ROM, it will be capable of reading the entire repository twice within the 2 seconds allotted by the repository hardware. This test can be performed using a logic analyzer. If the ROM Repository is not available for the test, the module should repeatedly read from a ROM or I/O registers to simulate the wait states incurred by reading from the repository.

b. Ensure that the ROM Repository storage code handles segment boundaries properly.

c. Ensure that uplink boot packets are relocated properly.

d. Ensure that uplink boot storage code handles segment boundaries properly.

e. Verify that no more than 256 bytes of stack space are used for any possible calling sequence in the module.

f. Verify that at least 2 seconds elapse between each attempt to synchronize to the start of the Repository.

8.2 Integration Testing

For integration tests, the following equipment is required:

a. Four Event Analyzer's

b. A populated System Manager board

c. A logic analyzer

d. Hardware which can:

   1. Reset the Event Analyzer

   2. Set the Uplink Boot bit in the CPU Status Register

   3. Deliver uplink boot packets to the DMA controller

   4. Monitor the state of the *LED* levels set in the CPU Control Register

The integration tests should consist of two parts; RAM-based integration tests, and ROM-based integration tests. The RAM-based integration tests use module code which has been linked to execute from RAM. The ROM-based integration tests use module code which is linked to its actual address, and which has been burned into EEPROM.

The RAM integration tests should verify that the module works properly with all of the hardware, and properly interprets the contents of the ROM Repository and uplink boot packets. These tests should include:

a. Verify that a repository boot works. In this case, the tester sets the Uplink Boot flag to zero. The tester should insert breakpoints after each piece of code which sets the *LED* levels, and verify that the levels are set properly. The levels should consist of the following sequence: BOOT_LED_RESET, BOOT_LED_REP_WAIT, BOOT_LED_REP_STORE, BOOT_LED_REP_EXEC. Once the loaded code is running, it should set the *LED*'s to FS_LED_RUNNING.

b. Verify that an Uplink Boot works. In this case, the tester sets the Uplink Boot flag to one, and delivers a sequence of uplink boot packets to the uplink interface. The tester should insert breakpoints after each piece of code which sets the *LED* levels, and verify that the levels are set properly. The levels should consist of the following sequence: BOOT_LED_RESET, BOOT_LED_UP_WAIT, BOOT_LED_REP_STORE, and BOOT_LED_REP_EXEC. Once the loaded code is running, it should set the *LED*'s to FS_LED_RUNNING. This test should also verify the correct operation of the DMA uplink channel.

c. Verify that Uplink Boot can handle a packet delivery rate of at least 32 packets per second.

d. Verify boots from multiple EA's. Retest with one EA continually requesting access to the repository to ensure that the other EA's in the subgroup manage to get access and succesfully boot. Characterize the behavior with more than 1 EA continually requesting access (boots are not required to succeed under this condition).

e. Verify that the ROM Boot Module properly re-synchronizes after a reset during a Repository Boot.

The ROM integration tests should verify that the module works when located and running from ROM. These tests should include:

a. A repository boot, during which the Uplink Boot flag is zero. The module should read, store and execute the contents of the repository. The code loaded and executed from the repository should set and leave the *LED* levels to FS_LED_RUNNING. The logic analyzer should be used to sample the state changes of the *LED* levels. The test should also measure the time it takes to read from the repository. This time must be less than 0.75 seconds. The sampled levels should consist of the following sequence:

   1. One or more samples in an unknown state

   2. One or more samples consisting of BOOT_LED_RESET

   3. One or more samples consisting of BOOT_LED_REP_WAIT

   4. One or more samples consisting of BOOT_LED_REP_STORE

   5. One or more samples consisting of BOOT_LED_REP_EXEC

   6. One or more samples consisting of FS_LED_RUNNING

b. An uplink boot, during which the Uplink Boot flag is one. The module should read, store and execute uplink boot packets. The code loaded and executed from the repository should set and leave the *LED* levels to FS_LED_RUNNING. The logic analyzer should be used to sample the state changes of the *LED* levels. The sampled levels should consist of the following sequence:

   1. One or more samples in an unknown state

   2. One or more samples consisting of BOOT_LED_RESET

3. One or more samples consisting of BOOT_LED_UP_WAIT
4. One or more samples consisting of BOOT_LED_UP_STORE
5. One or more samples consisting of BOOT_LED_UP_EXEC
6. One or more samples consisting of FS_LED_RUNNING

c. Perform a repository loading test, during which the EA is performing a repository boot, while uplink packets are being delivered to the repository hardware. The time taken to load the entire repository should be measured, and must take less than 1 second.

Appendix A

A.1 Single Event Upsets during Boot

This section describes the probability of an Single Event Update (SEU) during Boot. The probability of an SEU hit is 1 upset per 64Kbytes of RAM every 9 days. Since the boot executes entirely from ROM, and for a very short period of time, the effect of SEU hits during boot is negligible. To demonstrate this, consider that the boot uses 256 bytes of RAM for its stack and private data. If the boot were executing continuously, the probability is 1 SEU hit per 256 bytes every 2304 days (~6 years). Given that the ROM Boot executes for no more than 2 seconds (repository time-out period) for each reset, and that each EA is expected to be reset by the ground about 16 times a day, the probability that a given EA is affected during boot by an SEU is extremely small (on the order of 1 in 6 million days). Given that there are 8 EA's, and that the mission is expected to last at least 5 years, the probability that an SEU hit is seen during Boot during the life of the mission is less than 0.5%.