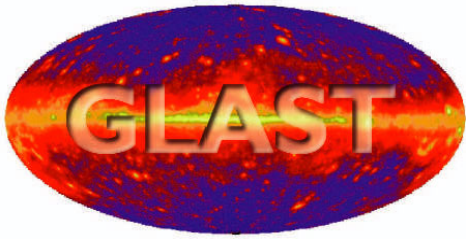

LAT Flight Software

Secondary Boot Code

Number:	LAT-TD-02150-11
Subsystem:	Data Acquisition/Flight Software
Supersedes:	None
Type:	Document Template
Author:	D.L. Wood
Created:	22 January 2002
Updated:	10 November 2008
Printed:	10 November 2008

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 secondary boot script 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
1.1	2 May 2007	Added descriptions for new secondary boot flags bits DIAGS_SBF_MOUNT_MEM and DIAGS_SBF_CACHE_WRITETHOUGH. Updated description of RTOS initialization to include new data cache configuration option. Added description of extended error diagnostic information available from SBC on secondary boot errors.	D.L. Wood
11	6 Nov 2008	Added descriptions of new secondary boot flag bits DIAGS_SBF_RESET_FORCE, DIAGS_SBF_TURBO_DEVICE, and DIAGS_SBF_TURBO.	A.P.Waite

Contents

Figures

Tables

0 Introduction

The LAT secondary boot is an 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

0 Initialize the RAD750 RTOS executable and associated board interface libraries.

1 Load the default set of application code modules from the secondary boot script.

2 Call the application code initialization functions from the secondary boot script.

3 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 `0x00030000`, 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 `0x0000FD80` 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 `0x00C00000`. 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 0x00C00000, the RTOS initialization can proceed assuming good working memory. The RTOS never implicitly allocates memory outside of the system memory pool. Any regions above 0x00C00000 may not have been tested at boot time and might be the responsibility of the applications.

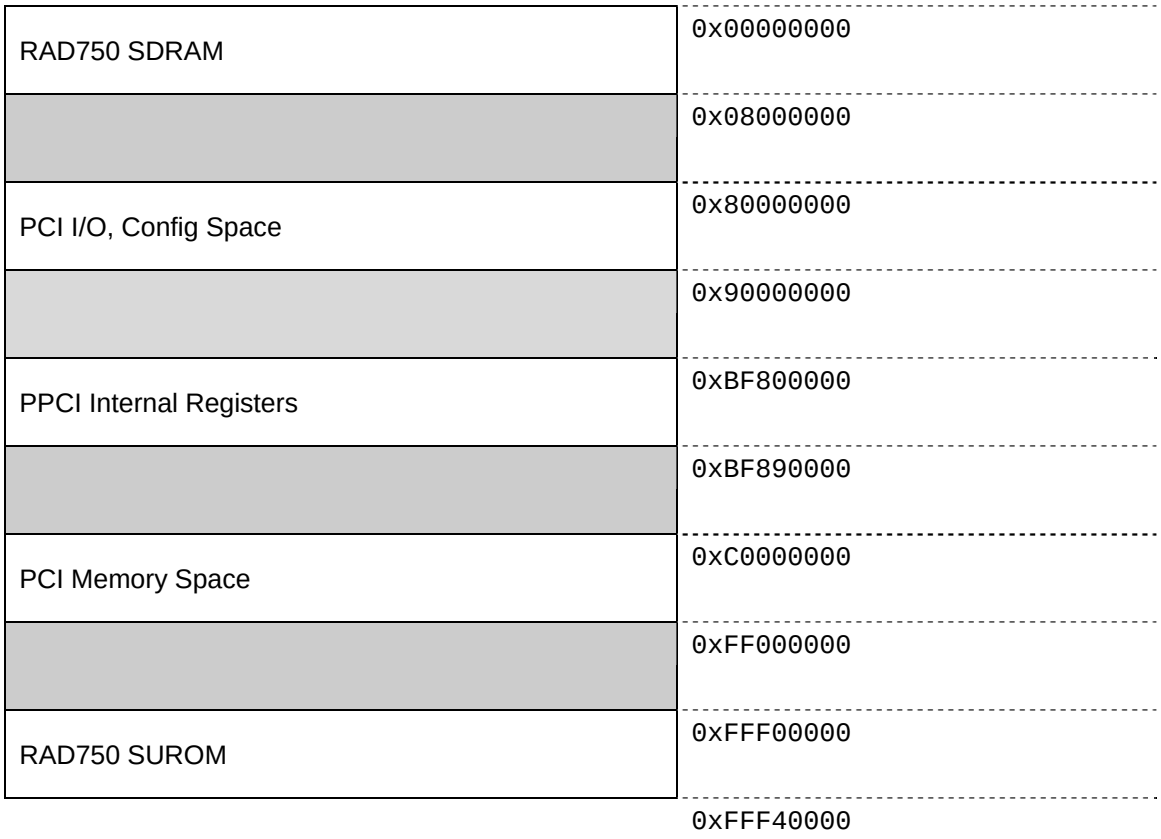

Figure 1 Secondary Boot Code / Application Hardware Memory Map

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.

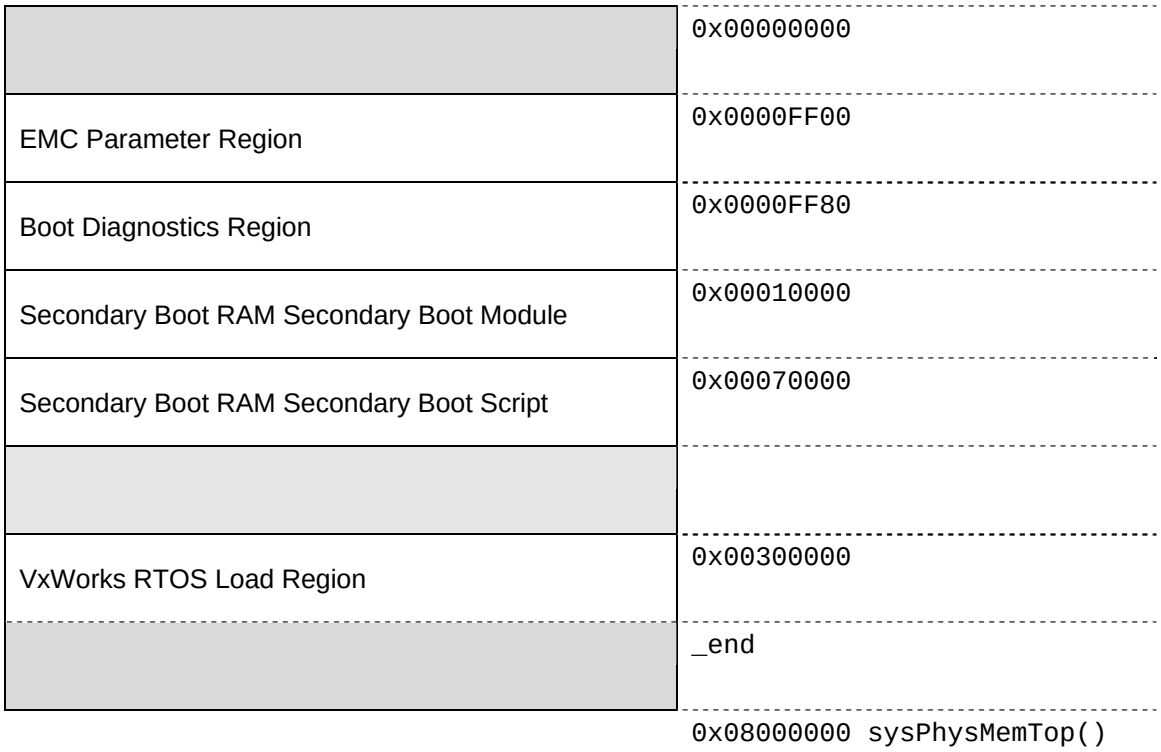

Figure 2 Initial Secondary Boot SDRAM Memory Map

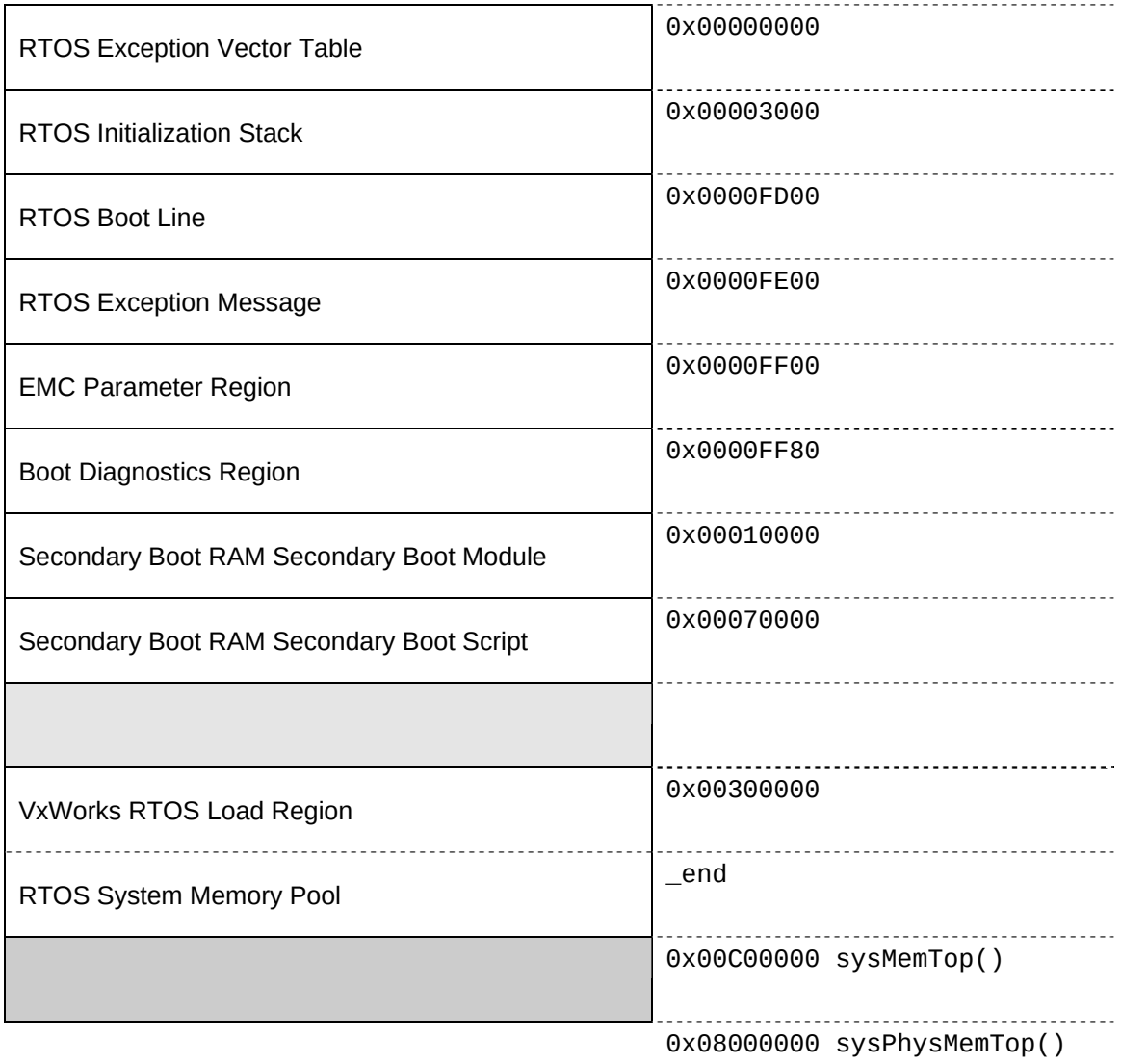

Figure 3 RTOS Initialization SDRAM Memory Map

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 SIB EEPROM.

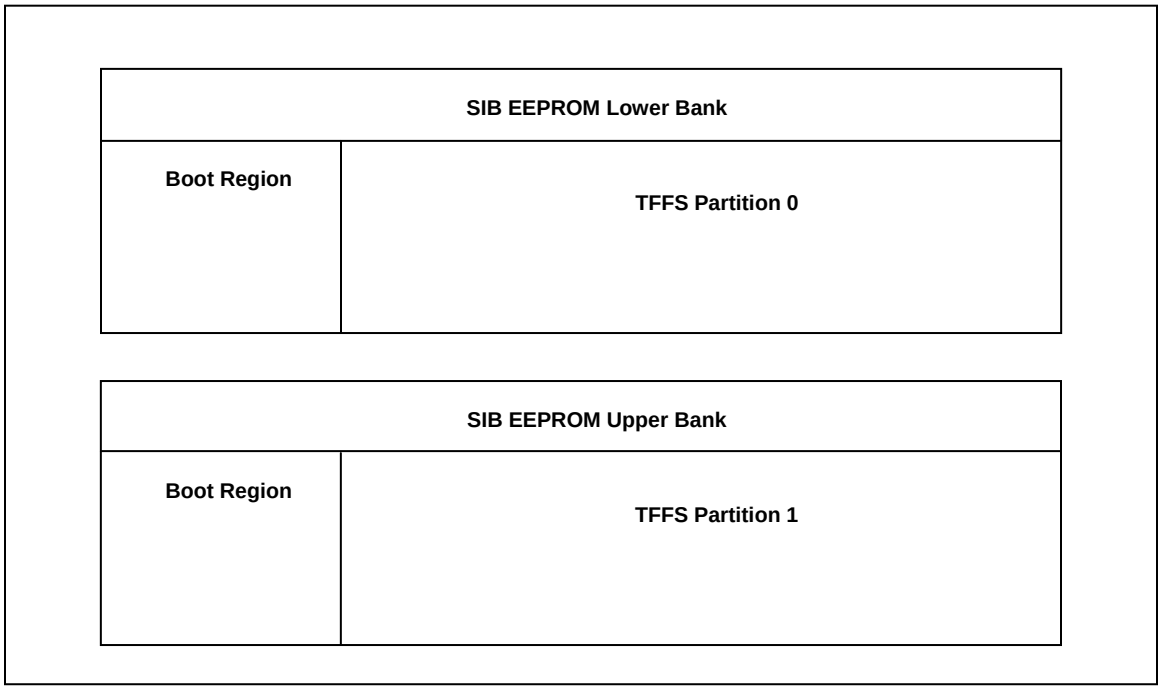

Figure 4 SIB EEPROM Layout

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 volume '0', and the upper EEPROM bank is registered as TFFS volume '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 a secondary boot module and begin execution of the code contained in that module.

1.0.1 PowerPC Processor Initialization

The standard VxWorks initialization sequence handles most of the details of setup and control of the RAD750 board processor. One departure from the standard sequence is how the LAT version of the RTOS handles management of the processor level 1 data cache mode. Normally, the data cache mode, write-through or write-back, is set at the RTOS build time by a configuration macro. For the LAT, the CPU data cache mode is managed by the secondary boot flag DIAGS_SBF_CACHE_WRITETHOUGH bit. Special additional code in the assembly function `sysInit()` interrogates this bit at startup, and configures the CPU MMU register DBAT0 to either place the SDRAM data cache mode to write-though or write-back. Write-back mode is more efficient and is the recommended way to operate, but write-back mode is useful for diagnosing hardware level exceptions.

1.0.2 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.

0 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.

1 Clear the cPCI bus reset signal (RST#) by clearing bit 0 in the PPCI Bus Reset Register (`0xBF870065`). This ensures that the boards on the cPCI backplane and that the cPCI interface of the PPCI are enabled.

2 Disable cPCI Master Abort transactions from generating a machine check by setting bit 7 in the PPCI Configuration Register (`0xBF87005C`). This prevents bus scans from generating a machine check.

3 Set bit 0 in PPCI User Defined A Register (`0xBF870070`) to enable 60X bus error signaling.

4 Enable PPCI OCB errors to generate machine checks by clearing bit 4 in the PPCI Error Status Mask Register (`0xBF810014`). 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.

5 Enable processor machine checks by setting bit 11 in the PPCI User Defined B Register (`0xBF8700A8`).

6 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.

7 Disable PPCI OSR Ordering by setting bit 4 in the PPCI Configuration Register (`0xBF87005C`). This is a workaround for PPCI Errata #12.

8 Set the BAR1 Speculative PCI Read bit and the BAR1 Prefetch Enable bit (bits 24 and 25) in the PPCI Configuration Register (`0xBF87005C`). This improves DMA performance between the RAD750 SDRAM and the cPCI bus.

9 Set PPCI to ignore the bus master's Latency Timer by setting bit 2 in the PPCI Configuration Register (`0xBF87005C`). This is a workaround for PPCI Errata #13.

10 Disable writes to the PPCI internal registers from cPCI bus by setting the BAR2 Write Disable bit (bit 20) in the PPCI Configuration register (`0xBF87005C`). This feature is not needed by the LAT and could prevent bad DMA addresses from corrupting the register settings.

11 Set the PPCI Cacheline Size Register (`0xBF87000C`) to 8.

12 Enable PCI protocol checking, PCI data phase timeouts, and PCI arbitration timeouts in the PPCI Error Checking Register (`0xBF870062`) 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 Register.

13 Map the RAD750 board 128 MB of SDRAM to PCI Memory Address `0x40000000` by setting the PPCI Base Address 1 Register (`0xBF870010`) to `0x40000000`. This is the PCI address which is used by cPCI masters to access the RAD750 main memory.

14 Connect the PPCI to the cPCI bus by setting bits 1, 2, 6, and 8 in the PPCI Control Register (`0xBF870004`). 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.

15 Enable machine check signals to the PPC processor by setting bit 0 in the PPC HID0 register.

16 Disable PPCI interrupts by clearing the PPCI Interrupt Enable Register (`0xBA280004`).

17 Connect the BSP function `ppciInt()` to the PPCI master interrupt level. This function dispatches interrupts signaled in the PPCI Interrupt Collection Register (`0xBA280000`).

18 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 (`0xBA280004`). These interrupts are signaled in the PPCI Misc Interrupt Status Register (`0xBA280094`).

19 Disable all PID interrupts by clearing the PPCI PID Interrupt Enable Register (`0xBA28001C`).

20 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 (`0xBA280004`). These interrupts are signaled in the PPCI PID Input Register (`0xBA280024`).

21 Disable all multiprocessor interrupts by clearing the PPCI MP Signaling Enable Register (`0xBA280088`).

22 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 (`0xBA280004`). These interrupts are signaled in the PPCI MP Signaling Register (`0xBA280080`).

23 Disable all PCI error interrupts by clearing the PPCI PCI Status 2 Mask Register (`0xBF870054`).

24 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 (`0xBA280004`). These interrupts are signaled in the PPCI PCI Status 2 Register (`0xBF870050`).

25 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.

26 Disable all memory controller interrupts by clearing the PPCI Memory Interrupt Enable Register (`0xBF8000C4`).

27 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 (`0xBA280004`). These interrupts are signaled in the PPCI Memory Status Register (`0xBF8000C0`).

28 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.

29 Disable all P60X bus error interrupts by setting bits 0, 1, and 2 in the PPCI P60X Error Mask Register (`0xBF810014`).

30 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 (`0xBA280004`). These interrupts are signaled in the PPCI P60X Error Status Register (`0xBF810010`).

31 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.

32 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 (`0xBA280004`). These interrupts are signaled in the PPCI MP Signaling Register (`0xBA280080`).

33 Set the PPCI RTC divisor to '4' by setting bits [3:2] in the PPCI Clock Control Register (`0xBF8600000`). This sets the RTC clock frequency to 8.25 MHz.

1.1 Loading the Secondary Boot Executable

The secondary boot process requires one or two modules accessible in either EEPROM or a temporary SDRAM buffer. Descriptions of modules are given below.

Module	Description
0	Secondary boot module. An ELF object file which is loaded at the end of the RTOS initialization process. The nominal secondary boot module reads the secondary boot script contained in module 1 and initializes the application code modules
1	Secondary boot script. 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.

Table 1 Secondary Boot Code Modules

The secondary boot process has the option of sourcing these modules from alternate locations as directed by the *DIAGS_SBF_XXX_SOURCE* fields in the secondary boot flags (see Figure 7). Secondary Boot Flags (*DIAGS_SEC_BOOT_FLAGS*) are set by the primary boot shell command that invokes secondary boot. The secondary boot flags values are captured into the boot diagnostics region for later retrieval / inspection. The RTOS file source information is provided for information only, since the RTOS is always copied (and possibly inflated) to SDRAM before execution.

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 fixed offsets in the boot region of each of the SIB EEPROM banks, shown in a typical arrangement below in Figure 5. The EEPROM bank header provides information about the offsets and sizes of each of the file regions in the boot partition.

EEPROM Bank Header	**0x00000**
RTOS NVRAM	0x000100 (nominal)
VxWorks RTOS Executable Image	**0x000200 (nominal)**
Secondary Boot Module (sbm)	**0x0E0000 (nominal)**
Secondary Boot Script (sbs)	**0x0F0000 (nominal)**
Reset Database	0x0F8000 (nominal)
	0x100000 (file system starts)

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.

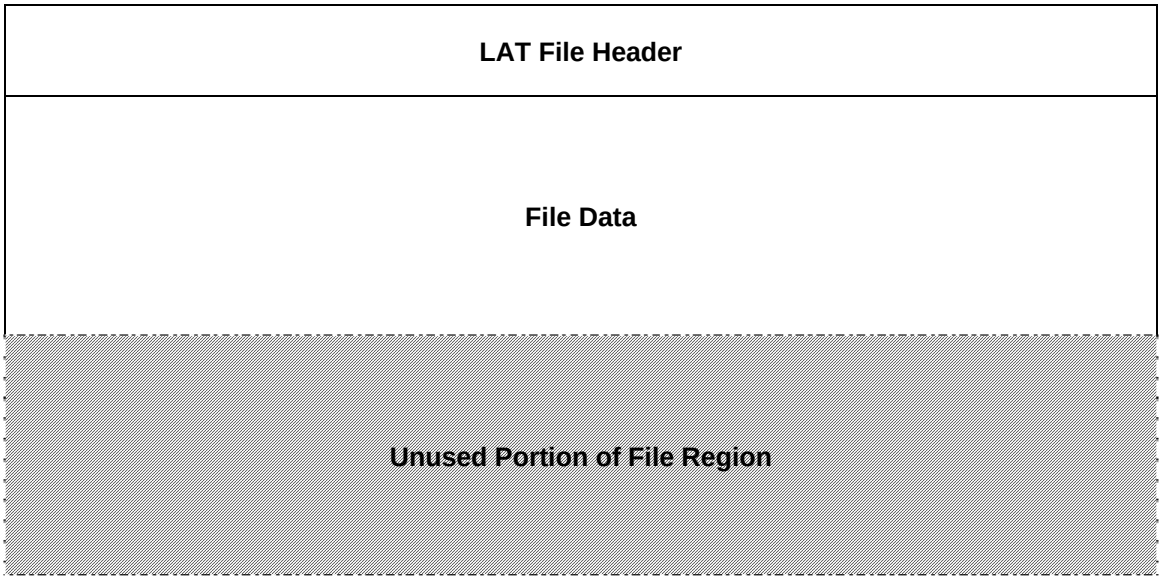

Figure 6 EEPROM Boot Region File Layout

Regardless of whether the secondary boot module 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 (nominal) LAT secondary boot module 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 script 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 `0x00C00000` 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 (bit 31 is LSB):

Bits	Mnemonic	Description	Notes
0 – 1	DIAGS_SBF_RTOS_SOURCE	RTOS image source flag indicating the location from which the RTOS executable was taken when starting the secondary boot process: 0 = RAM temporary area 1 = SIB EEPROM upper bank boot partition 2 = SIB EEPROM lower bank boot partition.	
2 – 3	DIAGS_SBF_MODO_SOURCE	Flags indicating the location from which the secondary boot process should source the secondary boot module: 0 = RAM temporary area 1 = SIB EEPROM upper bank boot partition 2 = SIB EEPROM lower bank boot partition	
4 – 5	DIAGS_SBF_MOD1_SOURCE	Flags indicating the location from which the secondary boot process should source the secondary boot script: 0 = RAM temporary area 1 = SIB EEPROM upper bank boot partition 2 = EEPROM lower bank boot partition	
6 – 15		Reserved.	
16	DIAGS_SBF_MOUNT_EE0	Mount the lower EEPROM bank as a TFFS device `/ee0`.	
17	DIAGS_SBF_MOUNT_EE1	Mount the lower EEPROM bank as a TFFS device `/ee1`.	
18	DIAGS_SBF_MOUNT_MEM	Mount secondary boot module RAM area as module source.	0
19	DIAGS_SBF_CACHE_WRITETHROUGH	Force CPU to cache write-through mode.	1
20 - 27		Reserved.	
28	DIAGS_SBF_RESET_FORCE	Force CPU reset even if tests say it may not be viable.	2
29 - 30	BOOT_DIAGS_SBF_TURBO_DEVICE	Device on which to store a reset database. 0 = Do not store a reset database 1 = SIB EEPROM upper bank boot partition 2 = SIB EEPROM lower bank boot partition	2
31	BOOT_DIAGS_SBF_TURBO	Perform a CPU “turbo reset” (using primary boot’s “fast boot” capability).	2

Figure 7 Secondary Boot Flags

`0` A developer only trick. It is possible to construct some temporary development/testing code, format it into a pseudo-secondary-boot-module, load it into the RAM area, and then have the full autoboot method find and load it. Sometimes referred to as “code smuggling”. This is not for the faint of heart. For more details, see the FMX manual.

`1` Another flag that lies firmly in the developer domain. This flag forces the CPU to operate in cache write-through mode. This can be essential in tracking down various obscure bugs. It should never be used for normal running due to the high performance penalty (a factor of 3-5 in processor speed).

2 A group of flags dedicated to supporting “turbo reset”. There is no credible scenario in which these flags would be used to move from primary boot’s boot shell mode to applications mode. These flags are set from applications mode, and use the “fast boot” bit in the primary boot flags in order to go from applications mode, through primary boot mode and back to applications mode, without ever entering primary boot’s boot shell mode.

The secondary boot flags word must be 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 module (see Section 1.1). Bits 3 – 4 determine where the secondary boot executable will find the secondary boot script (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). Bits 18 and 19 activate special debugging modes of the RTOS and SBC.

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.

2.3 Application Module File Formats

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

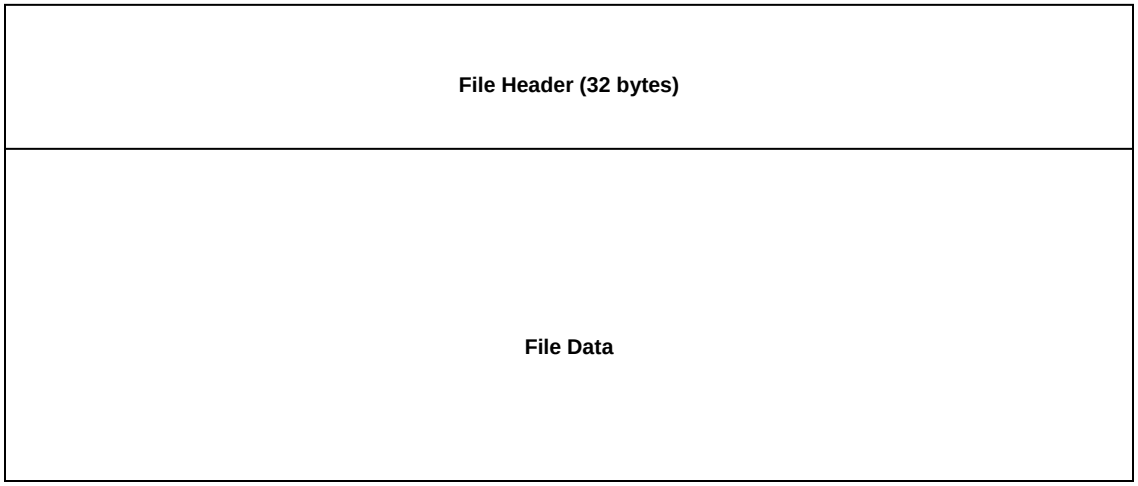

Figure 8 Uncompressed File Format

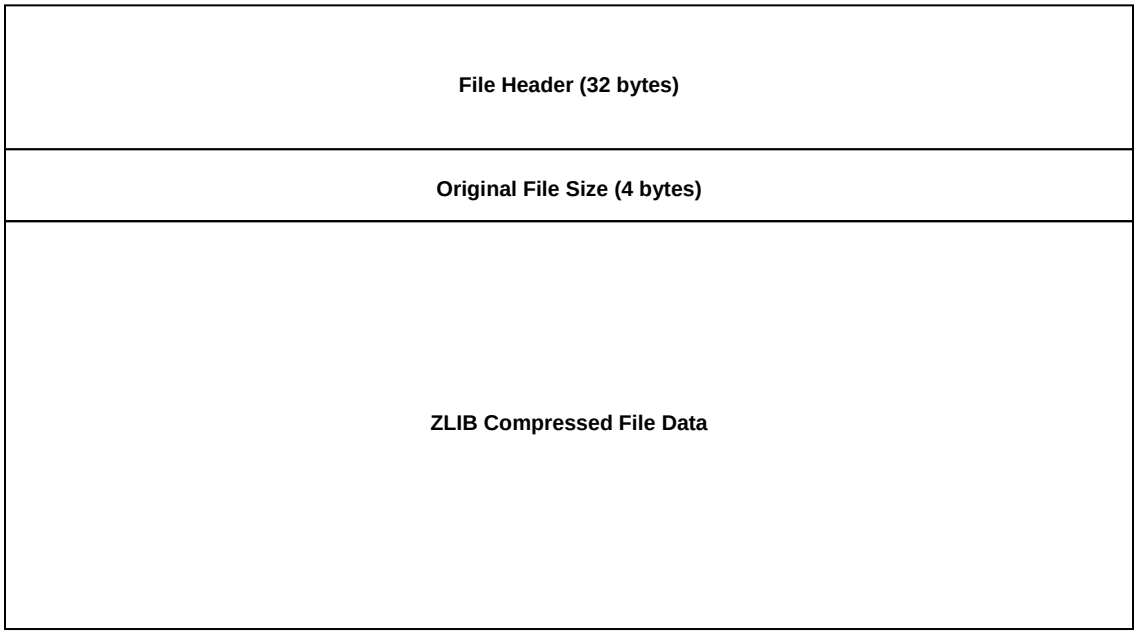

Figure 9 Compressed File Format

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 secondary boot script, 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 Secondary Boot Script

The secondary boot script 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 secondary boot script is shown in Figure 10.

Script Header
Number of Module File ID's
Module File ID's
Number of Initialization Functions
Initialization Function Symbol Names and Parameters

Figure 10 Secondary Boot Script Format

Essentially, the script 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 function 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 script is always parsed beginning with the module list.

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

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

'S'	'B'	'C'	Version Number

Figure 11 Secondary Boot Script Header Word

The version number of the script 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 script. 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 script 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 script.

Right after an application object module is loaded, the secondary boot code records 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:

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_KEY*	File key (taken from file header).
3	*CAB_K_SPARE_1*	Reserved.

Table 2 Secondary Boot Code CMX User Data Values

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 script. The detail of each initialization function entry is shown in the figure below.

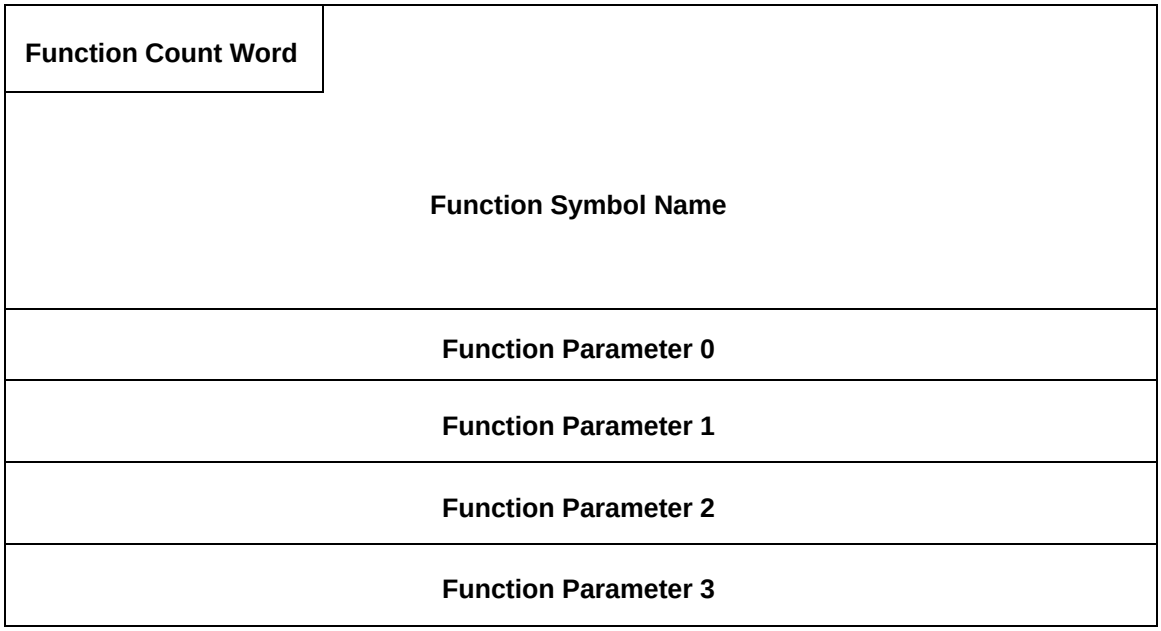

Figure 12 One Application Initialization Function Script Entry

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.

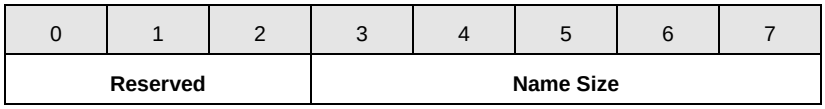

Figure 13 Application Initialization Function Count Word

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 script 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 entry. When the function execution completes, the secondary boot code examines the return value from the function. The secondary boot code expects that 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 `0x0000FF80` 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.

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.

Offset (bytes)	Mnemonic	Description	Data Source
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.

Table 3 SDRAM Boot Diagnostics Area

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).

MSG Code (DIAGS_SEC_BOOT_FAIL_IDX)		Description
VXW_IVXBNKH	0x3b09af8e	Reading the EEPROM boot bank header.
VXW_IVXEXH	0x3b0265de	Reading SBC module file header.
VXW_IVXEXF	0x3b0265fe	Reading SBC module file data.
VXW_IVXEXI	0x3b0265d2	Inflating SBC module file.
VXW_IVXEXL	0x3b0265a2	Loading SBC module file.

Table 4 Secondary Boot RTOS Index Codes

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

MSG Code (DIAGS_SEC_BOOT_FAIL_IDX)		Description
SBC_IMNTRAM	0x398858ba	Mounting RAM disk partition.
SBC_IMNTMEM	0x398860ea	Mounting memory driver partition.
SBC_IMNTEE	0x398216da	Mounting TFFS EEPROM partitions.
SBC_ILDBNKH	0x39875b8e	Reading the EEPROM boot bank header.
SBC_ILDDBH	0x3981d3be	Reading SBC module 1 file header.
SBC_ILDDBF	0x3981d3de	Reading SBC module 1 file data.

MSG Code (DIAGS_SEC_BOOT_FAIL_IDX)		Description
SBC_ILDDBI	0x3981d3b2	Inflating SBC module 1 file.
SBC_ILDDBM	0x3981d3da	Loading application module files.
SBC_ILDDBS	0x3981d35a	Calling application initialization functions.

Table 5 Secondary Boot Application Index Codes

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.

MSG Code (DIAGS_SEC_BOOT_FAIL_ERR)		Description	Data Word (DIAGS_SEC_BOOT_FAIL_DATA)
VXW_SUCCESS	0x3b0333a8	Success.	None.
VXW_EMEMALOC	0x3b17fc13	Memory allocation error (from system memory partition using `malloc()`)	The number of bytes requested in the allocation.
VXW_EEEREAD	0x3b049ddb	EEPROM read error (using `tffsRawIo()`)	The address the read started at.
VXW_EFILHDR	0x3b062467	SBC module file header validation error (using `FILE_hdrVerify()`).	The return value from `FILE_hdrVerify()`.
VXW_EFILCKSM	0x3b189c2b	SBC module file checksum validation error.	File checksum computed value.
VXW_EFILOPEN	0x3b189417	SBC module file open error (using `open()`).	The file ID storage value of the file being opened.
VXW_EMEMDEV	0x3b060213	Memory device creation failure (using `memDevCreate()`).	The value of the VxWorks `errno` variable.
VXW_EFILLOAD	0x3b18839b	SBC module file module load error (using `loadModule()`).	The file ID of the module file.
VXW_EFILSYM	0x3b061f9b	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 script function list.
VXW_EZLIBINI	0x3b1bd433	ZLIB initialization error (using `inflateInit()`).	The return value from `inflateInit()`.

MSG Code (DIAGS_SEC_BOOT_FAIL_ERR)		Description	Data Word (DIAGS_SEC_BOOT_FAIL_DATA)
VXW_EZLIBINF	0x3b1bd45f	ZLIB inflation error (using `inflate()`).	The return value from `inflate()`.
VXW_ESYSRAM	0x3b0447bb	RAM disk device creation error (using `ramDiskDevCreate()`).	The value of the VxWorks `errno` variable.
VXW_ESYSTFFS	0x3b1125fb	TFFS device creation error (using `tffsDevCreate()`).	The number of the device (2-3) which caused the error.
VXW_ESYSDOS	0x3b044e1b	DOSFS partition mount failure (using `dosFsDevCreate()`).	The number of the device (1-3) which caused the error.

Table 6 Secondary Boot Error Codes (VXW)

MSG Code (DIAGS_SEC_BOOT_FAIL_ERR)		Description	Data Word (DIAGS_SEC_BOOT_FAIL_DATA)
SBC_SUCCESS	0x398333a8	Success.	None.
SBC_EMEMALOC	0x3997fc13	Memory allocation error (from system memory partition using `malloc()`)	The number of bytes requested in the allocation.
SBC_EEEREAD	0x39849ddb	EEPROM read error (using `tffsRawIo()`)	The address the read started at.
SBC_EFILHDR	0x39862467	SBC module file header validation error (using `FILE_hdrVerify()`).	The return value from `FILE_hdrVerify()`.
SBC_EFILFRMT	0x3998956f	SBC module file data format or version error.	The value of the first 32-bit word of the file header.
SBC_EFILCKSM	0x39989c2b	SBC module file checksum validation error.	File checksum computed value.
SBC_EFILSIZE	0x39987b5b	An application module file is too large to load.	The size in bytes of the module object file.
SBC_EFILOPEN	0x39989417	SBC module file open error (using `open()`).	The file ID storage value of the file being opened.
SBC_EFILREAD	0x399875db	Module file read error (using `read()`).	The value of the VxWorks `errno` variable.
SBC_EFILWRIT	0x3998a3cf	Module file write error (using `write()`).	The value of the VxWorks `errno` variable.
SBC_EMEMDEV	0x39860213	Memory device creation failure (using `memDevCreate()`).	The value of the VxWorks `errno` variable.
SBC_EFILLOAD	0x3998839b	SBC module file module load error (using `loadModule()`).	The file ID of the module file.
SBC_EFILSYM	0x39861f9b	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 script function list.
SBC_EZLIBINI	0x399bd433	ZLIB initialization error (using `inflateInit()`).	The return value from `inflateInit()`.

MSG Code (DIAGS_SEC_BOOT_FAIL_ERR)		Description	Data Word (DIAGS_SEC_BOOT_FAIL_DATA)
SBC_EZLIBINF	0x399bd45f	ZLIB inflation error (using `inflate()`).	The return value from `inflate()`.
SBC_ESYSRAM	0x398447bb	RAM disk device creation error (using `ramDiskDevCreate()`).	The value of the VxWorks `errno` variable.
SBC_ESYSTFFS	0x399125fb	TFFS device creation error (using `tffsDevCreate()`).	The number of the device (2-3) which caused the error.
SBC_ESYSDOS	0x39844e1b	DOSFS partition mount failure (using `dosFsDevCreate()`).	The number of the device (1-3) which caused the error.
SBC_ESYSCHK	0x3984506b	A file system failed the a consistency check (using `chkdsk()`).	The number of the device (1-3) which caused the error.
SBC_ESYSMDRV	0x39914143	Memory device directory create error (using `memDevCreateDir()`)	The value of the VxWorks `errno` variable.
SBC_EMMAGIC	0x39864383	Memory device upload image error.	The failing magic word value.
SBC_EFUNCINI	0x39992b33	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.

Table 7 Secondary Boot Error Codes (SBC)

In addition to the data word value placed in the DIAGS_SEC_BOOT_FAIL_DATA slot of the diagnostics area, certain errors can leave behind additional information in the application information words (*DIAGS_APP_INFO*). The table below shows the usage of the application information words.

Application Information Word	Description
0	The file ID value of the target file which generated the error. This word is set for errors encountered in the SBC_ILDDBM phase of SBC operation. The file ID will refer to an unloadable application code module.
1	Reserved.
2	Reserved.
3	Reserved.
4	Reserved.
5	Reserved.
6	Reserved.
7	Reserved.

Table 8 Secondary Boot Panic Error Application Information

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.

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` (`r1`) 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.

Table 9 Exception Handler Application Information