Table 39. Format of Block List ***(Continued)***

Length of memory area 1
Device for area 1
Block Number for area 1
Address of memory area 2
Length of memory area 2
Device for area 2
Block Number for area 2
...
Address of memory area n
Length of memory area n
Device for area n
Block number for area n

7–119. The *hibernate block_list* must be contained in system memory below 4 GB.

7–120. The *Device* fields of the *hibernate block_list* must be real pointers to System Memory below 4 GB that point to Open Firmware Path Names.

7–121. A device specified in the *hibernate block_list* must correspond to an device in the Open Firmware Device Tree that has *write* and *read* methods, and has a device type of *block*.

7–122. The memory areas defined by the *hibernate block_list* must only include System Memory outside that reserved for firmware (both the RTAS data area and Open Firmware's memory defined by *real-base* and *real-size*).

7–123. The memory areas defined by the *hibernate block_list* must only include System Memory below 4 GB.

7–124. Execution of the *hibernate* call must cause the System Memory areas described in the *block_list* to be saved on disk on the device(s) specified starting at the specified block numbers.

Personal Use Copy - Not for Reproduction

7–125. After system memory is saved, *hibernate* must place the system into its lowest power state.

7–126. In an SMP system, all other processors must be in the *stopped* state before invoking *hibernate*. This is the responsibility of the operating system.

7–127. Prior to making the *hibernate* call, the operating system must put all I/O devices into a quiescent state: they must not be transferring data to or from memory, nor be able to cause an interrupt to any processor.

7–128. Open Firmware must deterministically initialize a platform. The Open Firmware device tree and device addresses assigned on a given platform must be the same on successive reboots and hibernate wakeups if the platform hardware configuration has not changed.

Software Implementation Note: The firmware is free to move and otherwise modify System Memory during this call.

7.3.8.2 Hibernate Restoration (Wakeup)

From the standpoint of the platform firmware, wakeup is no different from any other system initialization process. It is the responsibility of the operating system to:

1. Recognize that this is a wakeup
2. Find the saved hibernation image on disk
3. Restore the hibernation image to memory
4. Transfer control back to the proper point for continued execution.

Software Implementation Note: Wakeup from hibernate is entirely an operating system implemented function, initiated by booting the system. Upon booting, OS boot or initialization code runs and notices a hibernation image. This early code must be careful to instantiate RTAS in an area of memory which will not be destroyed when the OS places the hibernation image into memory.

Software Implementation Note: In a Multiboot situation, it is possible for different operating systems resident on a system to be hibernating at the same time. This is a quick way to switch between operating system environments. In this scenario, the user would indicate to system firmware a different OS to boot/wakeup next, hibernate the system, and do a reboot operation to wakeup that OS image.

Personal Use Copy - Not for Reproduction

Software Implementation Note: The OS timebase must be reestablished using the Real-Time Clock upon wakeup, since an unknown amount of time will have passed.

7.3.9 Reboot

During execution, it may become necessary to shut down processing and re-boot the system in a new mode. For example, a different operating system may need to be loaded, or the same operating system may need to be re-booted with different settings of System Environment Variables.

7.3.9.1 System-reboot

Requirements:

7–129. RTAS must implement a *system-reboot* call which resets all processors and all attached devices. After reset, the system must be booted with the current settings of the System Environment Variables (refer to Section 8.4.3, “System (0x70),” on page 145 for more information).

7–130. The RTAS *system-reboot* call must be implemented using the argument call buffer defined by Table 40 on page 135.

Table 40. *system-reboot* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *system-reboot*
	Number Inputs	0
	Number Outputs	1
Out	Status	On successful operation, does not return -1: Hardware error

Hardware Implementation Note: The platform must be able to perform a system reset and reboot. On a multiprocessor system, this should be a hard reset to the processors.

Personal Use Copy - Not for Reproduction

7.3.10 Caches

The following interfaces allow simple control of internal and external caches, primarily for turning these caches on or off for power management or for management of a multiprocessor system.

When flushing and disabling caches, the operating system must take care to flush caches in order (see Section 4.2.4, “Cache Memory,” on page 65) and take care to synchronize with access by other elements of the system. These operations shall only be called by a processor connected to the cache. The current processor shall be the only active processor connected to the cache.

7.3.10.1 Cache-control

Requirements:

7–131. For the Symmetric Multiprocessor or Power Management option: RTAS must implement a *cache-control* call to place the cache into a new state, using the argument call buffer defined by Table 41 on page 136.

Table 41. *cache-control* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *cache-control*
	Number Inputs	2
	Number Outputs	1
	Cache_id	phandle of the cache (for L1 caches, the phandle of the processor)
	How	New state of the cache
Out	Status	0: Success -1: Hardware error -3: Bad state for this cache

7–132. For the Symmetric Multiprocessor or Power Management option: When entering the new state indicated by the *How* parameter, the actions specified in Table 42 on page 137 must be performed on the caches.

Personal Use Copy - Not for Reproduction

Table 42. *Cache-control* states

State Number	State Name	Action
1	Copy-back	The cache must be enabled in a full powered copy-back mode.
2	Write-Through	If necessary, the cache must be flushed. The cache must then be enabled in a full powered Write-Through mode.
3	Low-power	If necessary, the cache must be flushed and invalidated. The cache must then be placed in a low power but still functioning mode.
4	Disabled	The cache must be flushed, and then put in the lowest powered disabled mode.
5	Flush	The cache contents are flushed, and the cache remains in its current state.
9000-9999	Reserved	Reserved for platform vendor specific actions.

7–133. For the Symmetric Multiprocessor or Power Management option: The RTAS *cache-control* operations performed on the specified cache must appear to be a single atomic operation as seen from the operating system. A return status of success implies “committed,” that the operation is complete, and that any dirty data is safely out of the cache. These operations must not violate the cache coherency of the caches.

7.3.11 SMP Support

In a Symmetric Multiprocessor (SMP) system, the operating system needs the ability to synchronize the clocks on all the processors. To do this, it must have the ability to stop all clocks at the same time. For example, the TBEN signal provided on the PowerPC 604 microprocessor can be used to implement this clock control function.

7.3.11.1 freeze-time-base

Requirements:

7–134. For the Symmetric Multiprocessor option: RTAS must implement a *freeze-time-base* call which *freezes*, or keeps from changing, the time

Personal Use Copy - Not for Reproduction

base register on all processors. This call must be implemented using the argument call buffer defined by Table 43 on page 138.

Table 43. *freeze-time-base* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *freeze-time-base*
	Number Inputs	0
	Number Outputs	1
Out	Status	0: Success -1: Hardware Error

7–135. For the Symmetric Multiprocessor option: The *freeze-time-base* operation must simultaneously affect every processor of an SMP system.

7.3.11.2 thaw-time-base

Requirements:

7–136. For the Symmetric Multiprocessor option: RTAS must implement a *thaw-time-base* call which *thaws*, or permits the change of, the time base register on all processors. This call must be implemented using the argument call buffer defined by Table 44 on page 138.

Table 44. *thaw-time-base* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *thaw-time-base*
	Number Inputs	0
	Number Outputs	1
Out	Status	0: Success -1: Hardware Error

Personal Use Copy - Not for Reproduction

7–137. For the Symmetric Multiprocessor option: The *thaw-time-base* operation must simultaneously affect every processor of an SMP system.

7.3.11.3 stop-self

The stop-self primitive causes a processor to stop processing OS or user code, and to enter a state in which it is only responsive to the *start-cpu* RTAS primitive. This is referred to as the RTAS *stopped* state.

Requirements:

7–138. For the Symmetric Multiprocessor option: RTAS must implement a *stop-self* call which places the calling processor in the RTAS *stopped* state. This call must be implemented using the argument call buffer defined by Table 45 on page 139.

7–139. For the Symmetric Multiprocessor option: RTAS must insure that a processor in the RTAS *stopped* state will not check stop or otherwise fail if a machine check or soft reset exception occurs. Processors in this state will receive the exception, but must perform a null action and remain in the RTAS *stopped* state.

Table 45. *stop-self* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *stop-self*
	Number Inputs	0
	Number Outputs	1
Out	Status	If successful, this call does not return -1: Hardware Error

Software Implementation Note: If this call succeeds, it will not return. The CPU will wait for some other processor to issue a *start-cpu* targeted to this processor.

Personal Use Copy - Not for Reproduction

7.3.11.4 start-cpu

The *start-cpu* primitive is used to cause a processor which is currently in the RTAS *stopped* state to start processing at an indicated location.

Requirements:

7–140. For the Symmetric Multiprocessor option: RTAS must implement a *start-cpu* call which removes the processor specified by the *CPU_id* parameter from the RTAS *stopped* state. This call must be implemented using the argument call buffer defined by Table 46 on page 140.

7–141. For the Symmetric Multiprocessor option: The processor specified by the *CPU_id* parameter must be in the RTAS *stopped* state entered because of a prior call by that processor to the *stop-self* primitive.

7–142. For the Symmetric Multiprocessor option: When a processor exits the RTAS *stopped* state, it must begin execution in real mode, at the real location indicated by the *Start_location* parameter, with register R3 set to the value of parameter *Register_R3_contents,* in the endian mode of the processor executing the *start-cpu* primitive. All other register contents are indeterminate.

Table 46. *start-cpu* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *start-cpu*
	Number Inputs	3
	Number Outputs	1
	Cpu_id	Token identifying the processor to be started, obtained from the *reg* value for the CPU in the Open Firmware device tree
	Start_location	Real address at which the designated CPU will begin execution
	Register_R3_contents	Value which will be loaded into Register R3 before beginning execution at Start_location
Out	Status	0: Success -1: Hardware Error

Personal Use Copy - Not for Reproduction

Non-VolatileMemory

8

This chapter describes the requirements relating to the Common Hardware Reference Platform Non-Volatile Memory. Non-Volatile Memory is the repository for system information that must be persistent across reboots, power management activities, and power cycles.

8.1 System Requirements

Requirements:

8–1. Platforms must implement at least 8 KB of Non-Volatile Memory. This is sufficient for a system with a single operating system installed. Allow 1 KB for each additional operating system installed.

8–2. Non-Volatile Memory must maintain its contents in the absence of system power.

8–3. Firmware must reinitialize NVRAM to a bootable state if NVRAM data corruption is detected.

8–4. Operating systems must reinitialize their own NVRAM partitions if NVRAM data corruption is detected. Operating systems may create free space from the first corrupted partition header to the end of NVRAM and utilize this area to initialize their partitions.

Hardware Implementation Note: Non-volatile memory is normally implemented with battery-powered RAM and is generally called NVRAM. This terminology will be used throughout the remainder of this chapter, although it should be understood that this is not the only possible implementation.

Software Implementation Note: Refer to Section 7.3.2, "NVRAM Access Functions," on page 104 for information on accessing NVRAM.

8.2 Structure

NVRAM is formatted as a set of partitions that adhere to the structure in Table 47 on page 143. Partitions are prefixed with a *header* containing **`signature`**, **`checksum`**, **`length`**, and **`name`** fields. The structure of the **`data`** field is defined by the partition creator/owner (designated by "**`signature`**" and "**`name`**").

Requirements:

8–5. NVRAM partitions must be structured as shown in Table 47 on page 143.

8–6. All NVRAM space must be accounted for by partitions.

8.3 Signatures

The **`signature`** field is used as the first level of partition identification. Table 48 on page 144 lists all the currently defined signature types and their ownership classes. The ownership class determines the permission of a particular system software component to create and/or modify partitions and/or partition contents. All partitions may be read by any system software component, but the ownership class has exclusive write permission. Global ownership gives read/write permission to all system software components. These restrictions are made to minimize the possibility of corruption of NVRAM during update activities.

Hardware and Software Implementation Note: It is recommended that partitions be ordered on the signature field with the lowest value signature partition at the lowest NVRAM address (with the exception of signature = 0x7F, free space). This will minimize the effect of NVRAM data corruption on system operation.

Personal Use Copy - Not for Reproduction

Table 47. NVRAM Structure

Field Name	Size	Description
signature	1 byte	The **signature** field is used to identify the partition type and provide some level of checking for overall NVRAM contamination. Signature assignments are given in Table 48 on page 144.

checksum	1 byte	The **checksum** field is included to provide a check on the validity of the header. The checksum covers the **signature**, **length**, and **name** fields and is calculated (on a byte by byte or equivalent basis) by: add, and add 1 back to the sum if a carry resulted as demonstrated with the following program listing.
		
		`unsigned char sumcheck(bp,nbytes)`
		`unsigned char *bp /* buffer pointer */`
		`unsigned int nbytes; /* number of bytes to sum */`
		`{`
		`unsigned char b_data; /* byte data */`
		`unsigned char i_sum; /* intermediate sum */`
		`unsigned char c_sum; /* current sum */`
		`for (c_sum = 0; nbytes; nbytes--)`
		`{`
		`b_data = *bp++; /* read byte from buffer */`
		`i_sum = c_sum + b_data; /* add to current sum */`
		`if(i_sum < c_sum) /* did a carry out result? */`
		`i_sum += 1; /* if so, add 1 */`
		`c_sum = i_sum; /* copy to current sum */`
		`}`
		`return (c_sum);`
		`}`
		
		This checksum algorithm guarantees 0 to be an impossible calculated value.

length	2 bytes Big-Endian format	The **length** field designates the *total* length of the partition, in 16-byte blocks, beginning with the signature and ending with the last byte of the data area. This allows a maximum partition length of 16 x 2**16 = 1 MB. Software Implementation Note: The **length** fields must always provide valid offsets to the next header since an invalid length effectively causes the loss of access to every partition beyond it.
name	12 bytes	The **name** field is a null-terminated (if less than 12 bytes) string used to identify a particular partition within a signature group. There is no general uniqueness requirement, but each OS partition name must be prefixed with a 3 character Organizationally Unique Identifier (OUI) assigned by the IEEE Registration Authority Committee to the OS vendor, *followed by a comma*. If an OS vendor does not have an assigned OUI, a prefix of "," (comma) will be used to signify "other." Before assigning a new name to a partition, software should scan the existing partitions and ensure that an unwanted name conflict is not created.
data	**Length** minus 16 bytes	The structure of the **data** area is controlled by the creator/owner of the partition.

Personal Use Copy - Not for Reproduction

Table 48. NVRAM Signatures

Signature	Signature Type	Ownership Class	# Required	Description
0x01 - 0x0F	Support Processor	Any support processor	0 to n	Reserved for support processor use.
0x50	Open Firmware	Firmware	1	Open Firmware Configuration Variables.
0x51	Reserved	Firmware	n/a	Reserved
0x52	Hardware	Firmware	1 to n	These partitions are used to store administrative machine data such as serial numbers, ec levels, etc. This is often referred to as Vital Product Data (VPD).
0x53 - 0x5F	Firmware	Firmware	0 to n	General firmware usage.
0x70	System	Global	1	The System partition is used to store environment variables that must be accessed by firmware and operating systems.
0x71	Configuration	Global	0 to n	Configuration partitions are used to store configuration data generated by the system firmware or the operating system that is required at boot time to properly configure the system.
0x72	Error Log	Global	0 to n	This partition is used to store the error logs built as a result of various RTAS calls.
0x73	Multi-boot	Global	0 or 1	This partition will contain an entry for each OS installed on the system which will participate in a multiple boot process.
0x74 - 0x7E	Global	Global	0 to n	General global usage.
0x7F	Free Space	Global	0 to n	This signature is used to mark free space in the NVRAM array. The **name** field of all signature 0x7F partitions must be set to 0x7...77.
0xA0 - 0xAF	OS	Any OS	0 to n	General operating system usage.
0x00, 0x10 - 0x4F, 0x60 - 0x6F, 0x80 - 0x9F, 0xB0 - 0xFF				Reserved

8.4 Architected Partitions

8.4.1 Open Firmware (0x50)

This partition is required for storing Open Firmware Configuration Variables.

Requirements:

8–7. The system NVRAM must include a 0x50 partition with the name *of-config* to store Open Firmware Configuration Variables.

Personal Use Copy - Not for Reproduction

8.4.2 Hardware (0x52)

This partition type is used to store Vital Product Data (VPD).

Requirements:

8–8. VPD must be stored in the 0x52 partition using the format defined in the *PCI Local Bus Specification,* Revision 2.1, section 6.4 [14].

8.4.3 System (0x70)

The System partition is required for storing environment variables that are common to all platforms and operating systems.

Requirements:

8–9. Every system NVRAM must contain a System partition with the partition name = *common.*

8–10. Data in the *common* partition must be stored as null-terminated strings of the form: *name=<string>* and be terminated with a string of at least two null characters.

8–11. All *names* used in the *common* partition must be unique.

8–12. Device and file specifications used in the *common* partition must follow IEEE Std 1275 nomenclature conventions.

Software Implementation Note: Open Firmware will look for the *boot-script* variable in the *common* System partition; *boot-script=<string>* is a list of Open Firmware commands to be executed during the boot startup sequence. See *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10] for information on the use of this variable.

8.4.4 Configuration (0x71)

8.4.4.1 ISA Configuration Data

ISA configuration data (integrated devices and plug-in cards) is passed to the OS in the OF device tree. In order to enable OS reconfiguration of ISA device

Personal Use Copy - Not for Reproduction

resources, the ISA configuration database will also be entered into NVRAM. This gives the operating system the ability to observe and modify the configuration which will be used by the firmware on the next system boot.

Requirements:

8–13. System NVRAM must include a 0x71 partition with the name *isa-config* to store configuration data for all ISA devices.

8–14. The *isa-config* data area must use the resource format given in the ISA/EISA/ISA-PnP binding to IEEE 1275, *IEEE Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices* [9].

8–15. During boot, firmware must use the ISA configuration data stored in the *isa-config* NVRAM partition to generate the ISA portion of the OF device tree.

Architecture Note: This NVRAM partition is used to store configuration data for both Plug and Play (PnP) and non-PnP ISA adapters. For non-PnP devices, Vendor IDs will be assigned and documented in *PowerPC Microprocessor Hardware Reference Platform: I/O Device Reference* [20].

8.4.4.2 Power Management Configuration Data

Power Management configuration data provides power state and transition data necessary to power manage add-in devices.

Requirements:

8–16. **For the Power Management option:** NVRAM must include a 0x71 partition with the name *pm-config* to store power management configuration data.

8–17. **For the Power Management option:** The *pm-config* data area must use the format given in the *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10].

Personal Use Copy - Not for Reproduction

8.4.5 Error Log (0x72)

NVRAM error logs are optional.

It is recommended that operating systems record RTAS-returned error logs in NVRAM to provide error information in the event of a checkstop/reboot. It is also recommended that an error log be provided for errors discovered during IPL POST.

Requirements:

8–18. If an operating system implements an NVRAM error log partition for RTAS,

- **a.** the partition name must be *rtas-err-log*.
- **b.** the error log format must be as given in Table 54 on page 176.
- **c.** error log entries must be filled by the operating system in a manner that will make the most recent log visible.

8–19. If firmware implements an NVRAM error log partition for POST,

- **a.** the partition name must be *post-err-log*.
- **b.** the error log format must be as given in Table 54 on page 176.
- **c.** error log entries must be filled by the firmware in a manner that will make the most recent log visible. If multiple entries are provided, they must be filled in a manner that will make the first and last error occurrences visible to the OS.

8.4.6 Multi-Boot (0x73)

The multi-boot partition is only required if multi-boot is implemented on a platform. The partition will contain entries for each operating system installed on the platform that will participate in the multi-boot process. A configuration variable, *multi-boot?*, is defined in *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10] which instructs the firmware whether or not to present the multi-boot menu to the user during the boot process. This variable may be accessed by the OS through the *boot-script* variable in the NVRAM partition named *common*.

Personal Use Copy - Not for Reproduction

Requirements:

8–20. **For the Multi-Boot option:** The multi-boot partition must be named *multi-boot*.

8–21. **For the Multi-Boot option:** Each participating operating system must be represented in the *multi-boot* partition as a 4-tuple of null-terminated strings of the form *name*=<*string*> as represented in Table 49 on page 148.

8–22. **For the Multi-Boot option:** The *multi-boot* partition must be terminated with a string of at least two null characters.

8–23. Device and file specifications used in the *multi-boot* partition must follow IEEE 1275, *IEEE Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices* [9] nomenclature conventions.

Table 49. Multi-Boot String Definitions

String Name	String Description
boot-name	Text giving the name of the file(OS) to boot. This will be used by the multi-boot utility to build the user menu.
boot-path	The Open Firmware device path and file specification of the file to boot.
icon-path	The Open Firmware device path and file specification of icon to be used on the user menu.
config-var	These are overrides to the OF configuration variables. The string is a list of <*configuration variable name*>=<*value*> substrings separated by spaces.

8.4.7 Free Space (0x7F)

Requirements:

8–24. All unused NVRAM space must be included in a signature = 0x7F Free Space partition.

8–25. All Free Space partitions must have the **`name`** field set to 0x7...77.

Personal Use Copy - Not for Reproduction

8.5 NVRAM Space Management

A partition should be added or extended by consuming free space. Any signature 0x7F partition whose length is greater than or equal to the space requirements of the new partition may be all or partially consumed by this process. If there is no signature 0x7F partition with sufficient free space, partitions may be moved or deleted to provide the required space. As partitions are created and modified, it is likely that free space will become fragmented; free space consolidation may become necessary.

Requirements:

8–26. A system software component must not move or delete any NVRAM partition unless it is in the ownership class of that partition (see Table 48 on page 144). There are two exceptions to this requirement:

- **a.** Open Firmware may, with the appropriate backup precautions, modify any area of NVRAM in the interest of space management and/or maintaining NVRAM data integrity. Firmware must maintain the required 1 KB of contiguous NVRAM space for each installed operating system following any such modifications.
- **b.** Upon detection of a corrupted partition, the operating system may create free space beginning with the header of the corrupted partition through the end of the NVRAM space, and use this space to reinitialize its partitions.

8–27. The NVRAM partition header checksum must be calculated as shown in Table 47 on page 143.

Software Implementation Note: Operations that manipulate NVRAM partitions should be serialized to prevent conflict with other active processes that require NVRAM access.

Software Implementation Note: Restrictions are placed on which system software components may manipulate which partitions. This is to help ensure system integrity by protecting system-level information that is required by the boot process.

Software Implementation Note: If an operating system finds it necessary to remove a partition owned by another operating system, proper user notification and options should be provided.

Personal Use Copy - Not for Reproduction

Software Implementation Note: A partition should not be extended through internal linkages to other partitions. If this is done, unrecoverable orphan partitions may result. Since any operating system may have any number of partitions defined, a safe way to extend an OS partition is to create an additional partition if free space allows.

Personal Use Copy - Not for Reproduction

I/O Devices

9

This chapter describes requirements for PCI and ISA devices. It adds detail to areas of PCI and ISA architecture that are either unaddressed or optional. It also places some requirements on firmware and operating systems for device support. It provides references to specifications to which devices must comply and gives design notes for devices that run on CHRP systems. This chapter references the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20] frequently. This document is available from IBM and contains the detailed register level architecture for I/O devices.

9.1 PCI Devices

Requirements:

9–1. PCI devices must comply with the *PCI Local Bus Specification,* Revision 2.1 [14].

PCI is rapidly becoming the dominant bus in the computer industry and will increasingly become the foundation for all I/O in CHRP systems as ISA devices are migrated to PCI.

9.1.1 Resource Locking

Requirements:

9–2. PCI devices, excepting bridges, must not depend on the PCI LOCK# signal for correct operation nor require any other PCI device to assert LOCK# for correct operation.

There are some legacy devices on legacy buses which require LOCK#. Additionally, LOCK# is used in some implementations to resolve deadlocks between bridges. These uses of LOCK# are permitted.

9.1.2 PCI Expansion ROMs

Requirements:

9–3. PCI expansion ROMs must have a ROM image with a code type of 1 for Open Firmware as provided in the *PCI Local Bus Specification,* Revision 2.1 [14]. This ROM image must abide by the ROM image format for Open Firmware as documented in the *PCI Bus Binding to IEEE 1275 Standard for Boot (Initialization, Configuration) Firmware* [12]

CHRP systems rely on Open Firmware - not BIOS - to boot. This is why strong requirements for Open Firmware device support are made.

Vital Product Data (VPD) is put in the PCI expansion ROM in accordance with the *PCI Local Bus Specification,* Revision 2.1 [14] when it is provided. Although this is an optional feature, it is strongly recommended that VPD be included in all PCI expansion ROMs.

9.1.3 Assignment of Interrupts to PCI Devices

Requirements:

9–4. All PCI devices must use the Open PIC interrupt controller. They must not use the legacy (8259 derived) interrupt controller which is reserved for ISA. The programming model for the legacy interrupt controller is defined in *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

Personal Use Copy - Not for Reproduction

9–5. PCI devices that do not share Peripheral Memory Space and Peripheral I/O Space of the same PHB must not share the same Open PIC interrupt source.

9–6. When PCI-to-ISA bridges with embedded ISA devices are provided in a PCI part, the part must provide the capability to attach to the legacy interrupt controller defined in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

For further information on the Open PIC interrupt controller refer to Chapter 6, "Interrupt Controller," on page 85.

It is strongly advised that system board designers assign one Open PIC interrupt pin for each interrupt source. Additionally, multi-function PCI devices should have multiple interrupt source pins.

9.1.4 PCI Devices with Required Register Definitions

Requirements:

9–7. PCI devices must implement the programming model (register level definition, interrupts, and so forth) for those devices which are in the minimum system requirements and are specified in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

9–8. The following PCI devices, when used, must adhere to the programming model provided in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20]:

- MESH SCSI Controller
- ADB (Apple Desktop Bus) Controller
- SCC (Serial Communications Controller)
- Bus master IDE Controller
- VGA Compatible Graphics Controller

9–9. If a PCI device is defined in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20] and a specific implementation of that device has additional registers beyond those defined as used by programming, then those additional registers must

Personal Use Copy - Not for Reproduction

be reported in the Open Firmware device tree as reserved (see Section 3.1, “Address Areas,” on page 23).

With regard to requirement 9–7, there may be additional registers defined for the initialization of these PCI devices which the firmware has knowledge of but programming does not. Requirement 9–9 addresses how this situation is handled in the Open Firmware device tree.

For MESH SCSI, ADB, and SCC see the *Macintosh Technology in the Common Hardware Reference Platform* [23] for specific physical, timing, and electrical requirements.

9.1.5 PCI-PCI Bridge Devices

Requirements:

9–10. PCI-to-PCI bridges must be compliant with the *PCI to PCI Bridge Architecture Specification* [16]

9–11. Firmware must initialize PCI-to-PCI bridges to work in CHRP systems. See *PCI Bus Binding to IEEE 1275 Standard for Boot (Initialization, Configuration) Firmware* [12].

9.1.6 Graphics Controller and Monitor Requirements for Clients

The graphics requirements for servers are different from those for portable and personal systems.

Requirements:

9–12. Portable and personal platforms must provide Bi-Endian graphics aperture support as described in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

9–13. Plug-in graphics controllers for portable and personal platforms must provide graphics mode sets in the Open Firmware PCI expansion ROM image in accordance with the *PCI Bus Binding to IEEE 1275 Standard for Boot (Initialization, Configuration) Firmware* [12].

Personal Use Copy - Not for Reproduction

Portable and personal platforms are strongly urged to support some mechanism which allows the platform to electronically sense the display capabilities of monitors.

For graphics controllers that are placed on the system board, the graphics mode sets can be put in system ROM. The mode set software put in the system ROM in this case would be FCode and would be largely or entirely the same as the FCode that would be in the PCI expansion ROM if the same graphics controller was put on a plug-in PCI card.

Refer to Section C.4, "Bi-Modal Devices," on page 276 and the PCI Multimedia Design Guide [29] for information on Bi-Endian graphics apertures.

9.2 ISA Devices

There is a legacy of programming for ISA devices which has little platform performance impact and thus will be preserved for the foreseeable future. The ISA devices of primary interest are listed here and their programming model in given in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

Requirements:

9–14. The following ISA devices, when used, must adhere to the programming model provided in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

- Legacy Interrupt Controller
- Serial Port Controller
- Parallel Port Controller
- Floppy Disk Controller
- DMA Controller
- Audio Controller
- Keyboard and Mouse Controller

9–15. When an ISA device is included in a PCI part, is required by the minimum requirements, and is among those specified in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20], it must completely retain the programming model.

Personal Use Copy - Not for Reproduction

9–16. The system tone and the system audio must be able to be used concurrently.

9–17. ISA devices included in a PCI part must route their interrupt signals to the legacy interrupt controller defined in the *PowerPC Microprocessor Common Hardware Reference Platform: I/O Device Reference* [20].

Personal Use Copy - Not for Reproduction

Error and Event Notification

10

10.1 Introduction

RTAS provides a mechanism which helps operating systems avoid the need for platform-dependent code that checks for, or recovers from, errors or exceptional conditions. The mechanism is used to return information about hardware errors which have occurred as well as information about non-error events, such as power-management interrupts or environmental conditions (for example, temperature or voltage out-of-bounds) which may need OS attention. This permits RTAS to pass hardware event information to the operating system in a way which is abstracted from the platform hardware. This mechanism primarily presents itself to the operating system via two RTAS functions, *event-scan* and *check-exception*, which are described further in Section 7.3.4, "Error and Event Reporting," on page 109.

The *event-scan* function is called periodically to check for the presence or past occurrence of a hardware event, such as a soft failure or voltage condition, which did not cause a program exception or interrupt (for example, an ECC error detected and corrected by background scrubbing activity). The *check-exception* function is called to provide further detail on what platform event has occurred when certain exceptions or interrupts are signalled. The events reported by these two functions are mutually exclusive on any given platform; that is, a platform may choose to notify the OS of a particular event type either through *event-scan* or through an interrupt and *check-exception*, but not both.

Since RTAS is platform-specific, it can examine hardware registers, can often diagnose many kinds of hardware errors down to a root cause, and may even perform some very limited kinds of error recovery on behalf of the operat-

ing system. The reporting format, described in this chapter, permits RTAS to report the type of error which has occurred, what entities in the platform were involved in the error, and whether RTAS has successfully recovered from the error without the need for further operating system involvement. RTAS may not, in many cases, be able to determine all the details of an error, so there are also returned values which indicate this fact. RTAS may optionally provide extended error diagnostic information, as described in Section 10.3.2.2, "Extended Error Log Format Returned by RTAS," on page 174.

A platform-aware OS can handle errors and events with as much sophistication as desired by providing platform-specific code and drivers, and a hardware vendor can provide such code and drivers for specific platforms to make an OS platform-aware. However, the abstractions provided by this architecture enable the handling of most platform errors and events without integrating platform-specific code into each supported OS.

Architecture Note: It is not a goal of the RTAS to diagnose all hardware failures. Most I/O device failures, for example, will be detected and recovered by an associated device driver. RTAS attempts to determine the cause of a problem and report what it finds, to aid the end user (by providing meaningful diagnostic data for messages) and to prevent the loss of error syndrome information. RTAS is never required to correct any problem, but in some cases may attempt to do so. System vendors who want more extensive error diagnosis may create operating system error handlers which contain specific hardware knowledge, or could use RTAS to collect a minimum set of error information which could then be used by diagnostics to further analyze the cause of the error.

10.2 RTAS Error and Event Classes

Table 50 on page 159 describes the predefined classes of error and event notifications that can be presented through the *check-exception* and *event-scan* RTAS functions. More detailed descriptions of these classes are given later in this chapter. Table 50 defines nodes in the Open Firmware device tree which, through an *open-pic-interrupt* property, may list the platform-dependent interrupts related to each class. From this information, operating systems know which interrupts may be handled by calling *check-exception*. The Open Firmware structure for describing these interrupts is defined in the *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10]. Table 50 also defines the mask parameter for the *check-exception* and *event-scan* RTAS functions which limits the search for errors and events to the classes specified.

Personal Use Copy - Not for Reproduction

Table 50. Error and Event Classes with RTAS Function Call Mask

Class Type	Open Firmware Node Name (where the *open-pic-interrupt* property lists the interrupts)	RTAS Function Call Mask (value = 1 enables class)
Internal Errors	*internal-errors*	bit 0
Environmental and Power Warnings	*epow-events*	bit 1
Power Management Events	*power-management-events*	bit 2

Requirements:

10–1. Platform-specific error and event interrupts that a platform provider wants the operating system to enable must be listed in the *open-pic-interrupt* property of the appropriate Open Firmware event class node, as described in Table 50 on page 159.

10–2. To enable platform-specific error and event interrupt notification, operating systems must find the list of interrupts (described in Table 50 on page 159) for each error and event class in the Open Firmware device tree, and enable them.

10–3. Operating systems must have interrupt handlers for the enabled interrupts described in requirement 10–2, which call the RTAS *check-exception* function to determine the cause of the interrupt.

10–4. Platforms which support error and event reporting must provide information to the OS via the RTAS *event-scan* and *check-exception* functions, using the reporting format described in Table 53 on page 172.

10–5. Optional Extended Error Log information, if returned by the *event-scan* or *check-exception* functions, must be in the reporting format described in Table 54 on page 176.

10–6. To provide control over performance, the RTAS event reporting functions must not perform any event data gathering for classes not selected in the event class mask parameter, nor any extended data gathering if the time critical parameter is non-zero or the log buffer length parameter does not allow for an extended error log.

10–7. To prevent the loss of any event notifications, the RTAS event reporting functions must be written to gather and process error and event data

Personal Use Copy - Not for Reproduction

without destroying the state information of events other than the one being processed.

10–8. Any interrupts or interrupt controls used for error and event notification must not be shared between error and event classes, or with any other types of interrupt mechanisms. This allows the operating system to partition its interrupt handling and prevents blocking of one class of interrupt by the processing of another.

10–9. If a platform chooses to report multiple event or error sources through a single interrupt, it must ensure that the interrupt remains asserted or is re-asserted until *check-exception* has been used to process all outstanding errors or events for that interrupt.

10.2.1 Internal Error Indications

Hardware may detect a variety of problems during operation, ranging from soft errors which have already been corrected by the time they are reported, to hard errors of such severity that the OS (and perhaps the hardware) cannot meaningfully continue operation. The mechanisms described in Section 10.1, "Introduction," on page 157 are used to report such errors to the OS. This section describes the architectural sources of errors, and describes a method that platforms can use to report the error. All OSs need to be prepared to encounter the errors reported as they are described here. However, in some platforms more sophisticated handling may be introduced via RTAS, and the OS may not have to handle the error directly. More robust error detection, reporting, and correcting are at the option of the hardware vendor.

The primary architectural mechanism for indicating hardware errors to an operating system is the machine check interrupt. If an error condition is surfaced by placing the system in check stop, it precludes any immediate participation by the operating system in handling the error (that is, no error capture, logging, recovery, analysis, or notification by the operating system). For this reason, the machine check interrupt is preferred over going to the check stop state. However, check stop may be necessary in certain situations where further processing represents an exposure to data integrity. To better handle such cases, a special hardware mechanism may be provided to gather and store residual error data, to be analyzed when the system goes through a subsequent successful reboot.

Less critical internal errors may also be signalled to the operating system through a platform-specific interrupt in the "Internal Errors" class, or by periodic polling with the *event-scan* RTAS function.

Personal Use Copy - Not for Reproduction

Architecture Note: The machine check interrupt will not be listed in the Open Firmware node for the “Internal Errors” class, since it is a standard architectural mechanism. The machine check interrupt mechanism is enabled from software by setting the MSR_{ME} bit =1. Upon the occurrence of a machine check interrupt, bits in SRR1 will indicate the source of the interrupt and SRR0 will contain the address of the next instruction that would have been executed if the interrupt had not occurred. Depending on where the error is detected, machine check interrupt may be surfaced from within the processor, via logical connection to the processor machine check interrupt pin, or via a system bus error indicator (for example, Transfer Error Acknowledge - TEA).

Requirements:

10–10. Operating systems must set MSR_{ME}=1 prior to the occurrence of a machine check interrupt in order to enable machine check processing via the *check-exception* RTAS function.

10–11. For hardware-detected errors, platforms must generate error indications as described in Table 51 on page 162, unless the error can be handled through a less severe platform-specific interrupt, or the nature of the error forces a check stop condition.

10–12. Platforms which detect and report the errors described in Table 51 on page 162 must provide information to the OS via the RTAS *check-exception* function, using the reporting format described in Table 53 on page 172.

10–13. To prevent error propagation and allow for synchronization of error handling, all processors in a multi-processor system must receive any machine check interrupt signalled via the external machine check interrupt pin.

10.2.1.1 Error Indication Mechanisms

Table 51 on page 162 describes the mechanisms by which software will be notified of the occurrence of operational failures during the types of data transfer operations listed below. The assumption here is that the error notification can occur only if a hardware mechanism for error detection (for example, a parity checker) is present. In cases where there is no specific error detection mechanism, the resulting condition, and whether the software will eventually recognize that condition as a failure, is undefined.

Personal Use Copy - Not for Reproduction

Table 51. Error Indications for System Operations

Initiator	Target	Operation	Error Type (if detected)	Indication to Software	Comments
Processor	N/A	Internal	Various	Machine check	Some may cause check stop
Processor	Memory	Load	Invalid address	Machine check	
			System bus time-out	Machine check	
			Address parity on system bus	Machine check	
			Data parity on system bus	Machine check	
			Memory parity or uncorrectable ECC	Machine check	
		Store	Invalid address	Machine check	
			System bus time-out	Machine check	
			Address parity on system bus	Machine check	
			Data parity on system bus	Machine check	
		External cache load	Memory parity or uncorrectable ECC	Machine check	Associated with Instruction Fetch or Data Load
		External cache flush	Cache parity or uncorrectable ECC	Machine check	
		External cache access	Cache parity or uncorrectable ECC	Machine check	Associated with Instruction Fetch or Data Transfer

Personal Use Copy - Not for Reproduction

Table 51. Error Indications for System Operations ***(Continued)***

Initiator	Target	Operation	Error Type (if detected)	Indication to Software	Comments
Processor	I/O	Load	Data parity on I/O bus	Machine check	
			Data parity on system bus	Machine check	
		Store	Data parity on I/O bus	Machine check	
			Data parity on system bus	Machine check	
		Load or Store	Address parity on system bus	Machine check	
			Invalid address	Machine check	
			I/O bus time-out	Machine check	
			Retry count expired	Machine check	
			Target-abort	Machine check	
			Invalid size	Machine check	
			No response from device	Ignore a Store, all-1's returned on a Load	Invalid address or configuration cycle to non-existent device
Processor	Invalid target address	Load or Store	No response from system	Machine check	
I/O	Memory	DMA - I/O to memory	Data parity on PCI bus	PHB Target-aborts operation	May be recoverable; see note following table
			Data parity on system bus	Machine check	
		DMA - memory to I/O	Data parity on PCI Bus	PHB signals PERR#	May be recoverable; see note following table
			Data parity on system bus	Machine check or PHB signals	
			Memory parity or uncorrectable ECC	Machine check	

Personal Use Copy - Not for Reproduction

Table 51. Error Indications for System Operations ***(Continued)***

Initiator	Target	Operation	Error Type (if detected)	Indication to Software	Comments
I/O	Memory	DMA - either direction	Address parity on I/O bus	Machine check	
			Address parity on system bus	Machine check	
			I/O bus time-out	Machine check	
			Invalid address	Machine check	
			PHB TCE extent	PHB target-aborts operation	May be recoverable; see note following table
			PHB page fault	PHB target-aborts operation	May be recoverable; see note following table
			PHB unauthorized access	PHB target-aborts operation	May be recoverable; see note following table
I/O	I/O	DMA - either direction	Data parity on PCI bus	Device signals PERR#	May be recoverable; see note following table
			Data parity on system bus	Machine check	
			Address parity on I/O bus	Machine check	
			Address parity on system bus	Machine check	
			I/O bus time-out	Machine check	
I/O	Invalid target address	DMA - either direction	No response from any device	PCI device master-aborts	Signal device driver as an external interrupt,
PCI I/O Device		Any	Internal, indicated by SERR#	SERR#, causing machine check	Use of SERR# for internal errors is discouraged
ISA I/O Device		Any	ISA bus IOCHK#	Machine check	

Implementation Note: I/O devices should detect the occurrence of PCI Target-abort or PCI data parity errors on DMA and report them via an external interrupt (for possible device driver recovery) or retry the

Personal Use Copy - Not for Reproduction

operation. Since system state has not been lost, reporting these errors via a Machine Check to the CPUs is inappropriate. Some devices or device drivers may cause a catastrophic error. Systems which wish to recover from these types of errors should choose devices and device drivers which are designed to handle them correctly.

10.2.2 Environmental and Power Warnings

Environmental and Power Warnings (EPOW) is an option that provides a means for the platform to inform the operating system of these types of events. The intent is to enable the operating system to provide basic information to the user about these problems and to minimize the logical damage done by these problems. For example, an operating system might want to abort all disk I/O operations in progress to ensure that disk sectors are not corrupted by the loss of power.

These warnings include action codes that the platform can use to influence the operating system behavior when various hardware components fail. For example, a fan failure where the system can continue to operate in the safe cooling range may just generate an action code of WARN_COOLING, but a fan failure where the system cannot operate in the safe cooling range may generate an action code of SYSTEM_HALT.

Implementation Note: Hardware can not assume that the operating system will process or take action on these warnings. These warnings are only provided to the operating system in order to allow the operating system a chance to cleanly abort operations in progress at the time of the warning. Hardware still assumes responsibility for preventing hardware damage due to environmental or power problems.

An operating system that wants to be EPOW-aware will look for the *epow-events* node in the Open Firmware device tree, enable the interrupts listed in its *open-pic-interrupt* property, and provide an interrupt handler to call *check-exception* when one of those interrupts are received.

Requirements:

10–14. If the platform supports Environmental and Power Warnings by including a EPOW device tree entry, then the platform must support the EPOW sensor for the *get-sensor-state* RTAS function.

10–15. The EPOW sensor, if provided, must contain the EPOW action code (defined in Table 52 on page 166) in the least significant 4 bits. In cases where multiple EPOW actions are required, the action code with the

Personal Use Copy - Not for Reproduction

highest numerical value (where 0 is lowest and 7 is highest) must be presented to the operating system. The platform may implement any subset of these action codes, but must operate as described in Table 52 for those it does implement.

10–16. To ensure adequate response time, platforms which implement the EPOW_MAIN_ENCLOSURE or EPOW_POWER_OFF action codes must do so via interrupt and *check-exception* notification, rather than by *event-scan* notification.

10–17. For interrupt-driven EPOW events, the platform must ensure that an EPOW interrupt is not lost in the case where a numerically higher-priority EPOW event occurs between the time when check-exception gathers the sensor value and when it resets the interrupt.

Implementation Note: One way for hardware to prevent the loss of an EPOW interrupt is by deferring the generation of a new EPOW interrupt until the existing EPOW interrupt is reset by a call to the RTAS *check-exception* function. Another way is to ignore resets to the interrupt until all EPOW events have been reported.

Table 52. EPOW Action Codes

Action Code	Value	Description
EPOW_RESET	0	No EPOW event is pending. This action code is the lowest priority.
WARN_COOLING	1	A non-critical cooling problem exists. An EPOW-aware operating system logs the EPOW information.
WARN_POWER	2	A non-critical power problem exists. An EPOW-aware operating system logs the EPOW information.
SYSTEM_SHUTDOWN	3	The system must be shut down. An EPOW-aware operating system logs the EPOW error log information, then schedules the system to be shut down in 10 minutes.
SYSTEM_HALT	4	The system must be shut down quickly. An EPOW-aware operating system logs the EPOW error log information, then schedules the system to be shut down in 20 seconds.

Personal Use Copy - Not for Reproduction

Table 52. EPOW Action Codes ***(Continued)***

Action Code	Value	Description
EPOW_MAIN_ENCLOSURE	5	The system may lose power. The hardware ensures that at least 4 milliseconds of power within operational thresholds is available. An EPOW-aware operating system performs any desired functions, masks the EPOW interrupt, and monitors the sensor to see if the condition changes. Hardware does not clear this action code until the system resumes operation within safe power levels.
EPOW_POWER_OFF	7	The system will lose power. The hardware ensures that at least 4 milliseconds of power within operational thresholds is available. An EPOW-aware operating system performs any desired operations, then attempts to turn system power off. An EPOW-aware operating system does not clear the EPOW interrupt for this action code. This action code is the highest priority.

Software Note: A recommended operating system processing method for an *EPOW_MAIN_ENCLOSURE* event is as follows: Prepare for shutdown, mask the EPOW interrupt, and wait for 50 milliseconds. Then call *get-sensor-state* to read the EPOW sensor. If the EPOW action code is unchanged, wait an additional 50 milliseconds. If the action code is *EPOW_POWER_OFF*, attempt to power off. Otherwise, the power condition may have stabilized, so interrupts may be enabled and normal operation resumed.

10.2.3 Power Management Events

Power Management Events, when implemented, are also reported through either the *check-exception* or *event-scan* functions, depending on whether the events are implemented to report through interrupts or not. The architected events are defined in Chapter 11, “Power Management,” on page 185 and are returned in the fixed portion of the RTAS return value (see Table 53 on page 172).

10.3 RTAS Error and Event Information Reporting

Architecture Note: All data formats listed in this section are either referenced as byte fields (and therefore are independent of Endian orientation), or an indicator in the data structure describes their Endian

Personal Use Copy - Not for Reproduction

orientation. Bits are numbered from left (high-order:0) to right (low-order:7).

10.3.1 Introduction

This section describes the data formats used to report events and errors from RTAS to the Operating System. A common format is used for errors and events to simplify software both in RTAS and in the OS. Both errors and events may have been analyzed to some degree by RTAS, and value judgments may have been applied to decide how serious an error is, or even how to describe it to the OS. These judgments are made by platform providers, since only they know enough about the hardware to decide how serious a problem it is, whether and how to recover from it, and how to map it onto the abstracted set of events and errors that a system is required to know about. There will be cases with some platforms where no reasonable mapping exists, and platform features may not be fully supported by the OS (an example might be a platform with power management features beyond those provided for in this specification). In such cases, error reports may also be non-specific, leaving platform-specific details to platform-aware software.

10.3.2 RTAS Error/Event Return Format

This section describes in detail the return value retrieved by an RTAS call to either the *event-scan* or *check-exception* function.

The return value consists of a fixed part and an optional Extended Error Report, described in the next section, which contains full details of the error. The fixed part is intended to allow reporting the most common problems in a simple way, which makes error detection and recovery simple for operating systems that want to implement a very simple error handling strategy. At the same time, the mechanism is capable of providing full disclosure of the error syndrome information for operating systems which have a more complete error handling strategy.

RTAS can return at most one return code per invocation. If multiple conditions exist, RTAS returns them in descending order of severity on successive calls.

10.3.2.1 Reporting and Recovery Philosophy, and Description of Fields

All RTAS implementations use a common error and event reporting scheme, as described in detail below. It is not required that error recovery be present in

Personal Use Copy - Not for Reproduction

RTAS implementations, nor is it required that a high degree of error recovery or survival be undertaken by operating systems. If such behavior is desired, then specific platform-dependent handlers can be loaded into the OS. However, this section defines return result codes and a philosophy which can be used if aggressive error handling is implemented in RTAS. This section describes the fields of the Error Report format, and the philosophy which should be applied in generating return values from RTAS or interpreting such return codes in an OS.

In general, an OS would look at the Disposition field first to see if an error has been corrected already by RTAS. If not corrected to the OS's satisfaction, the OS would examine the Severity field. Based on that value, and optionally on any information it can use from the Type and other fields, the OS will make a determination of whether to continue or to halt operations. In either case, it may choose to log information regarding the error, using the remaining fields and optional Extended Error Log.

The following sections describe the field values in Table 53 on page 172.

10.3.2.1.1 Version Field

This field is used only to distinguish among present and potential future formats for the remainder of the error report. This value will be incremented if extensions are made to the format described here. Future versions will be backward-compatible; the primary function of this field is for future OSs to identify whether an error report may contain some (unknown at present) feature that was added after the initial version of this specification.

10.3.2.1.2 Severity

This field represents the value judgment of RTAS of how serious the problem being reported should be considered by the OS. A platform-aware OS may choose to ignore the RTAS judgment, but non-platform-aware OSs have no way to make adequate value judgments, and should in general believe the RTAS assessment of the situation.

Errors which are believed to represent a permanent hardware failure affecting the entire system are considered "FATAL." OSs would not attempt to continue normal operation after receiving notice of such an error. OSs may not even be able to perform an orderly shutdown in the presence of a Fatal error, though they may make a policy decision to try.

Less serious errors, but still causing a loss of data or state, are considered "ERRORs." In general, continuing after such an error is questionable, since details of what has failed may not be available, or if available, may not map nicely onto any ongoing activity with which the OS can associate it. However, OSs may make a policy decision (for example, based on the error Type, the Initiator, or the Target) to continue operation after an Error.

Personal Use Copy - Not for Reproduction

There are some types of errors, such as parity errors in memory or a parity error on a transfer between CPU and memory, which occur synchronously with the current process execution context. Such errors are sometimes fatal only to the current thread of execution; that is, they affect only the current CPU state and possibly that of any memory locations being currently referenced. If that context of execution is not essential to the system's operation (for example, if an error trap mechanism is available in the OS and can be triggered to recover the OS to a known state), recovery and continuation may be possible. Or at least, since the memory of the machine is in an undamaged state, the system may be able to be brought down in an orderly fashion. Such errors are reported as having severity "ERROR_SYNC." It is OS dependent whether recovery is possible after such an error, or whether the OS will treat it as a fatal problem.

The "WARNING" return value indicates that a non-state-losing error, either fully recovered by RTAS or not needing recovery, has occurred. No OS action is required, and full operation is expected to continue unhindered by the error. Examples include corrected ECC errors or bus transfer failures which were retried successfully.

The "EVENT" return value is the mechanism RTAS uses to communicate event information to the OS. The event may have been detected by polling using *event-scan* or on the occurrence of an interrupt by calling *check-exception*. In either case, the Error Return value indicates the event which has occurred in the Type field. See the Type description below for a description of specific events and their expected handling.

The "NO ERROR" return value indicates that no error was present. In this case, the remainder of the Error Return fields are not valid and should not be referenced.

10.3.2.1.3 RTAS Disposition

An aggressive RTAS implementation may choose to attempt recovery for some classes of error so a non-platform-aware OS can continue operation in the face of recoverable errors. If RTAS detects an error for which it has recovery code, it attempts such action before it returns a value to the OS (that is, the mechanisms are linked in RTAS and cannot be separately accessed). Note that Disposition is nearly independent from Severity. Severity says how serious an error was, and Disposition says, regardless of severity, whether or not the OS has to even look at it. In general, an OS will first examine Disposition, then Severity.

A return value of "FULLY RECOVERED" means that RTAS was able to completely recover the machine state after the error, and OS operation can continue unhindered. The severity of the problem in this case is irrelevant, though for consistency a "FATAL" error can never be "FULLY RECOVERED."

A return value of "LIMITED RECOVERY" means that RTAS was able to recover the state of the machine, but that some feature of the machine has been

Personal Use Copy - Not for Reproduction

disabled or lost (for example, error checking), or performance may suffer (for example, a failing cache has been disabled). The RTAS implementation may return further information in the extended error log format regarding what action was done or what corrective action failed. In general, a conservative OS will treat this return the same as "NOT RECOVERED," and initiate shutdown. A less conservative OS may choose to let the user decide whether to continue or to shut down.

A value of "NOT RECOVERED" indicates that the RTAS either did not attempt recovery, or that it attempted recovery but was unsuccessful.

10.3.2.1.4 Optional Part Presence

This is a single flag, valid only if the 32-bit Error Return value is located in memory, which indicates whether or not an Extended Error Log Length field and the Extended Error Log follows it in memory. It will be set on an in-memory return result from RTAS if and only if the RTAS call indicated sufficient space to return the Extended Error Log, and the RTAS implementation supports the Extended Error Log.

10.3.2.1.5 Initiator

This field indicates, to the best ability of RTAS to determine it, the initiator of a failed transaction. (Note that in the "Initiator" field of Table 51 on page 162, the value "I/O" indicates one of the defined I/O buses or devices. This field contains finer-grained details of which type of I/O bus failed, if known, and "UNKNOWN" if RTAS cannot tell.)

10.3.2.1.6 Target

If RTAS can determine it, this field indicates the target of a failed transaction.

10.3.2.1.7 Type

This field identifies the general type of the error or event. In some cases (for example, INTERN_DEV_FAIL, or power management events), multiple possible events are grouped together under a common return value. In such cases, a platform-aware OS may use the Extended Error Log to distinguish them. A non-platform-aware OS will generally treat all errors of a given type the same, so it generally will not need to access the Extended Error Log information.

In the table, the EPOW and power management values are associated with a Severity of EVENT. All other values will be associated with Severity values of FATAL, ERROR, ERROR_SYNC, or WARNING, and may or may not be corrected by RTAS.

EPOW is an event type which indicates the potential loss of power or environmental conditions outside the limits of safe operation of the platform. See "Environmental and Power Warnings" on page 165 for more information.

Personal Use Copy - Not for Reproduction

Power Management event types are described in Section 11.2.1.2, “Definition of RTAS Abstracted Power Management Event Types,” on page 199. OSs implementing power management features will pass these events to the appropriate power management software.

Additional Type values will be added in future revisions of the specification. If an OS does not recognize a particular event type, it can examine the severity first, and then choose to ignore the event if it is not serious.

10.3.2.1.8 Extended Error Log Length

This optional 32-bit field is present in memory following the 32-bit Error Return value only if the Optional Part Presence flag is “PRESENT.” If present, it indicates the length in bytes of the Extended Error Log information which immediately follows it in memory. The length does not include this field or the Error Return field, so it may be zero.

10.3.2.1.9 RTAS Error Return Format Fixed Part

The summary portion of the error return is designed to fit into a single 32-bit integer. When used as a data return format in memory, an optional Length field and Extended Error Log data may follow the summary. The fixed part contains a “presence” flag which identifies whether an extended report is present.

In the table below, the location of each field within the integer is included in parentheses after its name. Numerical field values are indicated in decimal unless noted otherwise.

Table 53. RTAS Error Return Format (Fixed Part)

Bit Field Name (bit number(s))	Description, Values (Described in Section 10.3.2.1, “Reporting and Recovery Philosophy, and Description of Fields,” on page 168)
Version (0:7)	A distinct value used to identify the architectural version of message. Current version = (1)
Severity (8:10)	Severity level of error/event being reported: FATAL (5) ERROR (4) ERROR_SYNC (3) WARNING (2) EVENT (1) NO_ERROR (0) reserved for future use (6-7)

Personal Use Copy - Not for Reproduction

Table 53. RTAS Error Return Format (Fixed Part) ***(Continued)***

Bit Field Name (bit number(s))	Description, Values (Described in Section 10.3.2.1, "Reporting and Recovery Philosophy, and Description of Fields," on page 168)
RTAS Disposition (11:12)	Degree of recovery which RTAS has performed prior to return after an error (value is FULLY_RECOVERED if no error is being reported): FULLY_RECOVERED(0) Note: Cannot be used when Severity is "FATAL". LIMITED_RECOVERY(1) NOT_RECOVERED(2) reserved for future use (3)
Optional_Part_Presence (13)	Indicates if an Extended Error Log follows this 32-bit quantity in memory: PRESENT (1): The optional Extended Error Log is present. NOT_PRESENT (0): The optional Extended Error Log is not present.
Reserved (14:15)	Reserved for future use (0:3)
Initiator (16:19)	Abstract entity that initiated the event or the failed operation: UNKNOWN (0): Unknown or Not Applicable CPU (1): A CPU failure (in an MP system, the specific CPU is not differentiated here) PCI (2): PCI host bridge or PCI device ISA (3): ISA bus bridge or ISA device MEMORY (4): Memory subsystem, including any caches POWER_MANAGEMENT (5): Power Management sub-system Reserved for future use (6-15)
Target (20:23)	Abstract entity that was apparent target of failed operation (UNKNOWN if Not Applicable): Same values as Initiator field

Personal Use Copy - Not for Reproduction

Table 53. RTAS Error Return Format (Fixed Part) ***(Continued)***

Bit Field Name (bit number(s))	Description, Values (Described in Section 10.3.2.1, "Reporting and Recovery Philosophy, and Description of Fields," on page 168)
Type (24:31)	General event or error type being reported: Internal Errors: RETRY (1): too many tries failed, and a retry count expired TCE_ERR (2): range or access type error in an access through a TCE INTERN_DEV_FAIL (3): some RTAS-abstracted device has failed (for example, TODC) TIMEOUT (4): intended target did not respond before a time-out occurred DATA_PARITY (5): Parity error on data ADDR_PARITY(6): Parity error on address CACHE_PARITY (7): Parity error on external cache ADDR_INVALID(8): access to reserved or undefined address, or access of an unacceptable type for an address ECC_UNCORR (9): uncorrectable ECC error ECC_CORR (10): corrected ECC error RESERVED (11-63): Reserved for future use Environmental and Power Warnings: EPOW(64): See Extended Error Log for sensor value RESERVED (65-95): Reserved for future use Power Management Events(96-159): power management event occurred - see Table 62 on page 199 for defined event values Reserved for future use (160-223) Vendor-specific events(224-255): Non-architected Other (0): none of the above
Extended Error Log Length (32:63)	Length in bytes of Extended Error Log information which follows (see 10.3.2.2, "Extended Error Log Format Returned by RTAS," on page 174)

Typically, most operating systems care about, and have handlers for, only a few specific errors. Since coding of an error is unique in the above scheme, an operating system can check for specific errors, then if nothing matches exactly, look at more generic parts of the error message. This permits generic error message generation for the user, providing the basic information that RTAS delivered to the operating system. Platforms may provide more complete error diagnosis and reporting in RTAS, combined with off-line diagnostics which take advantage of the information reported from previous failures.

10.3.2.2 Extended Error Log Format Returned by RTAS

The following tables define an extended error log format by which the RTAS can optionally return detailed information to the software about a hardware error condition. This format is also intended to be usable as residual error log

Personal Use Copy - Not for Reproduction

data in NVRAM, so that the operating system could alternatively retrieve error data after an error event which caused a reboot.

Figure 12 and Table 54 on page 176 shows the general layout for the extended error log format, while Tables 55 through 60 show the detailed layout of bytes 12 through 39. The detail area format is determined by bits 4:7 of byte 2, which indicate the error log type.

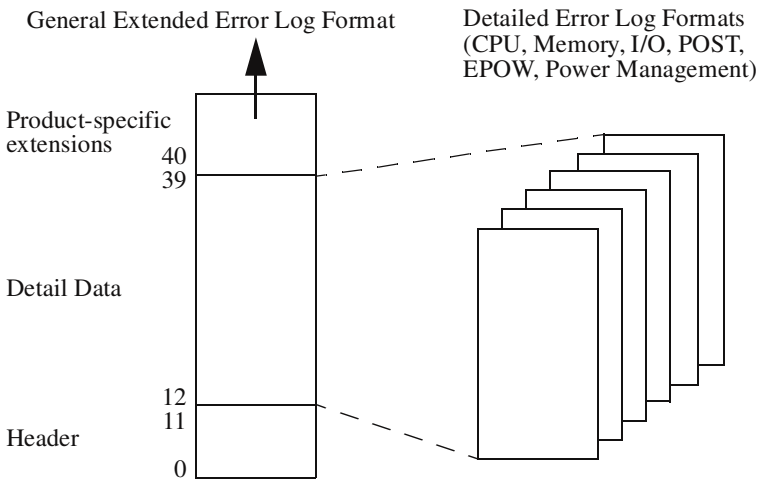

Figure 12. Layout of extended error log format from RTAS.

Extended product-unique data may be added to the end of the error log buffer (starting at byte 40) for capture and logging by the operating system. Platforms which provide extended data indicate the maximum length of the error log buffer in the *rtas-error-log-max* RTAS property in the Open Firmware device tree, so that the operating system can allocate a buffer large enough to hold the extended error log data when calling the RTAS *event-scan* or *check-exception* functions. If the allocated buffer is not large enough to hold all the error log data, the data is truncated to the size of the buffer.

Personal Use Copy - Not for Reproduction

Table 54. RTAS General Extended Error Log Format

Error Log Format		
Byte	Bit(s)	Description
0	0	1 = Log Valid
	1	1 = Unrecoverable Error
	2	1 = Recoverable (correctable or successfully retried) Error
	3	1 = Unrecoverable Error, Bypassed - Degraded operation (for example, Single CPU taken off-line, bad cache bypassed)
	4	1 = Predictive Error - Error is recoverable, but indicates a trend toward unrecoverable failure (for example, correctable ECC error threshold)
	5	1 = “New” Log (always 1 for data returned from RTAS)
	6	1 = Addresses/Numbers are Big-Endian format, 0 = Little-Endian Note: This bit will always be set to the Endian mode in which RTAS was instantiated.
	7	Reserved
1	0:7	Reserved
2	0	Set to 1 - (Indicates log is in PowerPC format)
	1:2	Reserved
	3	1 = No failing address was available for recording within the log’s Detailed Log Data, so the address field is invalid
2	4:7	Log format indicator, defines format used for bytes 12-39: (0) Reserved (1) CPU-detected failure, see Table 55 on page 178 (2) Memory-detected failure, see Table 56 on page 178 (3) I/O-detected failure, see Table 57 on page 180 (4) Power-On Self Test (POST) failure, see Table 58 on page 182 (5) Environmental and Power Warning, see Table 59 on page 183 (6) Power Management Event, see Table 60 on page 183 (7-11) Reserved (12-15) Vendor-specific

Note: Some of these elements are sensitive to Endian orientation, and are indicated by an “*” in the Byte column. Byte 0, Bit 6 of the log structure indicates the Endian orientation.

Personal Use Copy - Not for Reproduction