base address of the ISU with the lowest numbered interrupt first. All registers are aligned on 16-byte boundaries, as required by the Open PIC Specification.

Proposed requirements:

- Platforms must have an IDU.
- The registers in the IDU must be placed at offsets to the base address defined in the Open PIC Specification.
- Platforms must support at least one I/O interrupt group, either via the IDU or via an external ISU.
- The ISRs in an external ISU must be placed contiguously starting at the base address specified for it with the lowest numbered interrupt first.
- All registers in the CHRP interrupt controller must be aligned on 16-byte boundary and must be implemented in Little Endian format.

Hardware Implementation Note: A centralized, single-chip implementation is covered by the distributed scheme just outlined, as a special case. In such an implementation, the IDU supports the only I/O interrupt group in the system.

Software Implementation Note: Open Firmware *reg* properties for the interrupt controller supply base addresses of the IDU and of the external ISUs, if any. Open Firmware *interrupt-ranges* properties of the interrupt controller specify the interrupt groups supported by the IDU and by the external ISUs, if any. Software should use Open Firmware properties of the interrupt controller to locate the Open PIC registers.

Personal Use Copy - Not for Reproduction

Blank page when cut - 90

Run-Time Abstraction Services

7

7.1 RTAS Introduction

The Run-Time Abstraction Services (RTAS) functions are provided by CHRP platforms to insulate operating systems from having to know about and manipulate a number of key platform functions which ordinarily would require platform-dependent code. Operating systems call these functions rather than manipulating hardware registers directly, reducing the need for platform tailoring by OS suppliers. This method of abstracting access to these platform functions also permits hardware vendors considerable flexibility in hardware implementation. Since RTAS is provided by the hardware vendor, this approach places the responsibility for supporting the platform with the hardware vendor, not the OS vendor. This permits separating the schedules of hardware and software, permits different suppliers to provide each, and reduces the release and test requirements for OSs, since they can be tested to conform to the RTAS interfaces and not to every specific hardware configuration.

In order for platforms to achieve this separation of operating system code from platform dependencies, RTAS defines an interface between the platform and the operating systems that provides control of some of the common devices found on all CHRP platforms. RTAS is a system programming interface that is realized on a specific platform by an RTAS implementation. The RTAS implementation provides the platform specific processing of the common components while operating system drivers are necessary to provide device specific processing for I/O adaptors. In general, operating systems should not access RTAS resources directly. They should call RTAS to control the device.

RTAS limits itself to the run-time control of non-I/O, typically system board-resident hardware features. Traditionally, features such as these have almost always been implemented differently on different platforms. This list of features includes things like non-volatile memory, time-of-day clock, memory bridges, and power management control functions. Such differences traditionally require much effort and require platform-dependent code in OSs. RTAS permits an OS to operate over a much wider range of platforms without specialized code for each platform. The presence of RTAS does not prevent an OS from incorporating customized code to manipulate arbitrary platform hardware, provided such hardware is defined in the Open Firmware Device Tree as being present. We refer to such OSs as being "aware" of the platform feature.

The role of RTAS versus Open Firmware is very important to understand. Open Firmware and RTAS are both vendor-provided software, and both are tailored by the hardware vendor to manipulate the specific platform hardware. However, RTAS is intended to be present *during* the execution of the OS, and to be called by the OS to access platform hardware features on behalf of the OS, whereas Open Firmware *need not* be present when an OS is running. This frees Open Firmware's memory to be used by applications. RTAS is small enough to painlessly coexist with OSs and applications.

This chapter uses the term RTAS to refer both to the architected RTAS interface and to an RTAS implementation.

7.2 RTAS Environment

Since RTAS provides an interface definition between the operating system and the firmware provided by the platform vendor, both the operating system and the firmware on the hardware platform must use the calling convention as defined in this chapter.

RTAS must operate in an environment that does not interfere with the operating system. It can not cause any exceptions, nor can it depend on any particular virtual memory mappings. For this reason, it runs in real mode with exceptions disabled.

All RTAS functions are invoked from the operating system by calling the *rtas-call* function. The address of this function is obtained from Open Firmware when RTAS is instantiated. See requirement 7–13 for more details. RTAS will determine what function to invoke based on the data passed into the *rtas-call* function. This section describes the mechanisms used to invoke the *rtas-call* function, the machine state, register usage, resource allocation, and the invocation requirements.

Personal Use Copy - Not for Reproduction

7.2.1 Machine State

When RTAS functions are invoked, the calling processor shall have address translations, floating point, and most other exceptions disabled and it shall be running in privileged state.

Requirements:

7–1. RTAS must be called in "real mode," that is, all address translation must be disabled. Bits IR and DR of the MSR register must be zero.

7–2. RTAS must be called in privileged mode, and the PR bit of the MSR must be set to 0.

7–3. RTAS must be called with external interrupts disabled, and the EE bit of the MSR must be set to 0.

7–4. RTAS must be called with trace disabled, and the SE and BE bits of the MSR must be set to 0.

7–5. RTAS must be called with floating point disabled, and the FE0, FE1 and FP bits must be set to 0.

7–6. RTAS must be called with the SF and LE bits of the MSR set to the same values that were in effect at the time that RTAS was instantiated.

7–7. With the exception of the DR and RI bits, RTAS must not change the state of the machine by modifying the MSR.

7–8. If *rtas-call* is entered in a non-recoverable mode, indicated by having the RI bit of the MSR set equal to 0, then RTAS must not enter a recoverable mode by setting the RI bit to 1.

7–9. If called with RI of the MSR equal to 1, then RTAS must protect its own critical regions from recursion by setting the RI bit to 0 when in the critical regions.

Software Implementation Note: If the ME bit is left enabled, the operating system's exception handler must be aware that RTAS might have been running and that various processor registers might not be in the expected state for an interrupted operating system process. If the operating system cannot tolerate this, it should disable ME while RTAS is running, or set RI to zero which would preclude recoverability but permit logging machine checks.

Software Implementation Note: There are some provisions for recursive calls to RTAS error handling functions. Therefore, RTAS should set the

Personal Use Copy - Not for Reproduction

RI bit in the MSR if SRR0/SRR1 or any other RTAS resource is in a state where information would be lost and prohibit recovery.

Software Implementation Note: Requirement 7–6 implies that RTAS must be prepared to be instantiated in either Big-Endian or Little-Endian modes, and to be able to be instantiated in 64-bit mode on platforms that can support 64-bit execution.

7.2.2 Register Usage

Requirements:

7–10. Except as required by a specific function, RTAS must not modify the following operating environment registers: TB, DEC, SPRG0-SPRG3, EAR, DABR, SDR1, ASR, SR0-SR15, FPSCR, FPR0-FPR3, and any processor specific registers.

7–11. RTAS must preserve the following user mode registers: R1-R2, R13-R31, and CR.

7–12. RTAS must preserve the following operating environment registers: MSR, DAR, DSISR, IBAT0-IBAT3, and DBAT0-DBAT3.

Software Implementation Note: RTAS is entered in real mode (with address translation turned off). In this mode, all data accesses are assumed to be cached in copy back mode with memory coherence required. Since these settings may not be appropriate for all accesses, RTAS is free to use the Block Address Translation registers to set up its own virtual mappings. The operating system machine check handler can only depend on those registers that are required to be unchanged (see requirement 7–10).

Software Implementation Note: RTAS either may not change the preserved registers, or may save them as long as they are restored before returning to the operating system.

Software Implementation Note: The SRR0-SRR1, LR, CTR, XER registers, as well as any reservations made via the load and reserve instructions, need not be preserved.

Personal Use Copy - Not for Reproduction

7.2.3 RTAS Critical Regions

The operating system that uses RTAS is responsible for protecting RTAS and devices used by RTAS from any simultaneous accesses that could corrupt memory or device registers. Such corruption could be caused by simultaneous execution of RTAS code, or by a device driver accessing a control register that is also modified by RTAS. In a single processor system, most recursive calls to RTAS are prevented by clearing the EE bit of the MSR. This will limit recursive calls to RTAS to those made from the machine check handler. This handler may need to call various RTAS services such as *check-exception* or *system-reboot* even if the error was detected while in a RTAS service.

The operating system and RTAS must co-exist on the same platform. RTAS must not change device registers that are used by the operating system, nor may the operating system change device registers on devices used by RTAS. With the advent of more and more integration into common super parts, some of these registers may physically reside on the same component. In this section, device implies the collection of common registers that together perform a function. Each device must be represented in the Open Firmware Device Tree.

Requirements:

7–13. Except as noted in requirement 7–19, the operating system must ensure that RTAS calls are not re-entered and are not simultaneously called from different processors in a multi-processor system.

7–14. Any RTAS access to device or I/O registers specified in this document must be made in such a way as to be transparent to the operating system.

7–15. Any device that is used to implement the RTAS abstracted services must have the property *used-by-rtas* in the Open Firmware Device Tree. However, if the device is only used by the *suspend, hibernate, power-off,* and *system-reboot* calls, which do not return directly to the operating system, the property should not be set. The *display-character* device must be marked *used-by-rtas* only if it is a specialized device used only for *display-character.*

7–16. Platforms must be designed such that accesses to devices that are marked *used-by-rtas* have no side effects on other registers in the system.

7–17. Any operating system access to devices specified as *used-by-rtas* must be made in such a way as to be transparent to RTAS.

Personal Use Copy - Not for Reproduction

7–18. RTAS must not generate any exceptions (for example, no alignment exceptions, page table walk exceptions, etc.).

7–19. The operating system machine check and soft reset handlers may call the RTAS services:

- *nvram-fetch*
- *nvram-store*
- *check-exception*
- *set-indicator*
- *system-reboot*
- *set-power-level(0,0)*
- *power-off*

Software Implementation Note: While RTAS must not generate any exceptions, it is still possible that a machine check interrupt may occur during the execution of a RTAS function. In this case the machine check handler may be entered from the RTAS service.

Software Implementation Note: It is permissible for an operating system exception handler to make an RTAS call as long as requirements 7–13 and 7–10 are met. In particular, it is expected that the RTAS *check-exception* will be called from the operating system exception handlers.

7.2.4 Resource Allocation and Use

During execution, RTAS will require memory for both code and data. This memory may be in RAM, in a private memory area only known by the system firmware, or in memory allocated by the operating system for RTAS use. RTAS should use this memory for its stack and any state savings.

Requirements:

7–20. The operating system must allocate *rtas-size* bytes of contiguous real memory as RTAS private data area. This memory must be aligned on a 4096 byte boundary and may not cross a 256 megabyte boundary.

7–21. The RTAS private data area must not be accessed by the operating system.

Personal Use Copy - Not for Reproduction

7–22. Except for the RTAS private data area, the argument buffer, System Memory pointed to by any reference parameter in the argument buffer, and any other System Memory areas explicitly permitted in this chapter, RTAS must not modify any System Memory. RTAS may, however, modify System Memory during error recovery provided that such modifications are transparent to the operating system.

7–23. If the operating system moves or otherwise alters the addresses assigned to ISA or PCI devices that are marked *used-by-rtas* after it has instantiated RTAS, then the operating system must restart RTAS by calling *restart-rtas* prior to making any further RTAS calls.

7–24. RTAS must execute in a timely manner and may not sleep in any fashion nor busy wait for more than a very short period of time.

Software Implementation Note: A RTAS call should take the same amount of time to perform a service that it would take an operating system to perform the same function. A specific goal is that RTAS primitives should take less than a few tens of microseconds.

Software Implementation Note: Resources that need to be manipulated by RTAS, such as NVRAM, may reside on the PCI bus. If the operating system moves these devices, then RTAS must be restarted.

7.2.5 Instantiating RTAS

RTAS is instantiated by an explicit client interface service call into Open Firmware. The Open Firmware Device Tree contains a property (*rtas-size*, under the *rtas* node) which defines how much real memory RTAS requires. The operating system allocates *rtas-size* bytes of real memory, and then invokes the *instantiate-rtas* method of the RTAS node, passing this address as the *rtas-base-address* input argument. Firmware binds RTAS to that address, binds the addresses of devices that RTAS uses, performs any RTAS initialization, and returns the address of the *rtas-call* function that is appropriate for the current machine state.

Requirements:

7–25. The *instantiate-rtas* Open Firmware method must have the arguments specified in Table 11 on page 98.

Personal Use Copy - Not for Reproduction

Table 11. *instantiate-rtas* Argument Call Buffer

Parameter Type	Name	Values
In	*rtas-base-address*	Real Address of RTAS memory
Out	rtas-call	Real address used to invoke RTAS functions

7–26. RTAS must operate in the endian mode in effect at the time of the *instantiate-rtas* call.

Software Implementation Note: The firmware may provide two distinct implementations, one for each endianess, or may provide a single implementation that is bi-endian. The implementation technique is left up to the designer of the firmware.

7.2.6 RTAS Device Tree Properties

The Open Firmware Device Tree must contain an *rtas* device node that describes the implemented RTAS features and the output device supported by RTAS. Within this device node will be properties that describe the RTAS functions implemented by the firmware. For every implemented function, there will be an Open Firmware property whose name is the same as the RTAS function. The value of this property is passed into the *rtas-call* function when making a RTAS call. Note that some RTAS functions are optional and some are required. This is defined in Table 12 on page 99.

Requirements:

7–27. The Open Firmware Device Tree must contain a device node named *rtas* which describes the RTAS implementation.

7–28. The RTAS device node must have a property for each implemented RTAS function in Table 12 on page 99. The value of this property is a token that is passed into the *rtas-call* function to indicate which RTAS function to invoke.

Personal Use Copy - Not for Reproduction

Table 12. RTAS Tokens for Functions

RTAS property/function	Required?	Notes
restart-rtas	Required	
nvram-fetch	Required	Execution time proportional to amount of data
nvram-store	Required	Execution time proportional to amount of data
get-time-of-day	Required	
set-time-of-day	Required	
set-time-for-power-on		
event-scan	Required	
check-exception	Required	
read-pci-config	Required	
write-pci-config	Required	
display-character		
set-indicator	Required	Some specific indicators are required, and some are optional
get-sensor-state		
set-power-level	Required in Power Managed Platforms	
get-power-level		
assume-power-management		
relinquish-power-management		
power-off		Provided for platforms with software controlled power off capability, with or without other Power Management capability
hibernate		
suspend		
system-reboot	Required	

Personal Use Copy - Not for Reproduction

Table 12. RTAS Tokens for Functions ***(Continued)***

RTAS property/function	Required?	Notes
cache-control	Required in Power Managed Platforms	
	Required in SMP Platforms	
freeze_time_base	Required in SMP Platforms	Turn off time base
thaw_time_base		Turn time base back on
stop-self		
start-cpu		

7–29. The Open Firmware properties listed in Table 13 on page 100 must be in the RTAS Device Tree node prior to booting the operating system.

Table 13. Open Firmware Device Tree Properties

name	value
rtas-size	integer size of RTAS memory area in bytes
rtas-version	An integer encoding of the RTAS interface version. This document describes version 1.
rtas-event-scan-rate	The rate, in calls per minute, at which rtas-event-scan should be called by the operating system. See Section 7.3.4.1, “Event-scan,” on page 110.
rtas-display-device	The *phandle* of the device node used by the RTAS call, display-character
rtas-error-log-max	The maximum size of an extended error log.

7–30. All RTAS functions listed as “Required” in Table 12 on page 99 must be implemented in RTAS.

7–31. For the Power Management option: The functions listed as “Required in Power Managed Platforms” in Table 12 on page 99 must be implemented in RTAS.

7–32. For the Symmetric Multiprocessor option: The functions listed as “Required in SMP Platforms” in Table 12 on page 99 must be implemented in RTAS.

Personal Use Copy - Not for Reproduction

Software Implementation Note: It is permitted for RTAS not to implement those functions that are not appropriate or not needed on a particular platform. For example, if the system does not have power management features, then it is perfectly reasonable not to implement the RTAS power management calls.

Software Implementation Note: Vendors may introduce private RTAS calls of their own. If they do, the property names should be of the form *"vendor,property"* where *vendor* is a company name string as defined by Open Firmware. Future versions of this architecture will not choose RTAS property names that include a comma.

7.2.7 Calling Mechanism and Conventions

RTAS is called through a mechanism similar to the Open Firmware client interface service. An argument buffer is constructed which describes the desired RTAS call. This description includes an indication of the RTAS call that is being invoked, the number and value of the input parameters, the number of result values, and space for each of the result values.

Requirements:

7–33. In order to make an RTAS call, the operating system must construct an argument call buffer aligned on an eight byte boundary in physically contiguous real memory as described by Table 14 on page 101.

Table 14. RTAS Argument Call Buffer

Cell Number	Use
1	Token Specifying which RTAS Call
2	Number of Input Parameters
3	Number of Output Parameters
4	First Input Parameter
...	Other Input Parameters
4 + Number of Inputs -1	Last Input Parameter
4 + Number of Inputs	First Output Value

Personal Use Copy - Not for Reproduction

Table 14. RTAS Argument Call Buffer ***(Continued)***

Cell Number	Use
...	Other Output Parameters
4 + Number of Inputs + Number of Outputs -1	Last Output Value

7–34. If the system is a 32-bit system, or if the SF bit of the MSR was 0 when RTAS was instantiated, then all cells in the RTAS argument buffer must be 32-bit sign extended values that are aligned to 4 byte boundaries.

7–35. If the SF bit of the MSR was 1 when RTAS was instantiated, then all cells in the RTAS argument buffer must be 64-bit values that are aligned to 8 byte boundaries.

7–36. RTAS functions must be invoked by branching to the *rtas-call* address which is returned by the *instantiate-rtas* Open Firmware method (see Table 11 on page 98).

7–37. Register R3 must contain the argument buffer's real address when *rtas-call* is invoked.

7–38. Register R4 must contain the real address of the RTAS private data area when *rtas-call* is invoked (see requirement 7–20).

7–39. The Link Register must contain the return address when *rtas-call* is invoked.

Software Implementation Note: RTAS is not required to perform sanity checking of its input parameters. Using invalid values for any parameter in a RTAS argument buffer gives undefined results.

Software Implementation Note: The token that specifies the RTAS call is obtained by looking up the desired call from the *rtas* node of the Open Firmware Device Tree.

Software Implementation Note: Most operating systems will need to implement simple wrappers to provide an interface that is natural for the operating system (for example C function calls to the RTAS interface). These interfaces are operating system specific and are beyond the scope of this specification.

Personal Use Copy - Not for Reproduction

7.2.8 Return Codes

Requirements:

7–40. The first output value of all the RTAS functions must be a status word which denotes the result of the call. The status word will take on one of the values in Table 15 on page 103. Non-negative values indicate success.

Table 15. RTAS Status Word Values

Values	Status Word Meanings
0	RTAS function call succeeded.
-1	RTAS function call encountered a hardware error or failed for some unspecified reason.
-2	A necessary hardware device was busy, and the requested function could not be performed. The operation should be retried at a later time.
9000-9999	Reserved for vendor specific success codes.
-9000- (-9999)	Reserved for vendor specific error codes.
Additional Negative Numbers	An error was encountered. The meaning of this error is specific to the RTAS function that was invoked.
Additional Positive Numbers	The function succeeded. The meaning of the status word is specific to the RTAS function that was invoked.

7.3 RTAS Call Function Definition

This section specifies the semantics of all the RTAS calls. It specifies the RTAS function name, the contents of its argument call buffer (its token, input parameters, and output values) and semantics.

7.3.1 restart-rtas

If the operating system moves or otherwise alters addresses assigned to PCI or other buses, the resources used by RTAS may be assigned to different locations. RTAS needs to be informed of this change so it can continue to access its resources. It is the responsibility of RTAS to determine the new locations of its

Personal Use Copy - Not for Reproduction

devices by reading PCI configuration space or referencing unarchitected registers known to RTAS code.

Requirements:

7–41. RTAS must implement a *restart-rtas* function that uses the argument call buffer defined by Table 16 on page 104.

Table 16. *restart-rtas* Argument Call Buffer

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

7–42. If any device marked *used-by-rtas,* or a device used as the *rtas-display-device* if the OS will in the future call the *display-character* function, is moved or reconfigured, then the operating system must invoke *restart-rtas* before using any other RTAS function.

7–43. RTAS must update any necessary configuration state information based on the current configuration of the machine when *restart-rtas* is called.

7.3.2 NVRAM Access Functions

The architecture requires an area of non-volatile memory (NVRAM) to hold Open Firmware options, RTAS information, machine configuration state, operating system state, diagnostic logs, etc. The type and size of NVRAM is specified in the Open Firmware Device Tree. The format of NVRAM is detailed in Chapter 8, “Non-Volatile Memory,” on page 141.

In order to give a shrink wrapped operating system the ability to access NVRAM on different platforms that may use different implementations or locations for NVRAM, some layer of abstraction must be provided to the operating system. The functions in this section provide an interface for reading and writing NVRAM.

Personal Use Copy - Not for Reproduction

7.3.2.1 nvram-fetch

The RTAS function *nvram-fetch* will copy data from a given offset in NVRAM into the user specified buffer.

Requirements:

7–44. RTAS must implement an *nvram-fetch* function that returns data from NVRAM using the argument call buffer defined by Table 17 on page 105.

Table 17. *nvram-fetch* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *nvram-fetch*
	Number Inputs	3
	Number Outputs	2
	Index	Byte offset in NVRAM
	Buffer	Real address of data buffer
	Length	Size of data buffer (in bytes)
Out	Status	0: Success -1: Hardware Error -3: Parameter out of range
	Num	Number of bytes successfully copied

7.3.2.2 nvram-store

The RTAS function *nvram-store* will copy data from the user specified buffer to a given offset in NVRAM.

Requirements:

7–45. RTAS must implement an *nvram-store* function that stores data in NVRAM using the argument call buffer defined by Table 18 on page 106.

Personal Use Copy - Not for Reproduction

Table 18. *nvram-store* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *nvram-store*
	Number Inputs	3
	Number Outputs	2
	Index	Byte number in NVRAM
	Buffer	Real address of data buffer
	Length	Size of data buffer (in bytes)
Out	Status	0: Success -1: Hardware Error -3: Parameter out of range
	Num	Number of bytes successfully copied

7–46. If the *nvram-store* operation succeeded, the contents of NVRAM must have been updated to the user specified values. The contents of NVRAM are undefined if the RTAS call failed.

7–47. The caller of the *nvram-store* RTAS call must maintain the NVRAM partitions as specified in Chapter 8, “Non-Volatile Memory,” on page 141.

7.3.3 Time of Day

The minimum system requirements include a non-volatile real-time clock which maintains the time of day even if power to the machine is removed. Minimum requirements for this clock are described in Table 2 on page 19.

7.3.3.1 Time of Day Inputs/Outputs

Requirements:

7–48. The date and time inputs and outputs to the RTAS time of day function calls are specified with the year as the actual value (for example, 1995), the month as a value in the range 1-12, the day as a value in the range 1-31, the hour as a value in the range 0-23, the minute as a value in the

Personal Use Copy - Not for Reproduction

range 0-59, and the second as a value in the range 0-59. The date must also be a valid date according to common usage: the day range being restricted for certain months, month 2 having 29 days in leap years, etc.

7–49. Operating systems must account for local time, for daylight savings time when and where appropriate, and for leap seconds.

7–50. RTAS must account for leap years.

7.3.3.2 Get-time-of-day

Requirements:

7–51. RTAS must implement a *get-time-of-day* call using the argument call buffer defined by Table 19 on page 107.

Table 19. *get-time-of-day* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *get-time-of-day*
	Number Inputs	0
	Number Outputs	8
Out	Status	0: Success -1: Hardware Error -2: Clock Busy, Try again later
	Year	Year
	Month	1-12
	Day	1-31
	Hour	0-23
	Minute	0-59
	Second	0-59
	Nanoseconds	0-999999999

7–52. RTAS must read the current time and set the output values to the best resolution provided by the platform.

Personal Use Copy - Not for Reproduction

7.3.3.3 Set-time-of-day

Requirements:

7–53. RTAS must implement a *set-time-of-day* call using the argument call buffer defined by Table 20 on page 108:

Table 20. *set-time-of-day* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *set-time-of-day*
	Number Inputs	7
	Number Outputs	1
	Year	Year
	Month	1-12
	Day	1-31
	Hour	0-23
	Minute	0-59
	Second	0-59
	Nanosecond	0-999999999
Out	Status	0: Success -1: Hardware Error -2: Clock Busy, Try again later

7–54. RTAS must set the time of day to the best resolution provided by the platform.

Software Implementation Note: The operating system maintains the clock in UTC. This allows the various operating systems and diagnostics to co-exist with each other and provide uniform handling of time. Refer to Table 2 on page 19 for further details on the time of day clock.

Personal Use Copy - Not for Reproduction

7.3.3.4 Set-time-for-power-on

Some platforms provide the ability to set a time to cause the platform power on. The *set-time-for-power-on* call provides the interface to the operating system for setting this timer.

Requirements:

7–55. RTAS must implement the *set-time-for-power-on* call using the argument call buffer defined by Table 20 on page 108:

Table 21. *set-time-for-power-on* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *set-time-for-power-on*
	Number Inputs	7
	Number Outputs	1
	Year	Year
	Month	1-12
	Day	1-31
	Hour	0-23
	Minute	0-59
	Second	0-59
	Nanosecond	0-999999999
Out	Status	0: Success -1: Hardware Error -2: Clock Busy, Try again later

7–56. If the system is in a powered down state at the time scheduled by *set-time-for-power-on* (within the accuracy of the clock), then power must be reapplied and the system must go through its power on sequence.

7.3.4 Error and Event Reporting

The error and event reporting RTAS calls are designed to provide an abstract interface into hardware registers in the system that may contain correctable or

Personal Use Copy - Not for Reproduction

non-correctable errors and to provide an abstract interface to certain platform events that may be of interest to the operating system. Such errors and events may be detected either by a periodic scan or by an exception trap.

These functions are not intended to replace the normal error handling in the operating system. Rather, they enhance the operating system's abilities by providing an abstract interface to check for, report, and recover from errors or events on the platform that are not necessarily known to the operating system.

The operating system uses the error and event RTAS calls in two distinct ways:

1. Periodically, the operating system calls *event-scan* to have the system firmware check for any errors or events that have occurred.
2. Whenever the operating system receives an interrupt or exception that it can not fully process, it must call *check-exception.*

The first case covers all errors and events that do not signal their occurrence with an interrupt or exception. The second case covers those errors and events that do signal with an interrupt or exception. It is platform dependent whether any specific error or event causes an interrupt on that platform.

Requirements:

7–57. RTAS must return the event generated by a particular interrupt or event source by either *check-exception* or *event-scan*, but not both.

7.3.4.1 Event-scan

Requirements:

7–58. RTAS must implement an *event-scan* call using the argument call buffer defined by Table 22 on page 111.

Personal Use Copy - Not for Reproduction

Table 22. *event-scan* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *event-scan*
	Number Inputs	4
	Number Outputs	1
	Event Mask	Mask of event classes to process
	Critical	Indicates whether this call must complete quickly
	Buffer	Real address of error log
	Length	Length of error log buffer
Out	Status	1: No Errors Found 0: New Error Log returned -1: Hardware Error

7–59. The *event-scan* call must fill in the error log with a single error log formatted as specified in Section 10.3.2, “RTAS Error/Event Return Format,” on page 168. If necessary, the data placed into the error log must be truncated to *length* bytes.

7–60. RTAS must only check for errors or events that are within the classes defined by the *Event mask. Event mask* is a bit mask of error and event classes. Refer to Table 50 on page 159 for the definition of the bit positions.

7–61. If *Critical* is non-zero, then RTAS must perform only those operations that are required for continued operation. No extended error information will be returned.

7–62. The *event-scan* call must return the first found error or event and clear that error or event so it is only reported once.

7–63. The operating system must continue to call *event-scan* while a status of “New Error Log returned” is returned.

7–64. The *event-scan* call must be made at least *rtas-event-scan-rate* times per minute for each error and event class and must have the *Critical* parameter equal to 0 for this periodic call.

Personal Use Copy - Not for Reproduction

Software Implementation Note: In a multiprocessor system, each processor should call *event-scan* periodically, not always the same one. The *event-scan* function needs to be called a total of *rtas-event-scan-rate* times a minute.

Software Implementation Note: The maximum size of the error log is specified in the Open Firmware Device Tree as the *rtas-error-log-max* property of the RTAS node.

Software Implementation Note: This call does not log the error in NVRAM. It returns the error log to the operating system. It is the responsibility of the operating system to take appropriate action.

Software Implementation Note: For best system performance, the requested *rtas-event-scan* rate should be as low as possible, and as a goal should not exceed 120 scans per minute. Maximum system performance is obtained when no scans are required.

7.3.4.2 Check-exception

Requirements:

7–65. RTAS must implement a *check-exception* call using the argument call buffer defined by Table 23 on page 113.

Personal Use Copy - Not for Reproduction

Table 23. *check-exception* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *check-exception*
	Number Inputs	6
	Number Outputs	1
	Vector Offset	The vector offset for the exception. See *The PowerPC Architecture* [1], Section 5.4, “Interrupt Processing”.
	Additional Information	Information which RTAS may need to determine the cause of the exception, but which may be unavailable to it in hardware registers. See Table 24 on page 113 for details.
	Event Mask	Mask of event classes to process
	Critical	Indicates whether this call must complete quickly
	Buffer	Real address of error log
	Length	Length of error log
Out	Status	1: No Errors Found 0: New Error Log returned -1: Hardware Error

7–66. The operating system must provide the value specified in Table 24 on page 113 in the “Additional Information” parameter in the call to *check-exception*.

Table 24. Additional Information Provided to *check-exception* call

Source of Interrupt	Value of “Additional Information” Variable
External Interrupt	Interrupt number
Machine Check Exception	Value of register SRR1 at entry to machine check handler
System Reset Exception	Value of register SRR1 at entry to system reset handler
Other Exception	Value of register SRR1 at entry to exception handler

Personal Use Copy - Not for Reproduction

7–67. The *check-exception* call must fill in the error log with a single error log formatted as specified in Section 10.3.2, “RTAS Error/Event Return Format,” on page 168. The data in the error log must be truncated to *length* bytes.

7–68. If *Critical* is non-zero, then RTAS must perform only those operations that are required for continued operation. No extended error information will be returned.

7–69. The *check-exception* call must return the first found error or event and clear that error or event so it is only reported once.

7–70. RTAS must only check for errors or events that are within the classes defined by the *Event mask. Event mask* is a bit mask of error and event classes. Refer to Table 50 on page 159 for the definition of the bit positions.

Software Implementation Note: All operating system reserved exception handlers should call *check-exception* to process any errors that are unknown to the operating system.

Software Implementation Note: The interrupt number for external device interrupts is provided in the Open Firmware device tree as specified in the *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10].

7.3.5 PCI Configuration Space

Device drivers and system software need access to PCI configuration space. Chapter 3, “System Address Map,” on page 23 defines system address spaces for PCI memory and PCI I/O spaces. It does not define an address space for PCI configuration. Different PCI bridges may implement the mechanisms for accessing PCI configuration space in different ways. The RTAS calls in this section provide an abstract way of reading and writing PCI configuration spaces.

The PCI Local Bus Specification, Revision 2.1 defines two addressing modes for the PCI configuration space. The PCI access functions both take a *config_addr* input parameter which is similar to a Type 1 address. This address is a 24-bit quantity composed of bus, device, function, and register numbers, all concatenated (high to low). Bus is an 8-bit quantity, device a 5-bit quantity, function a 3-bit quantity, and register an 8-bit quantity. This allows the configuration of up to 256 buses (including sub-bridges), 32 devices per bus, 8 functions per device, and 256 bytes of register space per function. Thus, up to 8192

Personal Use Copy - Not for Reproduction

devices on various buses may be addressed. The *config_addr* for a device is obtained from the Open Firmware Device Tree.

The PCI Local Bus Specification, Revision 2.1 requires that unimplemented or reserved register space read as 0's, and that reads of the Vendor ID register of devices or functions which aren't present should be unambiguously reported (reading 0xFFFF is sufficient). Writes to unimplemented or reserved register space are specified as no-ops. Writes to devices or functions which aren't present are undefined. These operations are undefined if a bus is specified which doesn't exist.

Requirements:

7–71. For the RTAS PCI configuration space functions, the parameter *config_addr* must be a configuration address as specified by the *PCI Bus Binding to IEEE 1275 Standard for Boot (Initialization, Configuration) Firmware* [12].

7–72. All RTAS PCI Read/Write functions must follow the PCI Local Bus Specification, Revision 2.1 or later.

7.3.5.1 Read-pci-config

Requirements:

7–73. RTAS must implement a *read-pci-config* call using the argument call buffer defined by Table 25 on page 115.

Table 25. *read-pci-config* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *read-pci-config*
	Number Inputs	2
	Number Outputs	2
	Config_addr	Configuration Space Address
	Size	Size of Configuration Cycle in bytes, value can be 1, 2, or 4

Personal Use Copy - Not for Reproduction

Table 25. *read-pci-config* Argument Call Buffer ***(Continued)***

Parameter Type	Name	Values
Out	Status	0: Success -1: Hardware Error
	Value	Value Read from *config_addr*

7–74. The *read-pci-config* call must return the value from the configuration register which is located at *config_addr* in PCI configuration space.

7–75. The *read-pci-config* call must perform a 1-byte, 2-byte, or 4-byte configuration space read depending on the value of the *size* input argument.

7–76. The *config_addr* must be aligned to a 2-byte boundary if *size* is 2 and to a 4-byte boundary if *size* is 4.

7–77. The *read-pci-config* call of devices or functions which are not present must return *Success* with all ones as the output *value*.

7.3.5.2 Write-pci-config

Requirements:

7–78. RTAS must implement a *write-pci-config* call using the argument call buffer defined by Table 26 on page 116.

Table 26. *write-pci-config* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *write-pci-config*
	Number Inputs	3
	Number Outputs	1
	Config_addr	Configuration Space Address
	Size	Size of Configuration Cycle in bytes, can be 1, 2, or 4
	Value	Value to be written to *config_addr*

Personal Use Copy - Not for Reproduction

Table 26. *write-pci-config* Argument Call Buffer ***(Continued)***

Parameter Type	Name	Values
Out	Status	0: Success -1: Hardware Error

7–79. The *write-pci-config* call must store the value from the configuration register which is located at *config_addr* in PCI configuration space.

7–80. The *write-pci-config* call must perform a 1-byte, 2-byte, or 4-byte configuration space write depending on the value of the *size* input argument.

7–81. The *config_addr* must be aligned to a 2-byte boundary if *size* is 2 and to a 4-byte boundary if *size* is 4.

7–82. The *write-pci-config* call of devices or functions which aren't present must be ignored. A status of *Success* must be returned.

7.3.6 Operator Interfaces and Platform Control

The RTAS operator interface and platform control functions provide an OS with the ability to perform platform services in a portable manner. The RTAS operator interface provides the ability for an OS to notify the user about OS events during boot, to notify the user of abnormal events, and to obtain information from the platform. The platform control functions give the OS the ability to obtain platform-specific information and to control platform features.

These calls are all "best effort" calls. RTAS should make its best effort to implement the intent of the call. If the Platform Hardware does not implement some optional feature, it is permitted for RTAS to either return an error, or to virtualize the service in some way and return "Operation Succeeded."

Software Implementation Note: For example, a keyswitch could be virtualized by storing a keyswitch value in NVRAM and by providing a user interface to modify this value. The RTAS call *get-sensor-state* on the keyswitch would just return the value stored in NVRAM.

Software Implementation Note: If these services are only called prior to the use of any of the underlying devices by the OS, for example, during boot time, or only after the OS has finished using the devices, for example, during a crash, then the OS can avoid mutual exclusion and sharing concerns. Otherwise, synchronization per Section 7.2.3, "RTAS Critical Regions," on page 95, must be performed.

Personal Use Copy - Not for Reproduction

7.3.6.1 Display-character

The RTAS call *display-character* can be used by an operating system to display informative messages during boot, or to display error messages when an error has occurred and the operating system can not depend on its display drivers. This call is intended to display the characters on an LCD panel, graphics console, or attached tty. The precise implementation is platform vendor specific.

Requirements:

7–83. RTAS must implement a *display-character* call using the argument call buffer defined by Table 27 on page 118 to place a character on the output device.

7–84. The operating system must serialize all calls to *display-character* with any other use of the *rtas-display-device*.

Table 27. *display-character* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *display-character*
	Number Inputs	1
	Number Outputs	1
	Value	Character to be displayed
Out	Status	0: Success -1: Hardware error -2: Device busy, try again later

7–85. If a physical output device is used for the output of the RTAS *display-character* call, then it must have at least one line and 4 characters.

7–86. Certain ASCII control characters must have their normal meanings with respect to position on output devices which are capable of cursor positioning. In particular, ^M (0x0D) must position the cursor at column 0 in the current line, and ^J (0x0A) must move the cursor to the next line, scrolling old data off the top of the screen.

7–87. RTAS must not output characters to the *rtas-display-device* except for explicit calls to the *display-character* function.

Personal Use Copy - Not for Reproduction

Software Implementation Note: RTAS should try to produce output to the user. This could be to the system console, to an attached terminal, or to some other device. It could be implemented using a diagnostic processor or network. RTAS could also implement this call by storing the messages in a buffer in NVRAM so the user could determine the reason for a crash upon re-boot.

Software Implementation Note: This call will modify the registers associated with the *rtas-display-device*. The operating system may also access this device but must be aware that calls to *display-character* will change the state of the device.

7.3.6.2 Set-indicator

The RTAS *set-indicator* function provides the operating system with an abstraction for controlling various lights, indicators, and other resources on a hardware platform. If multiple indicators of a given type are provided by the platform, this function permits addressing them individually.

Requirements:

7–88. RTAS must implement a *set-indicator* call which sets the value of the indicator of type *Indicator* and index *Indicator-index* using the argument call buffer defined by Table 28 on page 119 and indicator types defined by Table 29 on page 120.

Table 28. *set-indicator* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *set-indicator*
	Number Inputs	3
	Number Outputs	1
	Indicator	Token defining the indicator
	Indicator-index	Index of specific indicator (0, 1, ...)
	State	Desired new state
Out	Status	0: Success -1: Hardware Error -2: Hardware busy, try again later -3: No such indicator implemented

Personal Use Copy - Not for Reproduction

Table 29. Defined Indicators

Indicator Name	Token Value	Defined Values	Default Value	Examples/Comments
Tone Frequency	1	Unsigned Integer (units are Hz)	1000	Generate an audible tone using the tone generator hardware (see Table 2 on page 19). RTAS will select the closest implemented audible frequency to the requested value.
Tone Volume	2	0-100 (units are percent), 0 = OFF	0	Set the percentage of full volume of the tone generator output, scaled approximately logarithmically.
System Power State	3	Off(0), On(1), Standby(2), Suspend(3), Hibernate(4)	On	An external indicator which may be provided to indicate the system power state. Colors and/or blinking may be used to indicate power states other than Off and On.
Warning Light	4	Off(0) Green(1) Amber(2) Red (3)	Off	Example Operating system policy: Green indicates no problems, amber indicates a near-abnormal voltage, temperature, or other environmental condition, red indicates an out-of-range condition which must be acted upon. Software should display the condition to the user before displaying amber or red.
Disk Activity Light	5	On (1), Off (0)	Off	Software may turn this light on when one or more disk transactions are in progress on any attached drive.
Hexadecimal Display Unit (LED)	6	0x0-0xFFFF: 4 Hex Digits to display 0xFFFFFFFF: Clear Display	Clear	Can be used to indicate boot progress, or fatal error indications when no other I/O device can be trusted. Values in the range 0xE000…0xEFFF are reserved for firmware and POST usage, and must be documented by the platform vendor if used. Other values are OS-specific, and must documented in OS documentation if used.
Battery Warning Time	7	Minutes Zero minutes = disabled	Zero	Sets the Battery Warning threshold. A Battery Warning event will be sent when approximately this many minutes of battery lifetime remains. The timing is more accurate with smaller parameter values. See Table 62 on page 199 and Section 11.2.3.5, “Battery-Related RTAS Calls,” on page 208 for further detail.
Condition Cycle Request	8	Disable (0) Enable (1)	Not in Progress	Setting this indicator requests the power circuitry to start a Battery Condition cycle. See Section 11.2.3.5, “Battery-Related RTAS Calls,” on page 208 for further detail.
Vendor Specific	9000-9999			Indicator values reserved for platform vendor use.

7.3.6.3 Get-sensor-state

The RTAS call *get-sensor-state* can be used by a shrink wrapped operating system to read the current state of various sensors on any Platform. If multiple

Personal Use Copy - Not for Reproduction

sensors of a given type are provided by the platform, this function permits addressing them individually.

Requirements:

7–89. RTAS must implement a *get-sensor-state* call which reads the value of the sensor of type *Sensor* which has index *Sensor-index* using the argument call buffer defined by Table 30 on page 121 and the sensor types defined by Table 31 on page 121.

Table 30. *get-sensor-state* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for get-sensor-state
	Number Inputs	2
	Number Outputs	2
	Sensor	Token defining the sensor type
	Sensor-index	Index of specific sensor (0, 1, ...)
Out	Status	0: Success -1: Hardware Error -2: Hardware Busy, Try again later -3: No such sensor implemented
	State	Current sensor's state

Table 31. Defined Sensors

Sensor Name	Token Value	Defined Values	Description
Key Switch	1	Off (0), Normal (1), Secure (2), Maintenance (3)	Key switch modes are tied to OS security policy. Suggested meanings: Maintenance mode permits booting from floppy or other external, non-secure media. Normal mode permits boot from any attached device. Secure mode permits no manual choice of boot device, and may restrict available functionality which can be accessed from the main operator station. Off completely disables the system.

Personal Use Copy - Not for Reproduction

Table 31. Defined Sensors ***(Continued)***

Sensor Name	Token Value	Defined Values	Description
Enclosure Switch	2	Open (1), Closed (0)	This is a switch which indicates that the computer enclosure is open. For safety reasons, opening the enclosure should generally cause power to be removed.
Thermal Sensor	3	Temperature (in Degrees Celsius)	If implemented, returns the internal temperature of the most temperature-sensitive portion of platform.
Lid Status	4	Open(1), Closed(2)	If implemented, permits a power managed OS to determine that a computer is not going to be used soon, and can be put into a reduced-power state
Power Source	5	AC (0), Battery (1) AC & Battery(2)	Indicates source of primary power.
Battery Voltage	6	Voltage (in units of Volts)	Current battery output voltage.
Battery Capacity Remaining	7	High(3) Mid(2) Low(1) Very Low(0)	Used mainly for batteries which do not provide information about their current state. The actual value may depend on the implementation, including battery chemistry.
Battery Capacity Percentage	8	0-1000 (in units of 0.1%)	Used mainly for batteries which do report their current state of charge.
Environmental and Power State (EPOW) Sensor	9	EPOW_Reset(0) Warn_Cooling(1) Warn_Power(2) System_Shutdown(3) System_Halt(4) EPOW_Main_Enclosure(5) EPOW_Power_Off(7)	RTAS assessment of the environment and power state of the platform. Refer to Section 11.2.1.2, “Definition of RTAS Abstracted Power Management Event Types,” on page 199 for further information.
Battery Condition Cycle State	10	None (0) In Progress (1) Requested (2)	Current state of the Battery Condition Cycle hardware. See Section 11.2.3.5, “Battery-Related RTAS Calls,” on page 208 for further information.
Battery Charging State	11	Charging (0) Discharging (1) No Current Flow (2)	Indicates whether the battery is currently being charged, being used as a power source (discharging), or neither.
Vendor Specific	9000-9999		Reserved for use by platform vendors.

Personal Use Copy - Not for Reproduction

7.3.7 Power Management

The ability to manage power consumption is a requirement for several conforming systems, such as laptop computers and "Green" or Energy Star computers. RTAS provides the mechanisms necessary for an operating system to manage the power consumption of the hardware. All policy decisions are up to the operating system.

Power management may be applied to an arbitrary power management domain. Power domains for a given platform are defined in the Open Firmware Device Tree. See Chapter 11, "Power Management," on page 185 and *PowerPC Microprocessor Common Hardware Reference Platform System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware* [10].

Power management will involve changing clock speeds, lowering or removing power from devices, and setting certain control bits inside of processors. The RTAS power management calls only apply to power management services *outside* the processors. It is the responsibility of the operating system to manage individual CPUs.

Although a fundamental principle of power management is that system software as opposed to platform hardware/firmware provides all power management policy, there are times during the boot life of an operating system when system software is unable to provide this control (for example, prior to the loading of the power management system software). During these times the platform must assume responsibility for making certain policy decisions. The *assume-power-management* and *relinquish-power-management* RTAS calls provide a mechanism for handing responsibility back and forth between system software and the platform. Refer to the Chapter 11, "Power Management," on page 185 for a discussion of how these calls are intended to be utilized in a power managed system.

Requirements:

7–90. For the Power Management option: The operating system must control the power management features of the processors and any attached power-manageable devices for which it has device drivers.

Software Implementation Note: The RTAS Power Management calls provide a mechanism, not a policy. It is the responsibility of the operating system to implement the Power Management policy. It is also the responsibility of the operating system to maintain any necessary state information.

Personal Use Copy - Not for Reproduction

7.3.7.1 Set-power-level

Requirements:

7–91. For the Power Management option: RTAS must implement a *set-power-level* call which changes the power level setting of a power domain. This call must be implemented using the argument call buffer defined by Table 32 on page 124.

Table 32. *set-power-level* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *set-power-level*
	Number Inputs	2
	Number Outputs	2
	Power_domain	Token defining the power domain
	Level	Token for the desired level for this domain
Out	Status	0: Success -1: Hardware Error -2: Busy, Try again later
	Actual_level	The power level actually set

7–92. For the Power Management option: *Power_domain* must be a power domain identified in the Open Firmware Device Tree.

7–93. For the Power Management option: *Level* must be a power level as specified by Section 11.2.1.1, “Definition of Domain Power Levels,” on page 198.

7–94. For the Power Management option: The *set-power-level* call must set the level of the specified domain to the power level specified by *level* or to the next higher implemented level.

7–95. For the Power Management option: The *set-power-level* call must return the power level actually set in the *Actual_level* output parameter.

Software Implementation Note: The *set-power-level*(0,0) call, if implemented, removes power from the root domain, turning off power to all domains. The external events which can turn power back on are

Personal Use Copy - Not for Reproduction

platform specific. The RTAS primitive *power-off* also removes power from the system, but permits specifying the events which can turn power back on.

Software Implementation Note: The implemented values for the Level parameter for each power domain are defined in the Open Firmware device tree.

7.3.7.2 Get-power-level

Requirements:

7–96. For the Power Management option: RTAS must implement a *get-power-level* call which returns the current setting of a power domain. This call must be implemented using the argument call buffer defined by Table 33 on page 125.

Table 33. *get-power-level* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *get-power-level*
	Number Inputs	1
	Number Outputs	2
	Power_domain	Token defining the power domain
Out	Status	0: Success -1: Hardware Error -2: Busy, try again later -3: Can't determine current level
	Level	The current power level for this domain

7–97. For the Power Management option: *Power_domain* must be a power domain identified in the Open Firmware Device Tree.

Software Implementation Note: The *get-power-level* call only returns information about power levels whose state is readable in hardware. It does not need to remember the last set state and return that value.

Personal Use Copy - Not for Reproduction

7.3.7.3 Assume-power-management

This call transfers control of the power management policy from the platform to system software.

Requirements:

7–98. For the Power Management option: RTAS must implement an *assume-power-management* call which transfers control of the power management policy from the platform to system software, using the argument call buffer specified in Table 34 on page 126.

Table 34. *assume-power-management* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *assume-power-management*
	Number Inputs	0
	Number Outputs	1
Out	Status	0: Success -1: Hardware Error

Hardware Implementation Note: Even after the successful completion of the *assume-power-management* call, platform hardware/firmware may take actions (including turning off platform power) without the direction of system software to provide safety or prevent the destruction of platform hardware. In these cases, the platform should make an effort to inform system software of its imminent intervention.

7.3.7.4 Relinquish-power-management

This call transfers control of the power management policy from system software to the platform.

Requirements:

7–99. For the Power Management option: RTAS must implement a *relinquish-power-management* call which transfers control of the power management policy from system software to the platform, using the argument call buffer specified in Table 34 on page 126.

Personal Use Copy - Not for Reproduction

7–100. For the Power Management option: If the power management policy is under control of the operating system, the *relinquish-power-management* call must be performed prior to the completion of the transition into the Suspend, Hibernate, or Off system power states.

Table 35. *relinquish-power-management* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *relinquish-power-management*
	Number Inputs	0
	Number Outputs	1
Out	Status	0: Success -1: Hardware Error

7.3.7.5 Power-off

This primitive turns power off on a system which is equipped to perform a software-controlled power off function. It may be implemented on platforms which do not implement any other Power Management functions.

Requirements:

7–101. If software controlled power-off hardware is present: The *power-off* function must turn off power to the platform, using the argument call buffer described in Table 36 on page 128.

7–102. If software controlled power-off hardware is present: *Power_on_mask*, which is passed in two parts to permit a possible 64 events even on 32-bit implementations, must be a bit mask of power management events, refer to requirement 11–3. If a bit in the *resume_mask* is set to 1, then the hardware should enable the corresponding hardware power-on mechanism.

Personal Use Copy - Not for Reproduction

Table 36. *power-off* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *power-off*
	Number Inputs	2
	Number Outputs	1
	Power_on_mask_hi	Mask of events that can cause a power on event - event mask values [0:31] (right-justified if the cell size is 64 bits)
	Power_on_mask_lo	Mask of events that can cause a power on event - event mask values [32:63] (right-justified if the cell size is 64 bits)
Out	Status	On successful operation, does not return -1: Hardware error

7.3.8 Suspend and Hibernate

Suspension is the process of placing the memory subsystem in a low power data retentive state and placing the processor in a powered off state. When power is restored to the processor, the firmware recognizes that the system was in a suspend state. The firmware re-initializes all devices and transfers control back to the operating system.

Hibernation is the process of saving a system image on disk and powering down the system. When power is reapplied, the system firmware reboots the system. Operating system boot or initialization code recognizes that the system was hibernating, restores the saved image, and returns control to the point of hibernation. Firmware is not involved in wakeup from hibernation, other than its usual role in rebooting the system.

For more details on power management refer to Chapter 11, “Power Management,” on page 185.

Software Implementation Note: Hibernation can be performed without the use of the RTAS *hibernate* primitive, but the primitive can be valuable for saving the last part of the OS if the OS cannot save itself without corrupting state information that is to be part of the saved image.

Personal Use Copy - Not for Reproduction

7.3.8.1 Suspend

The RTAS function *suspend* preserves the System Memory image exclusive of that devoted to firmware use, records suspend restoration parameters in the system's configuration memory, and sets the system power state to suspend in a system dependent manner (this usually means powering down everything except the memory sub-system and its refresh, and some mechanism to trigger the resume). When the system is subsequently powered up, the firmware reads the suspend restoration parameters, resets its devices, and returns to the caller of suspend.

Requirements:

7–103. The RTAS *suspend* function must implement the power-saving Suspend state using the argument call buffer defined in Table 37 on page 129.

Table 37. *suspend* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *suspend*
	Number Inputs	3
	Number Outputs	2
	Resume_mask_hi	Mask of events that can cause a resume event - event mask values [0:31] (right-justified if the cell size is 64 bits)
	Resume_mask_lo	Mask of events that can cause a resume event - event mask values [32:63] (right-justified if the cell size is 64 bits)
	Processor_number	Token defining the processor to be awakened on a resume event. This is the *reg* value for the processor, as provided in the Open Firmware device tree
Out	Status	0: Success -1: Hardware error
	Resume_event	Event that cause the resume

7–104. The RTAS *suspend* call must be made only from the processor identified by *processor_number*.

Personal Use Copy - Not for Reproduction

7–105. *Resume_mask*, which is passed in two parts to permit a possible 64 events even on 32-bit implementations, must be a bit mask of power management events, refer to requirement 11–3. If a bit in the *resume_mask* is set to 1, then the hardware should enable the corresponding hardware wakeup mechanism.

7–106. *Resume_event* must be a power management event that caused the resume.

7–107. In an SMP system, the operating system must have invoked the RTAS *stop-self* function on all other processors prior to invoking *suspend.*

7–108. All elements of the I/O sub-system must be in a quiescent state at the time of the call: they must not be transferring data to or from memory, nor be able to cause an interrupt to any processor.

7.3.8.1.1 Suspend Restoration (Resume)

Upon receiving a wakeup event that was enabled using *resume_mask*, the system will resume execution on the processor identified by *processor_number.*

Requirements:

7–109. Upon return from the RTAS *suspend* call, the state of the memory locations, exclusive of the first 256 bytes of real memory and the RTAS private data area, must be in the same state they were in at the time of the call to *suspend***.**

7–110. Upon return from *suspend*, execution must be on the processor indicated by *processor_number.*

7–111. Upon return from *suspend*, all processors except the processor identified by *processor_number* must be in the *stopped* state.

7–112. Upon return from *suspend,* the firmware must restore the registers and devices which are reserved for firmware to the same state as before the *suspend*.

7–113. Upon return from *suspend,* the firmware must place all elements of the I/O system into a safe state.

7–114. The return from *suspend* must restore the registers listed in requirement 7–10 to the same state that existed prior to the *suspend* call.

Software Implementation Note: RTAS need not reprobe the I/O subsystem on resume, and in fact should avoid doing so in order to make resume

Personal Use Copy - Not for Reproduction

quicker. Firmware is NOT required to restore the I/O subsystem to the previous state; this is the responsibility of the operating system.

Software Implementation Note: This call may violate requirements stated in requirement 7–10 in that it actually saves and restores the values.

Software Implementation Note: The first 256 bytes of real memory are available for firmware use during the suspend/resume operations.

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

Software Implementation Note: RTAS is not required to permit configuration changes of devices which are reserved for firmware use while in a Suspend state. If such devices are removed or modified while suspended, the results are unspecified.

Software Implementation Note: Changing the hardware configuration while in a Suspend state may cause a Configuration Change event as defined in Table 62 on page 199 if the platform provides this capability.

Software Implementation Note: Since the Open Firmware device tree is not updated in a Suspend/Resume sequence, it is OS dependent whether an OS supports platform hardware configuration changes while in the Suspend state.

7.3.8.1.2 Hibernate

The RTAS *hibernate* call provides the capability to save the current system image for later restoration. One way that this could be implemented (using Open Firmware, with minimal support in RTAS) is to save the address of the memory list, to reset the I/O subsystem (avoiding resetting the memory subsystem, as this may disrupt memory contents), and then to re-enter Open Firmware. Open Firmware can then recognize the block list, and instead of performing a normal boot, can use its device drivers to write the data to the specified device. Once the data is recorded, Open Firmware causes the system to enter a low power hibernate state. Other implementations which meet the Requirements are also possible.

Requirements:

7–115. The RTAS *hibernate* call must be implemented using the argument call buffer described by Table 38 on page 132.

Personal Use Copy - Not for Reproduction

Table 38. *hibernate* Argument Call Buffer

Parameter Type	Name	Values
In	Token	Token for *hibernate*
	Number Inputs	3
	Number Outputs	1
	Block_list	A real pointer to a block list
	Wakeup_mask_hi	Mask of events that can cause a wakeup event - event mask values [0:31] (right-justified if the cell size is 64 bits)
	Wakeup_mask_lo	Mask of events that can cause a wakeup event - event mask values [32:63] (right-justified if the cell size is 64 bits)
Out	Status	On successful operation, does not return -1: Hardware error

7–116. The *wakeup_mask*, which is passed in two parts to permit a possible 64 events even on 32-bit implementations, must be a bit mask of power management events, refer to requirement 11–3. If a bit in the wakeup_mask is set to 1, then the hardware should enable the corresponding hardware wakeup mechanism.

7–117. The area of memory described by the *hibernate block_list* must be written to the device(s) as indicated.

7–118. The hibernate *block_list* must start with a list length in bytes and then must have quadruples of real address, length of memory block, device id, and block number as shown in Table 39 on page 132. Additional requirements on the block list are:

a. The block list must be a sequence of cells as defined in requirements 7–34 and 7–35.

b. Block numbers are in 512 byte units.

Table 39. Format of Block List

Length of list in bytes
Address of memory area 1

Personal Use Copy - Not for Reproduction