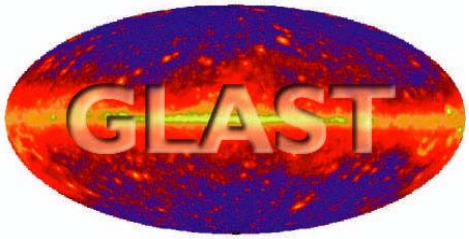


LAT Flight Software

Secondary Boot Code

Type: Design Description
Version: LAT-TD-02150-10
Author: D.L. Wood
Created: 20 May 2003
Updated: 21 October 2004
Printed: 10 December 2004

A description of the LAT secondary boot code for the SIU and EPU CPU boards. The detailed design of the secondary bootstrap code is presented. The RTOS initialization, application loading, and application initialization procedures are discussed.

Document Approval

Prepared By:			
	D.Wood	LAT Flight Software	Date
Approved By:			
	G.Haller	LAT Electronics Manager	Date
Approved By:			
	J.J.Russell	LAT Flight Software Manager	Date

Record Of Changes

Version	Date	Title Or Brief Description	Entered By
0.1	12 May 2003	Draft Release	A.P.Waite
0.3	7 November 2003	Finalize application database binary format. Detailed description of diagnostics features.	D.L Wood
0.4	11 February 2004	Matched description of diagnostics with mnemonics provided in PBC design description document	D.L. Wood
1.0	20 October 2004	Modified memory map definitions to match changes in primary boot code. Updated RAD750 hardware initialization details. Updated error and diagnostics details.	D.L. Wood

Contents

Figures

Tables

0 Introduction

The LAT secondary boot is a EEPROM or RAM based executable which is responsible for initializing the VxWorks RTOS and for loading and initializing the LAT application modules for both the SIU and EPU.

0.0 LAT Boot Code Overview

The LAT RAD750 CPU boards employs a two stage boot process which covers the time span between when the board is reset or powered on and when the application code is initialized. The primary boot code is responsible for the low level initialization of the RAD750 board, the execution of the primary boot code shell, and the loading and execution of a VxWorks RTOS executable image. The LAT secondary boot, discussed in this document covers the initialization of the VxWorks RTOS and the loading and initialization of the LAT application code modules. The two boot processes have been made as independent as possible, so that modifications to the secondary boot code modules may be made without any changes to the primary boot code

0.1 Secondary Boot Code Requirements

1. Initialize the RAD750 RTOS executable and associated board interface libraries.
2. Load the default set of application code modules from a code module database file listing.
3. Call the application code initialization functions, with default parameters, from a code module database file listing.
4. Report diagnostics and errors from the secondary boot process in a global memory area suitable for later examination.

0.2 Reference Documents

VxWorks Programmer's Guide 5.5, Wind River Systems, 1999.

Primary Boot Code, LAT Flight Software Design Description Document.

RAD750 Board Hardware User's Manual, *Document #234A533*, BAE Systems, December 2000.

RAD750 Board Software User's Manual, *Document #234A535*, BAE Systems, April 2001.

PPCI Bridge ASIC Master Errata List, Document #255A651, Rev 1.3, BAE Systems, April 2003.

MPC750 RISC Processor User's Manual, Motorola Inc., 1997.

GLAST Spacecraft Interface Board Hardware Specification, LAT Hardware Specification Document.

MSG User Manual, LAT Flight Software User Manual.

ZLIB User Manual, LAT Flight Software User Manual.

FILE User Manual, LAT Flight Software User Manual.

1 VxWorks RTOS Initialization

The primary boot code execution ends and the secondary boot code execution begins with the loading and execution of the VxWorks RTOS image.

1.0 RTOS Initialization

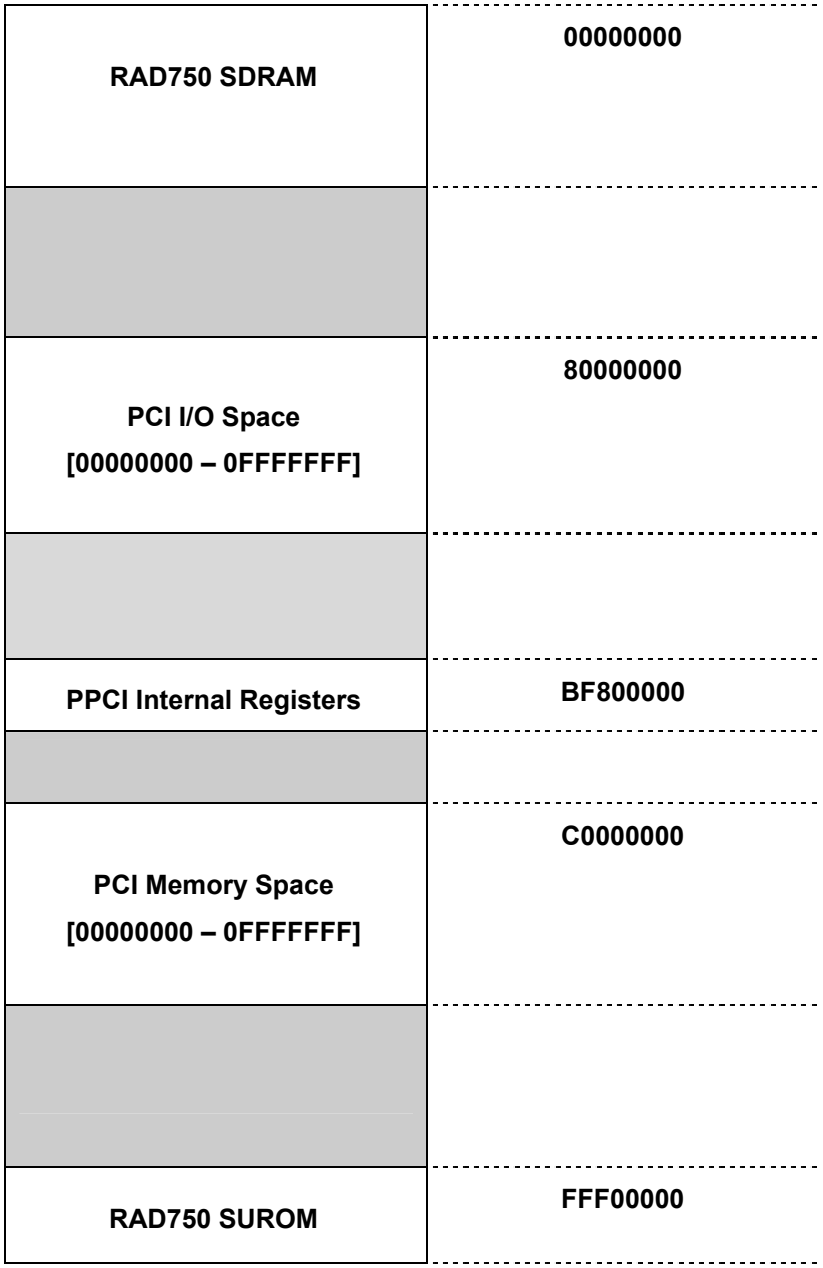
1.0.0 Overview

The RTOS inherits many of the initialization settings from the primary boot code. The base CPU memory map shown in Figure 1 is the result of the RAD750 hardware reset defaults and the EMC and PPC boot code that has already run. The addresses in the right hand column are the CPU virtual addresses for accessing the various memory mapped hardware regions. The RTOS and all subsequent application modules will continue to use this base memory map. Figure 2 shows the SDRAM mapping that the secondary boot code inherits when the RTOS initialization begins. The primary boot code has loaded the RTOS executable segments at address 00030000, where the first instruction of the RTOS initialization code is located (symbol *_sysInit*). The end of the RTOS executable segments is variable, depending of the size of the particular RTOS image that was loaded. At build time, the linker defines the symbol *_end*, which indicates the first available address above the RTOS load region. The boot diagnostics region contains useful information about the status of the primary boot process as well as information that the primary boot code passes to the secondary boot code. The RAM secondary boot module SDRAM regions possibly contain temporary versions of the secondary boot modules.

The RTOS initialization follows the standard VxWorks procedure. The first part of the initialization takes place in the same single threaded environment as the primary boot code. The code for this initial process is contained in the *usrInit()* function. The stack pointer is set to SDRAM address 0000FD80 and grows downward from there. The PCI bus configuration process is completed, and all hardware devices on the RAD750 board are initialized to a quiescent state. The PCI configuration process assigns bus addresses and interrupt levels to the boards found on the cPCI backplane. Once the *usrInit()* function completes, the multithreaded RTOS kernel is started with a call to *kernelInit()*. The call to the kernel initialization function requires that a contiguous region of memory be available to establish the RTOS system memory pool. The address immediately following the end of the RTOS load region (indicated by the symbol *_end*) is selected as the base of the system memory pool. The remainder of the boot region of SDRAM is then dedicated to the system memory pool. This sets the limit of the system memory pool at address 00C00000. The

RTOS initialization SDRAM memory map is shown in Figure 3. Because the primary boot code has already run a memory test on the lower region of SDRAM, up to 00C00000, the RTOS initialization can proceed assuming good working memory. The RTOS never implicitly allocates memory outside of the system memory pool. Any regions above 00C00000 may not have been tested at boot time and might be the responsibility of the applications.

Figure 1 - Secondary Boot Code / Application Hardware Memory Map



Once the kernel initialization has completed, the RTOS creates a special initialization task called *tRoot,* which runs the second RTOS initialization function *usrRoot()*. The remainder of the RTOS initialization process takes place in the context of this task. Once complete, the *tRoot* task exits. Again, the LAT RTOS images follow the standard VxWorks procedure.

Figure 2 - Initial Secondary Boot SDRAM Memory Map

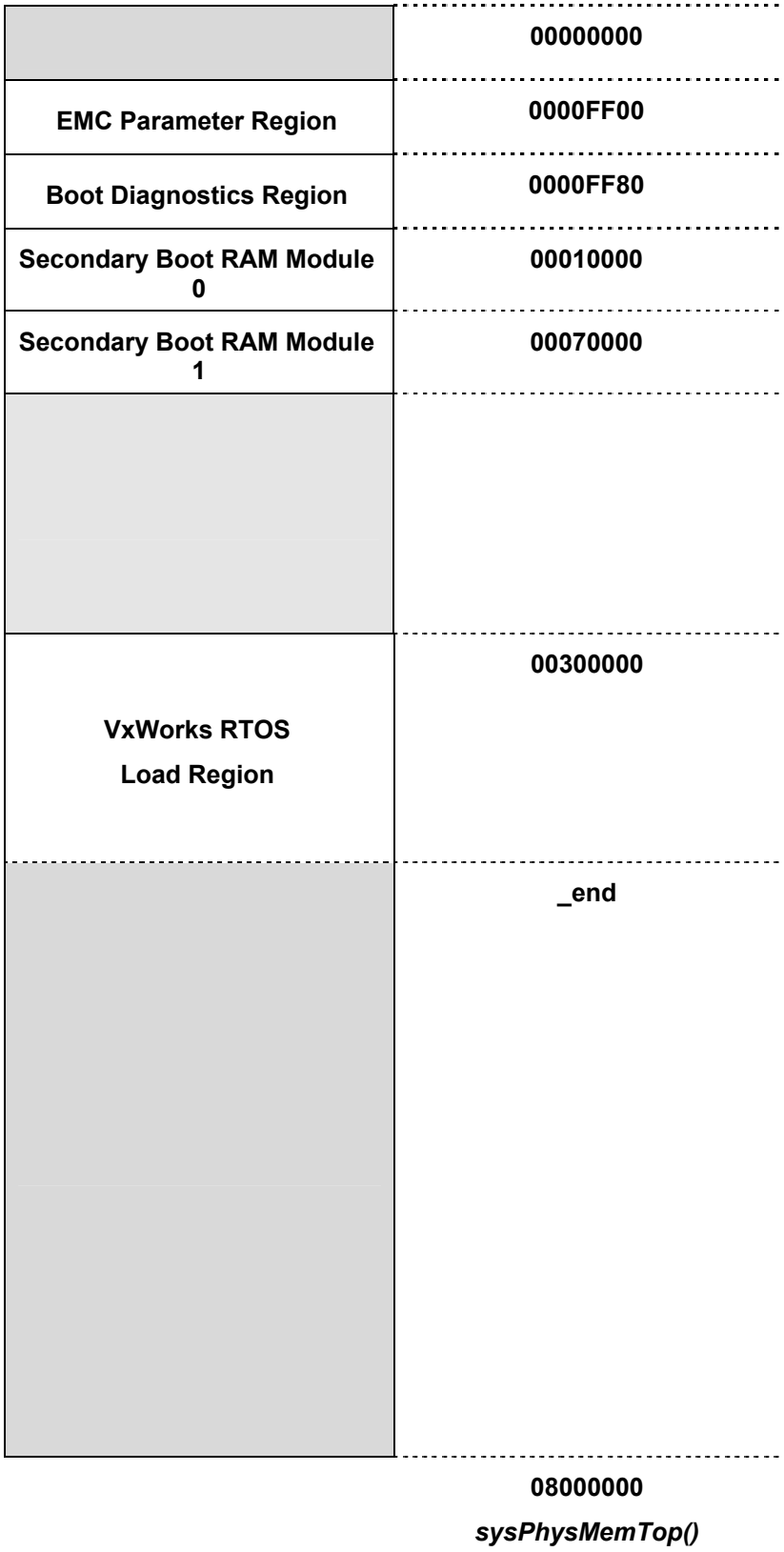
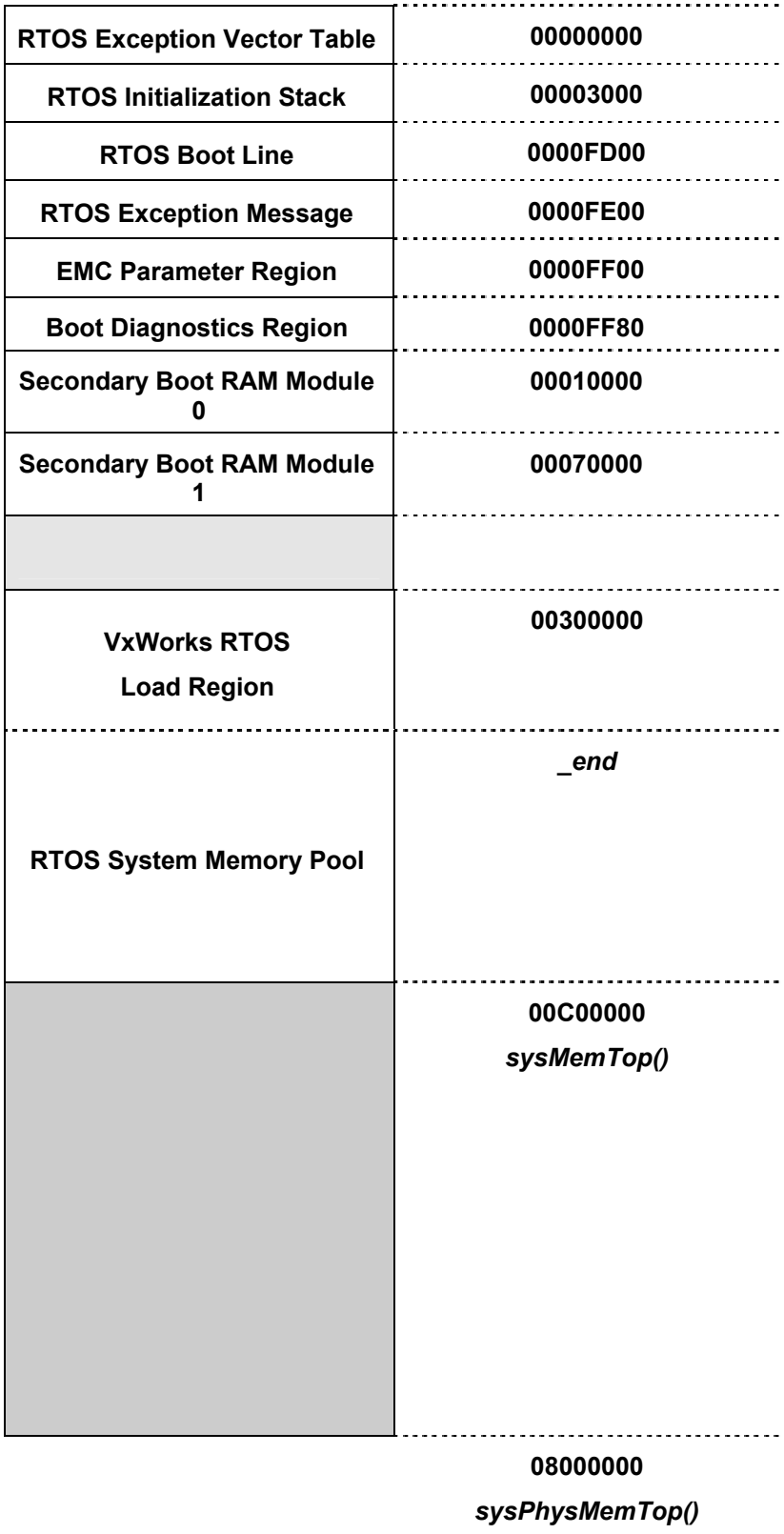
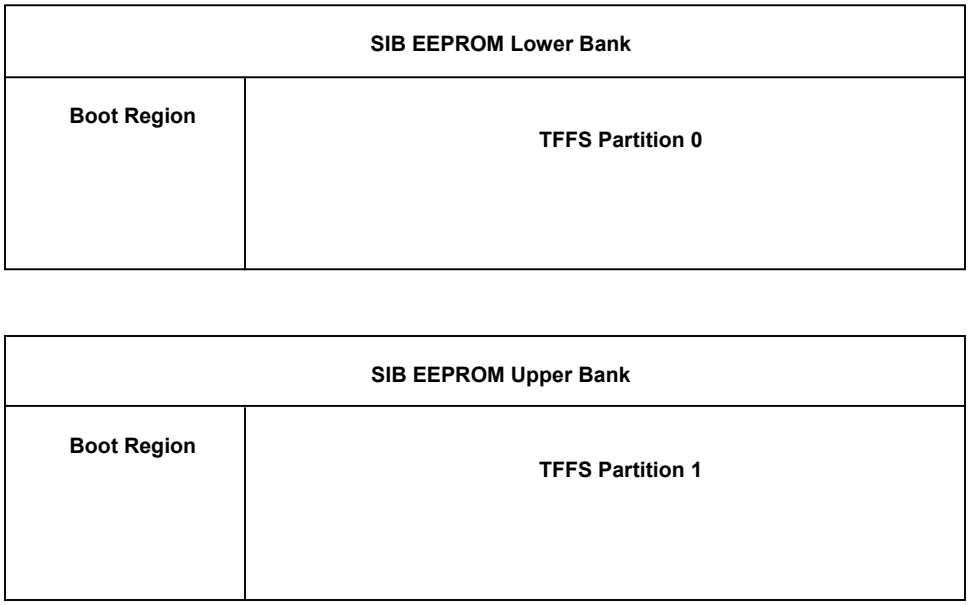


Figure 3 - RTOS Initialization SDRAM Memory Map



Included in the LAT build of VxWorks is the TFFS/DOSFS flash memory file system support. This feature allows the SIB EEPROM regions to be formatted as a standard file system. The application code modules and data objects are stored in EEPROM as files. Figure 4 below shows the setup of the of the SIB EEPROM.

Figure 4 - SIB EEPROM Layout



The first portion of an SIB EEPROM memory bank may be reserved for use by the primary and secondary boot codes and is not formatted by TFFS. The details of the boot region are shown in Figure 5. The remainder of the bank EEPROM is formatted as a TFFS/DOSFS file system. The RTOS contains TFFS drivers for the SIB EEPROM banks. The lower EEPROM bank is registered as TFFS drive number ‘0’, and the upper EEPROM bank is registered as TFFS drive number ‘1’.

The *usrRoot()* function finishes the initialization of the RAD750 board internal hardware and creates all of the RTOS objects and resources needed by the LAT application software. It is at the end of the RTOS initialization procedure that the LAT secondary boot process truly begins. The *usrRoot()* function, as a last step, will load secondary boot module 0 and begin execution of the code contained in that module.

1.0.1 RAD750 Hardware Initialization

In the RTOS initialization functions *sysHwInit()* and *sysHwInit2()*, many detailed operations are performed in configuring the RAD750 board hardware, particularly the PPCI bridge chip. The following list provides a time sequence of the RTOS hardware initialization.

1. The BSP function *excHandler()* is installed as the global exception hook. VxWorks will route all exception conditions to this function. The handler traps all exceptions, forcing a panic reboot after dumping the exception information into the boot diagnostics area.
2. Clear the cPCI bus reset signal (RST#) by clearing bit 0 in the PPCI Bus Reset Register (BF870065). This ensures that the boards on the cPCI backplane and that the cPCI interface of the PPCI are enabled.
3. Disable cPCI Master Abort transactions from generating a machine check by setting bit 7 in the PPCI Configuration Register (BF87005C). This prevents bus scans from generating a machine check.
4. Set bit 0 in PPCI User Defined A Register (BF870070) to enable 60X bus error signaling.
5. Enable PPCI OCB errors to generate machine checks by clearing bit 4 in the PPCI Error Status Mask Register (BF810014). Make sure that PPCI P60X errors do not generate machine checks by making sure bit 3 remains set. This is a workaround for PPCI Errata #7.
6. Enable processor machine checks by setting bit 11 in the PPCI User Defined B Register (BF8700A8).
7. The BSP function *sysMemProbe()* is installed as the RTOS memory probe callback. This function temporarily disables machine checks so that controlled probes to unknown addresses do not cause exceptions.
8. Disable PPCI OSR Ordering by setting bit 4 in the PPCI Configuration Register (BF87005C). This is a workaround for PPCI Errata #12.
9. Set the BAR1 Speculative PCI Read bit and the BAR1 Prefetch Enable bit (bits 24 and 25) in the PPCI Configuration Register (BF87005C). This improves DMA performance between the RAD750 SDRAM and the cPCI bus.
10. Set PPCI to ignore the bus master's Latency Timer by setting bit 2 in the PPCI Configuration Register (BF87005C). This is a workaround for PPCI Errata #13.
11. Disable writes to the PPCI internal registers from cPCI bus by setting the BAR2 Write Disable bit (bit 20) in the PPCI Configuration register (BF87005C). This feature is not needed by the LAT and could prevent bad DMA addresses from corrupting the register settings.
12. Set the PPCI Cacheline Size Register (BF87000C) to 08.
13. Enable PCI protocol checking, PCI data phase timeouts, and PCI arbitration timeouts in the PPCI Error Checking Regsiter (BF870062) by setting bits 0, 3, and 10. Set the arbitration latency timeout value to the maximum of 2048 bus clocks by writing '3' to bits 2:1 in the Error Checking Regsiter.
14. Map the RAD750 board 128 MB of SDRAM to PCI Memory Address 40000000 by setting the PPCI Base Address 1 Register (BF870010) to 40000000. This is the PCI address which is used by cPCI masters to access the RAD750 main memory.
15. Connect the PPCI to the cPCI bus by setting bits 1, 2, 6, and 8 in the PPCI Control Register (BF870004). This enables memory accesses from the RAD750 to the cPCI bus and also allows targets on the cPCI bus to access the RAD750 SDRAM through DMA. This also enables cPCI error checking in the bridge and allows the generation of the #SERR signal upon error.
16. Enable machine check signals to the PPC processor by setting bit 0 in the PPC HID0 register.

17. Disable all PPCI interrupts by clearing the PPCI Interrupt Enable Register (BA280004).
18. Connect the BSP function *ppciInt()* to the PPCI master interrupt level. This function dispatches interrupts signaled in the PPCI Interrupt Collection Register (BA280000).
19. Setup an interrupt dispatch for the PPCI miscellaneous interrupt level. The Misc interrupt level is enabled by setting bit 22 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signaled in the PPCI Misc Interrupt Status Register (BA280094).
20. Disable all PID interrupts by clearing the PPCI PID Interrupt Enable Register (BA28001C).
21. Setup an interrupt dispatch for the PPCI PID interrupt level. The PID interrupt level is enabled by setting bit 15 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signaled in the PPCI PID Input Register (BA280024).
22. Disable all multiprocessor interrupts by clearing the PPCI MP Signaling Enable Register (BA280088).
23. Setup an interrupt dispatch for the PPCI multiprocessor interrupt level. The MP interrupt level is enabled by setting bit 14 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signaled in the PPCI MP Signaling Register (BA280080).
24. Disable all PCI error interrupts by clearing the PPCI PCI Status 2 Mask Register (BF870054).
25. Setup an interrupt dispatch for the PCI error interrupt level. The PCI error interrupt level is enabled by setting bit 31 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signalled in the PPCI PCI Status 2 Register (BF870050).
26. Install a default handler for all of the PCI error interrupts except levels 0 and 4. The default handler logs information to the serial console. Enable these interrupt levels.
27. Disable all memory controller interrupts by clearing the PPCI Memory Interrupt Enable Register (BF8000C4).
28. Setup an interrupt dispatch for the memory controller interrupt level. The memory controller interrupt level is enabled by setting bits 25 and 26 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signalled in the PPCI Memory Status Register (BF8000C0).
29. Install a default handler for all of the memory controller interrupts except levels 0 - 2. The default handler logs information to the serial console. Enable these interrupt levels.
30. Disable all P60X bus error interrupts by setting bits 0, 1, and 2 in the PPCI P60X Error Mask Register (BF810014).
31. Setup an interrupt dispatch for the P60X bus error interrupt level. The P60X bus error interrupt level is enabled by setting bit 30 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signalled in the PPCI P60X Error Status Register (BF810010).
32. Install a default handler for all of the P60X bus error interrupts except levels. The default handler logs information to the serial console. Enable these interrupt levels.
33. Setup an interrupt dispatch for the PPCI multiprocessor interrupt level. The MP interrupt level is enabled by setting bit 14 in the PPCI Interrupt Enable Register (BA280004). These interrupts are signaled in the PPCI MP Signaling Register (BA280080).

34. Set the PPCI RTC divisor to ‘4’ by setting bits [3:2] in the PPCI Clock Control Register (BF8600000). This sets the RTC clock frequency to 8.25 MHz.

1.1 Loading the Secondary Boot Executable

The secondary boot process requires two modules to be stored in either EEPROM or a temporary SDRAM buffer. Descriptions of the two modules are listed below.

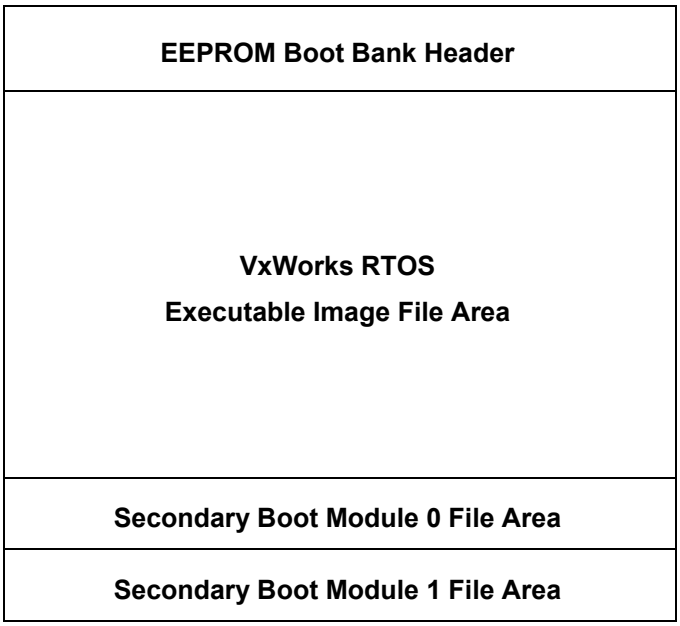
Table 1 - Secondary Boot Code Modules

Module	Description
0	Secondary boot executable. An ELF object file which is loaded at the end of the RTOS initialization process. Reads the application data base contained in module 1 and initializes the application code modules
1	Secondary boot application database. An application data base file which contains a list of the application ELF object files to be loaded. Also contains a listing of the applications initialization function symbols names and a short listing of parameter values to pass to the initialization functions.

The secondary boot process has the option of sourcing these modules from alternate locations. The Secondary Boot Flags (*DIAGS_SEC_BOOT_FLAGS*) member of the boot diagnostics region in SDRAM provides the direction the secondary boot code should take (see Table 4). If set, the *DIAGS_SBF_XXX_SOURCE* flags indicate that the files are to be taken from EEPROM. If clear, the flags indicate that the secondary boot code sould use the temporary versions in the SDRAM buffers. The values for these flags are set by the primary boot code when the RTOS Execute command is invoked. The RTOS file source information is provided for information only, since the RTOS always executes out of an SDRAM area of memory.

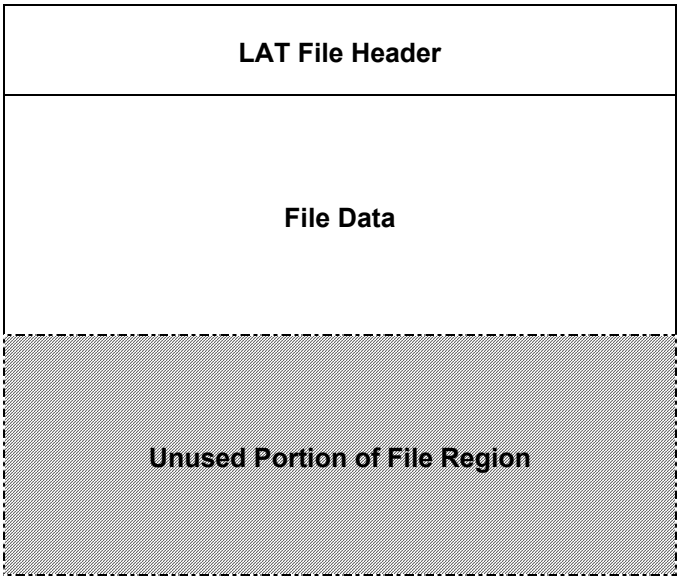
Since the secondary boot modules may be contained in either SDRAM or EEPROM, the first action of the secondary boot executable load process is to establish a mapping to EEPROM. The secondary boot code modules are stored at offsets in the boot region of the SIB EEPROM lower bank, shown in a typical arrangement below in Figure 5. The bank header provides information about the offsets and sizes of each of the file regions in the boot partition.

Figure 5 - EEPROM Boot Region Memory Map (Nominal)



Each one of the file areas in the boot region can contain a single file. The format of each file region is shown below.

Figure 6 - EEPROM Boot Region File Layout



Regardless of whether the secondary boot code module 0 is stored in EEPROM or SDRAM, the last step of the RTOS initialization will copy the file contents into a temporary buffer and invoke the VxWorks ELF loader to link in the secondary boot executable sections. If the file header indicates that the file is stored in ZLIB compressed format, the file is first inflated before running the module loader. Next, the system symbol table is queried for the symbol name *SBC_init*, which is the name of the secondary boot code executable initialization function. This symbol should have been entered into the system table by the executable load process. The *SBC_init()* function is then called.

2 Application Initialization

Once the LAT secondary boot code executable has initialized, it performs two fundamental steps in the initialization of the LAT application flight software. First, the RAM and EEPROM file systems are either created or mounted. Second, the secondary boot code file module 1 is read in. This file contains the information needed to find the application code module files in the newly mounted EEPROM file systems and to perform all of the calls to the application initialization functions.

2.0 Application Memory Setup

The LAT secondary boot code uses the VxWorks system memory pool for all of its dynamic allocation needs. Setup of the application memory pool starting at address 00C00000 is deferred until the application modules have been loaded and initialized.

2.1 Secondary Boot Flags

The secondary boot flags member of the boot diagnostics region contains information about how the secondary boot code should act when initializing (see Table 4). The bits of the secondary boot flags word are listed below.

Table 2 - Secondary Boot Flags

Bits	Mnemonic	Description
0 - 1	DIAGS_SBF_RTOS_SOURCE	RTOS image source flag indicates the location from which the RTOS executable was taken when starting the secondary boot process. 0 = RTOS image was sourced from RAM temporary area; 1 = RTOS image was sourced from SIB EEPROM lower bank boot partition; 2 = RTOS image was sourced from SIB EEPROM upper bank boot partition.
1 - 2	DIAGS_SBF_MOD0_SOURCE	Flags indicate the location from which the secondary boot process should source module 0. 0 = SSB Module 0 should be sourced from RAM temporary area; 1 = SSB Module 0 should be sourced from SIB EEPROM lower bank boot parition; 2 = SSB Module 0 should be sourced from SIB EEPROM upper bank boot parition

3 - 4	DIAGS_SBF_MOD1_SOURCE	Flags indicate the location from which the secondary boot process should source module 1. 0 = SSB Module 1 should be sourced from RAM temporary area; 1 = SSB Module 1 should be sourced from SIB EEPROM lower bank boot partition; 2 = SSB Module 1 should be sourced from SIB EEPROM lower bank boot partition
5 - 15		Reserved.
16	DIAGS_SBF_MOUNT_EE0	Flag indicates that the secondary boot code should attempt to mount the lower bank EEPROM TFFS filesystem.
17	DIAGS_SBF_MOUNT_EE1	Flag indicates that the secondary boot code should attempt to mount the upper bank EEPROM TFFS filesystem.
18 - 31		Reserved.

The secondary boot flags word is set when the primary boot code begins execution of the RTOS. The value of the flags word is taken from the Boot RTOS Start telecommand, which is sent to the primary boot code to begin execution of the secondary boot code. The primary boot code will examine bits 0 – 1 to determine which RTOS image to execute. The first stage of the secondary boot code will examine bits 1 – 2 to determine where to find the secondary boot executable file (see Section 1.1). Bits 3 – 4 determine where the secondary boot executable will find the application intialization database file (see Section 2.4). Bits 16 – 17 indicate which of the TFFS EEPROM filesystems the secondary boot code should attempt to mount (see Section 2.2.1).

2.2 Mounting the File Systems

The LAT secondary boot code creates or mounts the VxWorks file systems needed for flight operations.

2.2.0 Creating the RAM Disk

A standard VxWorks RAM disk formatted with the DOSFS file system is created out of the system memory pool. The RAM disk is 1 MB byte in size. Once created and formatted, the RAM disk is mounted as device "*/ram*".

2.2.1 Mounting the TFFS Partitions

The portion of the SIB lower EEPROM bank not reserved for the boot codes is scanned for a valid TFFS/DOSFS formatted partition. If found, the partition is mounted as device "*/ee0*". The SIB upper EEPROM bank is scanned for a valid TFFS/DOSFS formatted partition. If found, the partition is mounted as device "*/ee1*". Either of these two steps may be skipped by clearing the *DIAGS_SBF_MOUNT_EEx* flags bits in the secondary boot flags word (see Table 2).

2.3 Application Module File Formats

The secondary boot code deals with two types of files in the on-board filesystems: the application database file and the application software object module files. In both cases, the file must be prefixed with the standard LAT file header in order to pass the secondary boot code validation

process. Also in both cases, the file data may be optionally compressed with the ZLIB encoder. The file data compression bit in the file header indicates whether or not the file data is compressed. The figures below show the uncompressed and compressed formats of the files.

Figure 7 - Uncompressed File Format

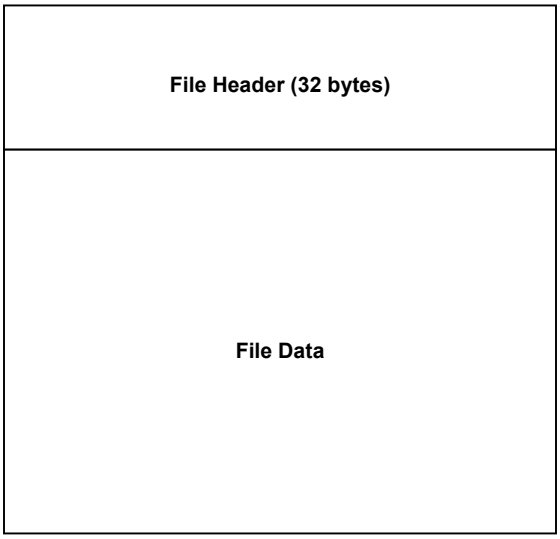
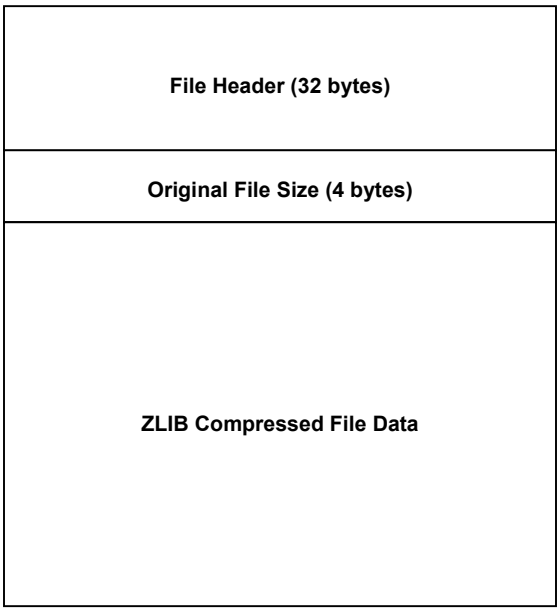


Figure 8 - Compressed File Format



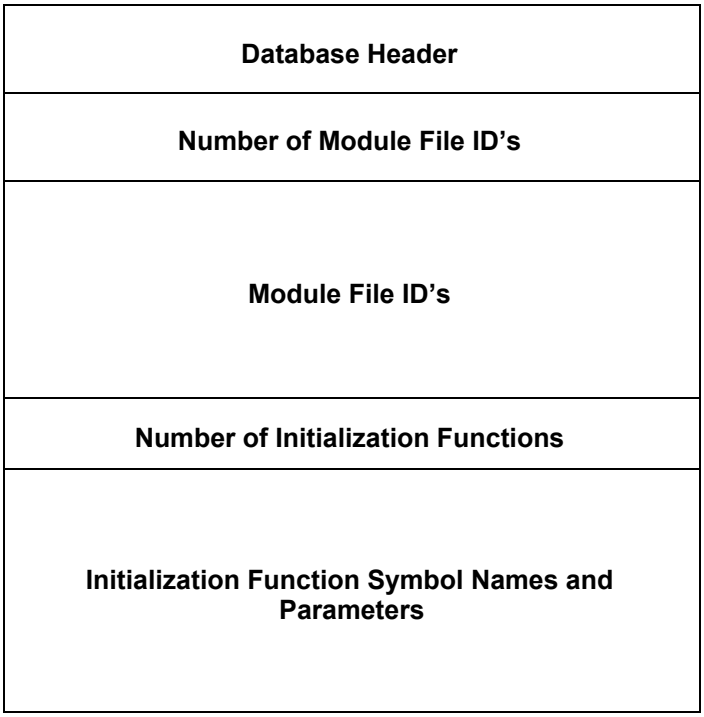
The file header is fixed size and always uncompressed. If the data is compressed, it follows the LAT ZLIB basic compressed format. A four byte header precedes the compression stream. This header value indicates the size in bytes of the original file data which follows in compressed form. The standard ZLIB compression stream follows the four byte header. In the case of the

application database file, the file data content is described in Section 2.4. In the case of the application software object module files, the file data content is a standard PPC ELF relocatable object file as produced by the VxWorks 5.5 tool set. This file content is readable by the VxWorks module loader.

2.4 The Application Database

The secondary boot application database is stored as a single file in the second stage module area, either in the EEPROM boot region or in the SDRAM temporary buffer. The top level format of the application database file is shown in Figure 9.

Figure 9 - Application Database File Format



Essentially, the database contains two distinct components. The first component contains a list of the application module files that need to be loaded. Each entry contains one file ID value which directs the module loader to the file system paths of the module files. The second component contains a list of the application initialization function symbol names. These symbols are referenced in the VxWorks system symbol table. Also, each initialization entry contains a small number of parameter values which may be passed to each initialization function. Because the initialization functions are not available until their respective code modules have been loaded, the application database is always parsed beginning with the module list.

All of the sections of the database file are aligned to four bytes. The counts of elements are stored as 32-bit, big-endian words.

The database header consists of one 32-bit word, shown below.

Figure 10 - Application Database Header Word

'S'	'B'	'C'	Version Number

The version number of the database format described in this document is '1'.

2.4.0 Loading the Applications Modules

The number of application modules to load is the first 32-bit word in the application database file. Each module file ID is stored as a 32-bit, big-endian value. The application code modules are stored in ELF format, possibly compressed by the ZLIB utility. Thus, the secondary boot code might first inflate each code module in the list before it is loaded. It is the responsibility of the application database provided to ensure that the modules in the list are represented in correct load order for symbol dependencies. The secondary boot code always loads the modules in the order listed in the database file.

Right after an application object module is loaded, the secondary boot code record information about the module in the CMX run-time constituent list. This enables users to link together the CMX constituent build-time information to the VxWorks module run-time information. The CMX constituent user data fields are used by the secondary boot code as follows:

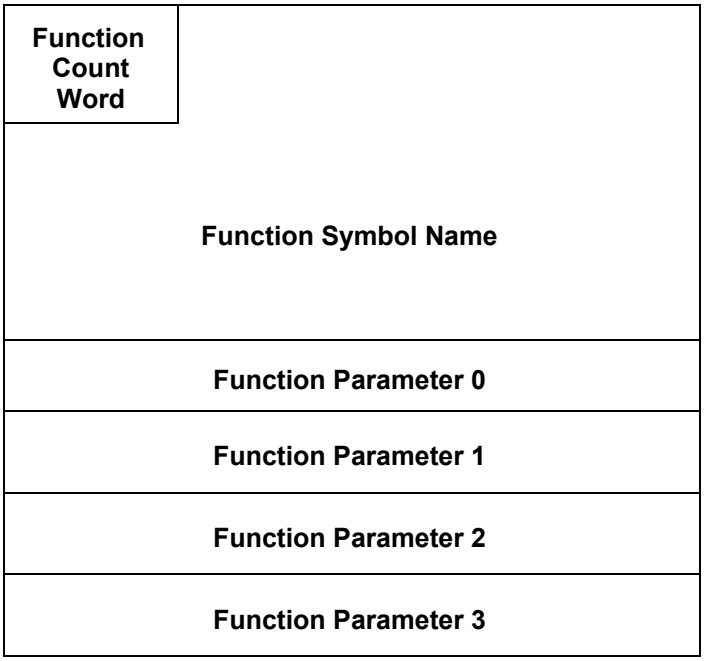
Table 3- Secondary Boot Code CMX User Data Values

CMX Data Word	Enum Value	Description
0	*CAB_K_MODULE*	The VxWorks module ID (type *MODULE_ID*) of the application module.
1	*CAB_K_FILE*	The 32-bit file storage ID of the module file.
2	*CAB_K_SPARE_0*	Reserved
3	*CAB_K_SPARE_1*	Reserved.

2.4.1 Calling the Application Initialization Functions

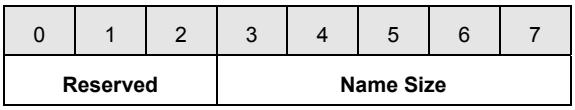
Immediately following the last module file ID entry is the 32-bit count of the number of initialization function entries following in the remainder of the application database. The detail of each initialization function entry is shown in the figure below.

Figure 11 - Application Initialization Function Database Entry



As can be seen, each application initialization function is allowed four settable parameters. Each initialization function symbol name is limited to 31 characters. An 8-bit count word precedes each symbol string, providing the character count for the string. The ASCII string character bytes follow. The format of the count word is show below.

Figure 12 - Application Initialization Function Count Word



Name Size – The number of characters in the function symbol name string

The symbol name will be padded with '\0' characters to make the total length of the symbol name storage be 32-bit aligned. The symbol name size counter, however, always provides the true length of the initialization function symbol names. The secondary boot code always calls the initialization functions in the order they are listed in the application database file. Following the function name string is a list of 4 32-bit, big-endian words. These are the parameter values for the named function.

Each function listed in the initialization script must have been contained in one of the object modules loaded by the same script or must have been contained in the RTOS executable. Each function should have the following prototype:

```
unsigned int AppInitFunction(unsigned int p0, unsigned int p1, unsigned
      int p2, unsigned int p3);
```

The function is called with values *p0*, *p1*, *p2*, and *p3* from the function database entry. When the function execution completes, the secondary boot code examines the return value from the function. The secondary boot code expects tha the return value is formatted as a MSG code. A MSG severity of level **ERROR** indicates an error which will halt the secondary boot process. In such a case, the function return value is recorded in the SDRAM boot diagnostics region, the secondary boot error bit in the primary boot flags is set, and a warm reboot is executed (see Section 3.1). The return value from the failing application initialization function may then be examined using the primary boot code memory dump facility.

3 Diagnostics and Errors

The LAT secondary boot process stops the execution of the primary boot code. This disables the output of boot telemetry. Therefore, until the application level communications software has been loaded and initialized, the secondary boot code has no means of reporting diagnostic and error information through a communications channel while it is executing. In these cases, the secondary boot code must write information into the SDRAM boot diagnostics region, and if the error is critical, perform a warm reboot so that the error information can be examined through the primary boot code shell telemetry.

3.0 Diagnostics

The LAT secondary boot code uses the SDRAM diagnostics area at address 0000FF80 to provide some detail on the symptoms of errors encountered. In the case of a critical error, where a warm reboot action is taken, the secondary boot code leaves behind information in the diagnostics region that can be examined after reboot by the primary boot code memory dump. The layout of the boot portion of the SDRAM diagnostics area is described in the primary boot code document.

The secondary boot code uses the LAT flight software MSG tool to generate value, macros, and documentation notes for the various error and information codes. At runtime, however, the MSG codes are delivered in reserved areas of the SDRAM diagnostics area instead of through normal telemetry channels. Memory dump tools may be used to gather the information.

The secondary boot code utilizes multiple words in the SDRAM diagnostics area to report error and progress conditions.

Table 4 - SDRAM Boot Diagnostics Area

Offset (bytes)	Mnemonic	Description	Data Source
0x04	DIAGS_SEC_BOOT_FLAGS	Secondary Boot Flags.	Set by PBC from parameters specified in the Boot RTOS Execute Telecommand.
0x0C	DIAGS_EXC_VEC	Exception Vector	Exception handler.
0x10	DIAGS_EXC_SRR0_REG	Exception SSR0 Register Contents	Exception handler.
0X14	DIAGS_EXC_SRR1_REG	Exception SSR1 Register Contents	Exception handler.
0X18	DIAGS_EXC_DAR_REG	Exception DAR Register Contents	Exception handler.
0X1C	DIAGS_EXC_DSISR_REG	Exception DSISR Register Contents	Exception handler.
0x20	DIAGS_PCI_STATUS2_REG	Exception PCI Status 2 Register Contents	Exception handler.
0x24	DIAGS_MEM_STATUS_REG	Exception Memory Status Register Contents	Exception handler.
0x28	DIAGS_EXC_TASK_ID	Exception task ID.	Exception handler.
0x44	DIAGS_SEC_BOOT_FAIL_ERR	Secondary Boot Error Code	Secondary boot.
0x48	DIAGS_SEC_BOOT_FAIL_IDX	Secondary Boot Error Index	Secondary boot.
0x4C	DIAGS_SEC_BOOT_FAIL_DATA	Secondary Boot Error Data	Secondary boot.
0x60-0x7C	DIAGS_APP_INFO	Application Information	Secondary boot or Application.

The Secondary Boot Error Code (*DIAGS_SEC_BOOT_FAIL_ERR*) member provides the general type of error encountered. The Secondary Boot Error Index provides an enumeration of the various stages of the secondary boot process. Its value indicates where in the secondary boot process the error occurred. The Secondary Boot Data word (*DIAGS_SEC_BOOT_FAIL_DATA*) contains additional information describing the error. Its meaning depends on the type of error reported.

The Secondary Boot Index (*DIAGS_SEC_BOOT_FAIL_IDX*) member of the diagnostics region is updated as the secondary boot code progresses through its various actions. If an error is signaled, the index member provides insight into the source of the error. The first set of index

codes relate to the progress of the secondary boot code at the end of the RTOS initialization (Section 1.1).

Table 5 - Secondary Boot RTOS Index Codes

MSG Facility	MSG Code (DIAGS_SEC_BOOT_FAIL_IDX)	Description
VXW	VXW_IVXBNKH	Reading the EEPROM boot bank header.
	VXW_IVXEXH	Reading SBC module 0 file header.
	VXW_IVXEXF	Reading SBC module 0 file data.
	VXW_IVXEXI	Inflating SBC module 0 file.
	VXW_IVXEXL	Loading SBC module 0 file.

The second set of index codes relate to the progress of the secondary boot code through the application initialization (Section 2).

Table 6 - Secondary Boot Application Index Codes

MSG Facility	MSG Code (DIAGS_SEC_BOOT_FAIL_IDX)	Description
SBC	SBC_IMNTRAM	Mounting RAM disk partition.
	SBC_IMNTEE	Mounting TFFS EEPROM partitions.
	SBC_ILDBNKH	Reading the EEPROM boot bank header.
	SBC_ILDDBH	Reading SBC module 1 file header.
	SBC_ILDDBF	Reading SBC module 1 file data.
	SBC_ILDDBI	Inflating SBC module 1 file.
	SBC_ILDDBM	Loading application module files.
	SBC_ILDDBS	Calling application initialization functions.

If a reboot is required, then the secondary boot code will also modify the Primary Boot Flags member (*DIAGS_PRIM_BOOT_FLAGS*) of the diagnostics region.

3.1 Errors

If an error encountered during the secondary boot process warrants a reboot, the secondary boot code sets the Secondary Boot Error Flag (*DIAGS_PBF_SEC_RESET*) in the Primary Boot Flags member, clears all other Primary Boot Flags, and performs a warm reboot at address FFF00104. This returns execution to the SUROM base primary boot code. The memory dump facilities of the primary boot shell may then be used to diagnose the error report information left over by the secondary boot code.

Errors may be reported as the RTOS BSP initialization process enters the secondary boot phase. The error MSG codes and values for the Secondary Boot Info word are shown below.

Table 7 – Secondary Boot Error Codes

MSG Facility	MSG Code (DIAGS_SEC_BOOT_FAIL_ERR)	Description	Data Word (DIAGS_SEC_BOOT_FAIL_DATA)
SBC or VXW	XXX_SUCCESS	Success.	None.
	XXX_EMEMALOC	Memory allocation error (from system memory partition using *malloc()*)	The number of bytes requested in the allocation.
	XXX_EEEREAD	EEPROM read error (using *tffsRawIo()*)	The address the read started at.
	XXX_EFILHDR	SBC module file header validation error (using *FILE_hdrVerify()*).	The return value from *FILE_hdrVerify()*.
	XXX_EFILFRMT	SBC module file data format or version error.	The value of the first 32-bit word of the file header.
	XXX_EFILCKSM	SBC module file checksum validation error.	File checksum computed value.
	XXX_EFILSIZE	An application module file is too large to load.	The size in bytes of the module object file.
	XXX_EFILOPEN	SBC module file open error (using *open()*).	The file ID storage value of the file being opened.
	XXX_EFILREAD	Module file read error (using *read()*).	The value of the VxWorks *errno* variable.

	XXX_EFILWRIT	Module file write error (using *write()*).	The value of the VxWorks *errno* variable.
	XXX_EMEMDEV	Memory device creation failure (using *memDevCreate()*).	The value of the VxWorks *errno* variable.
	XXX_EFILLOAD	SBC module file module load error (using *loadModule()*).	The file ID of the module file.
	XXX_EFILSYM	Module file symbol lookup failure (using *symFindByName()*).	If the index value (*DIAGS_SEC_BOOT_FAIL_IDX*) is *SBC_ILDDBS*, this is the index of the function symbol name in the database function list.
	XXX_EZLIBINI	ZLIB initialization error (using *inflateInit()*).	The return value from *inflateInit()*.
	XXX_EZLIBINF	ZLIB inflation error (using *inflate()*).	The return value from *inflate()*.
	XXX_ESYSRAM	RAM disk device creation error (using *ramDiskDevCreate()*).	The value of the VxWorks *errno* variable.
	XXX_ESYSTFFS	TFFS device creation error (using *tffsDevCreate()*).	The number of the device (2-3) which caused the error.
	XXX_ESYSDOS	DOSFS partition mount failure (using *dosFsDevCreate()*).	The number of the device (1-3) which caused the error.
	XXX_ESYSCHK	A file system failed the a consistency check (using *chkdsk()*).	The number of the device (1-3) which caused the error.
	XXX_EFUNCINI	An application initialization function returned a critical error code.	If the index value (*DIAGS_SEC_BOOT_FAIL_IDX*) is *SBC_ILDDBS*, this is return value of the initialization function that reported a critical error.

3.2 Exceptions

Early in the execution of the secondary boot code, a special temporary exception handler is installed to trap exceptions during the secondary boot process.The LAT secondary boot code exception handler is common for all types of exceptions. Essentially, no exception conditions are acceptable during the execution of the secondary boot code. The exception handler is installed as a VxWorks exception hook (using *excHookAdd()*). This handler is installed early in the BSP

initialization, in the function *sysHwInit()*. The secondary boot code exception handler stores the exception vector number and PPC SSR0, SSR1, DAR, and DSISR register values in the appropriate areas in the boot diagnostics SDRAM region (*DIAGS_EXC_XXX* members). The secondary boot code then sets the software exception flag (*DIAGS_PBF_EXC_RESET*) in the Primary Boot Flags member, clears all other Primary Boot Flags, and performs a warm reboot to the primary boot code at address FFF00104 in SUROM. This returns execution to the SUROM base primary boot code. The memory dump facilities of the primary boot shell may then be used to diagnose the exception information left over by the secondary boot code.

The exception hook additionally provides information in the boot diagnostics application information words (*DIAGS_APP_INFO*). The table below shows the usage of the application information words.

Table 8- Exception Handler Application Information

Application Information Word	Description
0	The contents of the PPC *lr* register at the time of the exception.
1	The contents of the PPC *sp* (*r*1) register at the time of the exception.
2	The address of the VxWorks exception stack frame (ESF).
3	Reserved.
4	Reserved.
5	Reserved.
6	Reserved.
7	Reserved.