Advanced RISC Computing Specification

Version 1.2

© 1991, 1992 MIPS Technology Inc.—Printed in the United States of America.
2071 North Shoreline Blvd., Mountain View, California 94039-7311 U.S.A.

All rights reserved. This product and related documentation is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of MIPS and its licensors, if any.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States government is subject to restrictions as set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

THE COPYRIGHT HOLDER DISCLAIMS ALL WARRANTIES OF MERCHANTABILITY AND FITNESS FOR USE, AND FURTHER DISCLAIMS ANY AND ALL DAMAGES ARISING FROM USE OF THIS SPECIFICATION INCLUDING BUT NOT LIMITED TO DIRECT, INDIRECT, CONSEQUENTIAL AND SPECIAL DAMAGES WHICH MAY ARISE FROM ANY USE OF THIS SPECIFICATION. THE USER OF THIS SPECIFICATION HEREBY AGREES TO HOLD THE COPYRIGHT HOLDER, ITS AGENTS, PREDECESSORS, ASSIGNS, AND SUCCESSORS HARMLESS FROM ANY DAMAGES, HOWEVER DENOMINATED, WHICH MAY ARISE FROM ITS USE OF THIS SPECIFICATION.

THE TECHNICAL MATERIALS IN THIS SPECIFICATION MAY BE COVERED BY ONE OR MORE PATENTS. ACCESS TO THIS SPECIFICATION DOES NOT DIRECTLY, INDIRECTLY, OR BY IMPLICATION LICENSE ANY PATENTS WHICH COVER THE MATERIALS SET FORTH HEREIN. POSSESSION OF THIS SPECIFICATION DOES NOT AUTHORIZE USE OF THIS SPECIFICATION.

THIS SPECIFICATION IS SUBJECT TO CHANGE WITHOUT NOTICE. NO REPRESENTATIONS, EXPRESS OR IMPLIED, BY IMPLICATION, ESTOPPEL OR OTHERWISE RESTRICT THE RIGHT TO CHANGE OR REVISE THIS SPECIFICATION.

USER’S USE OF ANY PORTION OF THIS SPECIFICATION SHALL BE DEEMED TO BE AN ACCEPTANCE OF THE ABOVE DISCLAIMERS AND CONDITIONS OF USE.

TRADEMARKS
All other product names mentioned herein are the trademarks of their respective owners.

Table of Contents

Preface *11*

1. Introduction to the ARC Specification **14**

1.1 Covered By the ARC Specification 15
1.2 Covered By Addenda To The ARC Specification 16
1.3 Not Specified by the ARC Specification 16
1.4 Conventions Used In the ARC Specification 16
1.5 Conformance 17

Part 1: Base Specification

2. System Architecture **19**

2.1 Architectural Working Statement 19
2.2 System States 20
2.2.1 State Diagram 20
2.3 Software Subsystems 22
2.3.1 Application Software 23
2.3.2 Operating System Software 23
2.3.3 Hardware Abstraction Layer (HAL) Software 23
2.3.4 Device Drivers 24
2.3.5 Loader, Installer, and Independent Utility Software 24
2.4 Hardware Subsystems 25
2.4.1 Processing Subsystems 25
2.4.2 Peripheral Attachment Subsystems 25
2.5 Firmware 26
2.6 System Interface Definitions 26

3. Platform Hardware 27
3.1 System Configurations 27
3.2 Server System Configuration 28
Consequences of Non-Compliance 29
3.2.1 Processor Unit 30
3.2.2 Floating Point Unit 31
3.2.3 Cache 31
3.2.4 Memory 31
3.2.5 Timing Function Support 31
3.2.6 Real Time Clock 32
3.2.7 System Timer 32
3.2.8 Console 32
3.2.9 CD-ROM 32
3.3 Desktop System Configuration 32
3.3.1 Keyboard 32
Requirements 33
3.3.2 Pointing Device 33
Requirements 33
3.3.3 Video Subsystem 34
Requirements 34
3.3.4 Audio 35
3.4 Optional Hardware 37
3.4.1 Floppy Drive 37
3.4.2 Serial Ports 37
3.4.3 Parallel Port 38
3.4.4 SCSI Interface 39
3.4.5 Network Interface 40
Ethernet 40
Token Ring 40
3.5 Additional Hardware 41
3.6 Media Formats 41
3.6.1 Media Formats for System Load 42
3.6.2 System Partition Formats 42
3.6.3 Diskettes (5 1/4-inch and 3 1/2 inch) 42
3.6.4 CD-ROM 43
3.6.5 Disk Storage Devices 43
3.6.6 Network 43
3.6.7 Data Interchange 44
3.7 Processing Subsystem 44
3.7.1 Related Consequences 45
3.8 Peripheral Attachment Subsystems (I/O Bus) 46
3.8.1 Requirements 46
3.8.2 Related Consequences 46

4. Platform Firmware 47
4.1 Firmware Conventions 47
4.1.1 Calling Procedures 47
Parameter Passing 48
Status Codes 48
4.1.2 Memory Utilization 49
4.1.3 Stack and Data Addressability 50
4.1.4 Object Formats 50
4.2 The Firmware Environment 51
4.2.1 Exception Block 51
4.2.2 System Parameter Block 51
4.2.3 Restart Block 53
Restart Procedure 54
4.2.4 Environment Variables 55
Console Initialization Environment Variables 55
Software Loading Environment Variables 56
Time Zone Environment Variable 57
Firmware Search Path Environment Variable 57
4.2.5 System Configuration Data 57
Component Class and Type 59
Component Flags 63
Component Version and Revision 64
Component Key 64
Affinity Mask 65
Configuration Data Size 66
Component Identifier 66
System Topology Constraints 67
4.2.6 Additional Configuration Data 67
4.2.7 Devices, Partitions, Files, and Path Specifications 71
Path Specifications 72
4.2.8 System Partition 73
4.3 Standard Firmware Functions 75
4.3.1 Program Loading 75
4.3.2 Program Termination 78
4.3.3 Configuration Functions 79
4.3.4 Input/Output Functions 82
4.3.5 Environment Functions 91
4.3.6 Miscellaneous Functions 92
4.3.7 The Firmware Function Vector 96
4.3.8 Platform-Specific Firmware Functions 97
4.3.9 Adapter-Specific Firmware Functions 97
4.4 Loaded-Program Conventions 97
4.5 Interrupts and Exceptions 100
Invoking Exception Handlers 100
Exception Handler Routines 101
Loaded Program Access to Exceptions 101

5. System Console 102
5.1 Functionality 102
5.1.1 Basic Console Input 102
5.1.2 UNICODE Console Input 104
5.1.3 Basic Console Output 104
5.1.4 UNICODE Console Output 107
5.2 Operational Characteristics 108

6. Multiprocessor Platforms 109
6.1 MP Architecture Overview 109
6.2 Processor Subsystem and Caches 110
6.2.1 Processor Instruction set 110
6.2.2 User Application Portability Considerations 110
6.2.3 Symmetry and Shared Memory 110
6.2.4 Homogeneity of CPUs 110
6.2.5 Hardware-Enforced Cache Coherency 111
6.2.6 Cache Coherency During I/O Transfers 111
6.2.7 Atomic Writes 111
6.2.8 Strong Ordering 111
6.2.9 Processor Identification 111
6.2.10 Timer Interrupts 112
6.2.11 Optional Powerfail Interrupt 112
6.3 I/O Subsystem 112
6.3.1 Symmetry 112
6.4 Interprocessor and I/O interrupts 112
6.4.1 Interprocessor Interrupts 112
6.4.2 Interprocessor Interrupt Priority 113
6.4.3 I/O interrupt Assignment 113
6.5 Boot and Reset functions 113
6.5.1 Boot Master CPU 113
6.5.2 Starting CPUs 113
6.5.3 Program Termination Function Semantics for MP Machines 114

Part 2: Developing Material

7. Network Bootstrap Protocol Mappings 116
7.1 BOOTP/TFTP/UDP/IP/ARP Protocols 116
7.2 Networked System Partition 117
7.2.1 BOOTP/TFTP Protocol References 117
7.2.2 System Interface Mapping 118
Open() 118
Read() 119
Write() 119
Close() 119
GetReadStatus() 119

Mount() ... 120
Seek() ... 120
GetDirectoryEntry() ... 120
7.2.3 Protocol Clarifications ... 121
Token-Ring MAC Requirements ... 121
Ethernet MAC Requirements ... 121
LLC Requirements ... 121
BOOTP Request Frame Requirements ... 122
BOOTP Response Frame Requirements ... 122
TFTP RRQ Frame Requirements ... 122
TFTP ERROR Frame Requirements ... 123
ICMP Frame Requirements ... 123
ARP Frame Requirements ... 123
7.2.4 Server Considerations ... 124
BOOTP vs. RARP ... 124
LLC Support ... 124
Filename Support ... 124
7.3 IBM DLC RIPL/LLC Protocols ... 124
7.3.1 Protocol References ... 125
7.3.2 System Interface Mapping ... 125
Open() ... 125
Read() ... 126
Write() ... 126
Close() ... 126
GetReadStatus() ... 126
Mount() ... 127
Seek() ... 127
7.3.3 Protocol Clarifications ... 127
Token-Ring MAC Requirements ... 128
Ethernet MAC Requirements ... 128
FIND Frame Requirements ... 128
FOUND Frame Requirements ... 130
SEND.FILE.REQUEST Frame Requirements ... 130
FILE.DATA.RESPONSE Frame Requirements ... 132
LOAD.ERROR Frame Requirements ... 132
PROGRAM.ALERT Frame Requirements ... 132
7.3.4 Server Considerations ... 133

Glossary ... 134

Figures

Figure 2-1 System States 20
Figure 2-2 Firmware State 21
Figure 2-3 Program State 22
Figure 3-1 6 Position DIN connector for EISA Keyboard/Mouse Connectors 34
Figure 3-2 15-Pin D_SUB for EISA Video Connector 34
Figure 3-3 Combination Receptacle for EISA Video Connector 34
Figure 3-4 9-Pin D-SUB for Asynchronous Serial 38
Figure 3-5 25-Pin D-SUB for Parallel Connector 39
Figure 3-6 50-Pin Single Ended SCSI Connector 39
Figure 3-7 Ethernet BNC Jack 40
Figure 3-8 RJ-45 Connector 40
Figure 3-9 15-Pin D-SUB receptacle for Ethernet AUI Connector........................40
Figure 3-10 9-Pin D-Sub plug connector for Token Ring.......................................41
Figure 4-1 System Parameter Block Structure 51
Figure 4-2 RestartBlock 53
Figure 4-3 Boot Status Bits...54
Figure 4-4 Example System 58
Figure 4-5 COMPONENT Data Structure 59

Tables

Table 3-1 Keyboard Standard....33
Table 3-2 Pointing Device Characteristics....33
Table 3-3 Video Characteristics....34
Table 3-4 ARC System Audio Requirements....36
Table 3-5 Audio Interface Electrical Requirements....36
Table 3-6 Recommended Audio Connectors....36
Table 3-7 Serial Interface Characteristics....38
Table 3-8 Parallel Interface Characteristics....38
Table 3-9 SCSI Interface Characteristics....39
Table 3-10 Possible Ethernet Media Connectors....40
Table 4-1 Status Codes....48
Table 4-2 Component Flag Bit Usage....64
Table 4-3 Path Mnemonics....72
Table 4-4 Firmware Vector....96
Table 5-1 Function Key Control Sequences....103
Table 5-2 Control Sequences....105
Table 5-3 Additional SGR Control Sequences....106
Table 5-4 Single Character Control Functions....107

Code Samples

Code Example 4-1 Exception Handling 100
Code Example 4-2 Restoring Registers..101

Preface

The Advanced RISC Computing Specification has been developed to define a set of standard capabilities for MIPS based computing systems. These capabilities have been chosen to allow significant opportunities for innovation by platform developers while at the same time presenting a standard environment for operating system and application software.

The goal of all participants has been to promote the development of a new class of computing systems. They provide for the vendor innovation that is usually characteristic of only proprietary systems, along with the ubiquity of systems typically based on rigid hardware standards such as the PC.

This specification was developed as a group effort by members of the Advanced Computing Environment (ACE) initiative. Early in the process of the development of this specification, representatives from several companies worked closely together to write the original drafts of the specification.

The companies involved in writing the original draft of this specification were Compaq Computer Corporation, Silicon Graphics Computer Systems, The Santa Cruz Operation, MIPS Computer Systems Inc., Digital Equipment Corporation, and Microsoft Corporation. The initial drafts of this specification were distributed to all members of the ACE initiative for review, and the feedback from many reviewers was incorporated.

How This Book Is Organized

This book is organized in the following fashion:

Chapter 1, "Introduction to the ARC Specification," is an overview of the specification itself, describing what it does and does not cover.

Chapter 2, "System Architecture," describes the software, hardware, and firmware layers.

Chapter 3, "Platform Hardware," defines what standard and optional hardware make up a system, the technical specifications for these items, and specific configurations defined by the specification.

Chapter 4, "Platform Firmware," describes the conventions, environment, and functions performed by the firmware.

Chapter 5, "System Console," describes the feature set of the console.

Chapter 6, "Multiprocessor Platforms," describes hardware and firmware changes to support multiprocessors.

Chapter 7, "Network Bootstrap Protocol Mappings," describes the mapping of network protocols onto ARC I/O functions.

Glossary is a list of words and phrases found in this book and their definitions.

Cited References

The following documents are referenced in the ARC Specification:

- *ANSI X3.64 - 1979*. Additional Control for Use with American National Standard Code for Information Interchange. Issuing Organization ANSI - American National Standards Institute.
- *ANSI X3.131-1990 (Revision 10c).* Information Systems - Small Computer System Interface (SCSI) Document Number: X3.131, Issuing Organization ANSI - American National Standards Institute.
- *EIA 232 - 1990.* Electrical Industries Association.
- *IEEE 802.3-90.* Information Processing Systems - Local Area Networks - Part 3: Carrier Sense Mult. Access with Collision Detection (CSMA/CD) Access Method and Phys. Layer Spec (IEEE Computer Society Doc.) Second Edition (ISO 8802 .3 1990). Correction Sheet - 1990, Supp. 802.3H - 1990, Supp. 802.3I - 1990.
- *IEEE 802.4-90.* Information Processing Systems - Local Area Networks - Part 4: Token-Passing Bus Access Method and Physical Layer Specifications First Edition (IEEE Computer Society Document).

- *IEEE 802.5-89.* Standards for Local Area Networks Token Ring Access Method and Physical Layer Specifications (IEEE Computer Society Document).
- *ISO/DP 6429.* ISO 7-Bit and 8-Bit coded character sets - Control Functions.
- *ISO/IEC 9945-1 1990.* Information Technology - Portable Operating System Interface (POSIX)-Part I: System Application Program Interface (API) (C Language).
- *MIPS Assembly Language Programmer's Guide*, Chapter 9, "Object File Format." Document No. ASM-01
- *MIPS R-Series Architecture Reference Manual*.
- *Microsoft MS-DOS Programmer's Reference, Version 5*, Microsoft Press, Document No. SY0766b-R50-0691, Library of Congress No. QA76.76.O63M745 1991
- See Chapter 7, "Network Bootstrap Protocol Mappings," for more references to various network protocol specifications.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table PR-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	On-screen computer output and sample code.	`\typedef enum{`
AaBbCc123	The names of commands and functions.	The Execute function
AaBbCc123	Command-line placeholder, to be replaced with a real name or value; book titles, new words or terms, or words to be emphasized	The configuration information is in *filename*. Read Chapter 6 in *User's Guide*. These are called *class* options.

Introduction to the Arc Specification 1

The Advanced RISC Computing Specification (ARC Spec) defines the architecture for an industry standard computing platform based on the MIPS family of microprocessors. This document defines system hardware and firmware components, functions, interfaces, and data formats. The following applies to all products developed to this specification:

- Applications developed to this standard (and based upon a particular operating environment) execute correctly across all compliant system implementations that execute the required operating environment.
- Application data are shareable and exchangeable among all compliant systems for media defined within the standard. At the level of standardization addressed by this specification, data exchange implies compatibility of physical media and low-level data formatting (for example, partitioning and sector formatting) across compliant systems.
- Media and devices that may be used across systems as system load (boot) sources perform correctly across all compliant systems.
- Operating environments (such as Windows NT or UNIX) developed to the interfaces defined in this standard install, load and operate correctly across all compliant implementations.

The ARC Spec defines a common base of functionality for compliant implementations, while allowing differentiation among competitive system products. It is sufficiently complete that hardware platform developers can determine what must be implemented to achieve a compliant system, while leaving room for value-added.

The base functionality defined for ARC-compliant systems allows a wide range of operating system, application and peripheral suppliers to have a common development target among a large number of multivendor system products.

System developers may implement features or capabilities beyond those defined in this base standard. That is, the absence of a feature or capability from this standard does not imply that such a feature or capability is precluded. System developers choosing to implement features and capabilities beyond the scope of the base standard must provide the additional hardware and software components necessary for the additional features to be usable by operating systems and applications.

The ARC Spec strives to define the minimum requirements for functional compatibility without constraining the designer to specific details such as register-level interfaces or I/O subsystems. Hardware platform developers maintain complete freedom of implementation below the level of interfaces and functions defined here. Systems may be developed that deviate from the base standard at the level of functions and interfaces defined in the ARC Spec. In such cases, nonconformance can be expected to have associated consequences, the nature of such consequences being dependent upon the nature and degree of noncompliance with the ARC Spec base standard. The burden of these consequences is assumed to be borne by the system developer.

1.1 Covered By the ARC Specification

This section explains the ARC Spec. The areas discussed are what is in the specification itself, what is does not cover, and how it is enhanced and extended.

The ARC Spec covers these main areas:

- System Architecture

  The system architecture refers to all components within the system, i.e. operating system, application software, hardware, firmware, operating states, etc. These components define the overall structure of a system and its interfaces.

- Hardware Standards

  The hardware standards define the base platform and peripherals. Some of the areas are connectors, signals, and protocols.

- System Configuration Standards

  The configuration standards describe the required and recommended configuration rules pertaining to the various types of configurations.

- Media Standards

  All types of media are defined so applications can know what is available.

- Firmware Standards

  The firmware defines the software services provided by the platform vendor. These are unique services provided by the firmware only.

- Console

  This section describes what functionality a console has and how it supports different character sets.

- Multiprocessor Platforms

  This section defines hardware and firmware requirements for supporting shared-memory, tightly coupled multiprocessor RISC systems.

- Network Bootstrap Protocols

  This section contains implementation notes on how ARC-compliant machines can boot from a multitude of ARC and non-ARC network servers using specific network bootstrap protocols.

1.2 Covered By Addenda To The ARC Specification

Addenda to this standard are used to specify requirements and recommendations that may add to the ARC Spec. These will be incrementally developing material and occur mostly in the area of I/O subsystems. For example, there could be an addendum describing the EISA bus.

1.3 Not Specified by the ARC Specification

The following areas are not specified by the ARC Spec. This allows the I/O and application program domains to have specific differences on each platform.

- System Operator Interfaces

  Any messages that scroll to the console screen are not covered by the specification. This occurs mostly during the bootup stage.

- Operating System Standards

  There is no mention of any operating system specifics, such as Windows NT or Unix.

- Application Programming Standards

  No mention is made of how code should be written or what languages can be used.

- Networking Protocol Standards

  The ARC Spec doesn’t specify protocols themselves, but does discuss some of the mappings of existing protocols necessary for booting on a network.

1.4 Conventions Used In the ARC Specification

- Specifications

  Specifications are given informally. Each section tries to clearly define what is Required and what is Recommended. See the Conformance section for specific details.

- Rationale

  Rationale explains why certain features are defined the way they are, but does not dictate how they should be implemented. This allows the platform vendor great flexibility in implementing their portions while staying within the boundaries of this standard. Where the rationale for various requirements or recommendations is provided, it is separated from the body of the specification by the following notation.

Rationale –

- Notes

  When a specific idea or piece of information needs to be emphasized, a note will be used, separated from the body of the specification as shown below:

Note –

- Programming Interfaces

  All programming interfaces specified by this specification are described using defined typographical standards and ANSI C as an informal specification language.

- Evolving Material

  Some materials, such as the network protocol mappings, are not yet well studied. There is some risk in the elevation of this material to “standard” level at this time. As experience and further study clarifies the issues discussed in these “preliminary” sections, the material would be moved into the body of the specification.

1.5 Conformance

This section defines more precisely what constitutes conformance to the ARC Spec and also works to legitimize the flexibility originally required by many participants. In order from the closest adherence to the ARC Spec to the least, the categories are:

1. Compliant Systems

   Compliant systems conform to all requirements and all recommendations laid out in this standard.

2. Compatible Systems

   Compatible systems conform to all requirements laid out in this standard, but do not follow all recommendations.

3. Non-Compatible Systems

   Non-Compatible systems do not conform to all requirements laid out in this standard.

Part 1 — Base Specification

System Architecture 2

This chapter discusses the system architecture. Diagrams are used to display the various architectural layers, and each layer is discussed in detail at the architectural level. Functional detail, where defined by this standard, is provided in the appropriate chapter later in this document.

Collectively, the data formats, subsystems, and interfaces described in this chapter comprise the “standard architecture.” These elements allow the definition of a systems architecture for compliant systems, while allowing maximum designer flexibility in the actual hardware implementation. This flexibility is achieved by defining mechanisms so that the specifics of the hardware implementation are abstracted by either firmware, the HAL, or by loadable operating system-dependent components provided by the hardware system developer for each of the relevant subsystems.

2.1 Architectural Working Statement

From an architectural perspective, this specification combines a set of subsystems with a set of system interfaces in a manner that will support binary compatible operating systems and applications. In addition, these components are defined to maximize product flexibility and to foster innovation by allowing a wide variety of implementations within the standard.

2.2 System States

The overall architecture for the ARC Spec consists of a four-state system. As can be seen in Figure 2-1, each state has unique operating assumptions and ramifications on the environment. Each state and its transition is described below.

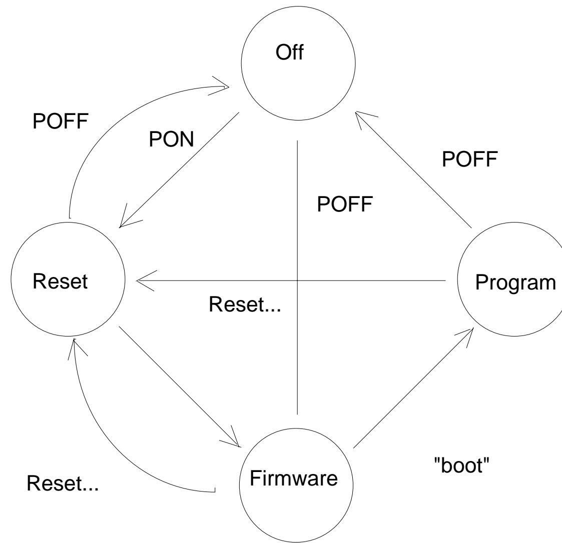


Figure 2-1 System States

All ARC-compliant systems always operate in one of four distinct states:

1. Off State
2. Reset State
3. Firmware State
4. Program State

2.2.1 State Diagram

The state diagram for this system can be represented as shown below:

```
OFF      ------- (On)            ------> RESET
RESET    ------- (Off)           ------> OFF
RESET    ------- (restart        ------> PROGRAM
RESET    ------- (automatic)     ------> FIRMWARE
FIRMWARE ------- (Off)           ------> OFF
FIRMWARE ------- (implicit)      ------> PROGRAM
PROGRAM  ------- (Off)           ------> OFF
PROGRAM  ------- (Termination)   ------> RESET
```

The circles represent the machine for each state. The arrows are the transitions from one state to the next. POFF represents the PowerOff state; PON represents the PowerOn state; boot is the state of the machine booting the operating system; and Reset is the state of the machine being physically reset. Each description below further describes the actions that occur at each level.

Off State All ARC systems preserve non-volatile memory in this state along with the local time. An implementation may also preserve main memory. In this case, no time during which main memory is preserved is specified.

Reset State This state is entered when power is turned on, when a system reset is asserted, and when various program termination functions are invoked from program mode. For systems that support powerfail/restart, while in the RESET state, platform firmware must check the validity of the restart block. If the restart block is valid, the system must reenter the PROGRAM state at the address specified in the restart block without modifying the contents of memory. If the restart block is invalid, platform and/or vendor specific operations are performed, and control then enters the FIRMWARE state. If main memory is not preserved across a powerfail, then the restart block is not used.

Firmware State This state is entered automatically from the RESET state. In this state the platform firmware may enter an interactive mode and automatically initiate a system bootstrap operation which loads and invokes software from disk media or from the network. The firmware provides various input/output, configuration, and utility routines that may be called by this software. The software loaded in the firmware state is restricted in what state it may modify. In particular, it may not modify processor or peripheral control registers. While in the FIRMWARE state, only platform firmware is allowed to modify this state.

Note – While this specification acknowledges that a platform may provide an interactive firmware mode, it does not require such a mode nor does it specify anything about what capabilities this mode might provide or the nature of the operator interface.

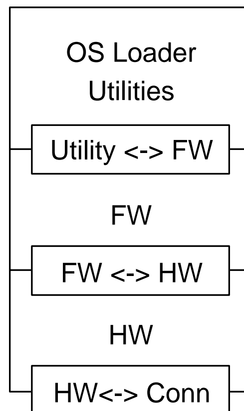


Figure 2-2 Firmware State

Program State This state is entered implicitly when software loaded in the FIRMWARE state takes over responsibility for management of all hardware state. Once the PROGRAM state is entered, in general the system may not safely return directly to the FIRMWARE state. In the PROGRAM state, loaded software is responsible for all input/output, memory management, interrupt and exception handling, and so on.

Note – See Section 4.5, "Interrupts and Exceptions," on page 100, for a discussion of a somewhat restrictive technique where the FIRMWARE state may be temporarily reentered from PROGRAM state.

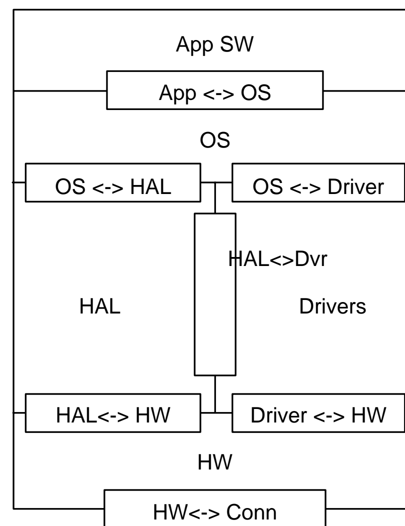


Figure 2-3 Program State

2.3 Software Subsystems

This section describes the software components of the ARC Spec. They are divided up into the following areas:

1. Application Software
2. Operating System (OS) Software
3. Hardware Adaptation Layer (HAL) Software
4. Device Drivers
5. Independent Utility Software
6. Loader Software

2.3.1 Application Software

For an application, the Application Binary Interface (ABI) is defined by the operating environment on which it executes, with certain constraints defined by characteristics of the underlying platform. Compliance with these requirements assures that an application will execute on any ARC-compliant platform/OS combination. In particular, the instruction set, any rules on instruction sequencing imposed by the hardware, and the interpretation of the byte order within multibyte data are part of the ABI (regardless of the OS involved). Therefore, these elements must be standardized.

Hardware and software developers are expected to conform to the following:

- ARC-compliant systems must implement the user mode instruction set defined by the MIPS I architecture. Since MIPS II is a superset of MIPS I, hardware platforms can be based on either the MIPS I or the MIPS II architecture. In either case, they must support both the base instruction set and the floating point instructions.
- Application developers should develop for the MIPS I instruction set (user mode instructions; MIPS I is a subset of MIPS II).
- Little endian byte ordering is the standard interpretation of multibyte data.

2.3.2 Operating System Software

- Compliant systems employ operating systems utilizing the “shrink-wrap” HAL/Driver layering model. Direct “ports” of operating systems do not comply with the model (such systems would be compatible, not compliant).
- Operating system developers should develop products that execute on platforms using processors having system coprocessors compatible with either the R3000 or the R4000 microprocessor. The OS has knowledge of machine language.
- Operating system code may be implemented using either MIPS I or MIPS II instruction sets.

2.3.3 Hardware Abstraction Layer (HAL) Software

- Provides services not normally provided by the OS.
- Provides the interface between the hardware and the operating system. This allows the OS and device drivers to be fairly machine independent.
- For a given OS maps the defined interface for that OS onto the system hardware implementation and encapsulates all hardware dependencies that are not otherwise part of the referenced standard. Hal has knowledge of platform architecture.

- Interfaces are defined by the specific operating system and are incorporated into the ARC Spec by reference to OS specifications.
- Is loaded and bound to the OS at system load time.
- HALs are developed and provided by hardware system developers for each operating environment embraced within the ARC environment.

Rationale – To enable the development of shrink-wrap operating environments, operating environments interface to a programmatic abstraction of the underlying hardware (the HAL) rather than to the hardware itself. To minimize any performance impact of this abstraction, the abstraction is loadable rather than being fixed in firmware. This allows the abstraction to be optimized for each supported operating system. The appropriate abstraction is loaded as the operating system is loaded and initialized. This requires that the OS loader be cognizant of the HAL and that the OS load/initialization process effect the loading and binding of the HAL with the OS.

- All implementations must incorporate means to enforce strong ordering of read/write transactions where specifically required for correct operation. (Essentially, strong ordering assures that a sequence of writes, or reads and writes, completes in the order in which they are executed.) Implementation of this capability may be in hardware, software procedures embedded in firmware or a HAL, or a combination of both. The platform must provide the capability to know what the write order will be.

2.3.4 Device Drivers

- Drivers are OS and hardware platform specific.
- Typically supplied by the platform vendor, the OS vendor, or the peripheral vendor.
- Drivers should be developed for the MIPS I instruction set.

2.3.5 Loader, Installer, and Independent Utility Software

- Installer installs software to the hard disk
- Loader loads OS into memory, and is OS specific.
- Consists of programs run when the system is in Firmware State and are used to perform various utility functions (such as peripheral card configuration, disk formatting and partitioning).
- May utilize ARC standard firmware capabilities, or capabilities defined by addenda applicable to the system on which they are employed.
- May not directly access platform hardware.

2.4 Hardware Subsystems

Two categories of hardware subsystems are defined:

1. Processing Subsystems
2. Peripheral Attachment Subsystems (generally I/O buses)

Within each category, the ARC Spec permits, but does not require, multiple types. The I/O subsystem may vary significantly between implementations, while significantly less flexibility is permissible in the implementation of the processing subsystems.

2.4.1 Processing Subsystems

Since the primary objective of the ARC Spec is to facilitate the development of compatible systems from many different vendors that will execute the same software, all processing subsystem requirements are based on achieving this goal.

Processing subsystems include the combination of central and floating point processors, main memory, memory caches (such as internal, external, primary, secondary), Real Time Clock (RTC), precision timing capability, and associated interconnect structures that combine to execute code. All processing subsystems are based on MIPS processors, in little endian byte ordering, and include floating point capability.

2.4.2 Peripheral Attachment Subsystems

Peripheral attachment subsystems (or buses), provide system access to standard and optional devices. Buses that conform to different specifications constitute different types. The ARC Spec may incorporate different types of buses; for example, the Extended Industry Standard Architecture (EISA) is an extension of the ISA bus, the latter is the industry standard for *x*86-based personal computers.

The ARC Spec does not limit the type of bus that may be employed, nor does it define a specific manner in which a given bus is implemented or supported. The specification only requires that an implementation conform to the interface requirements embodied in the base standard, the interface defined for each supported operating system and the relevant addenda. It recognizes that other buses such as the VMEbus and Futurebus+ are relevant in various market segments and establishes a framework in which such buses can be accommodated within the standard if and when that is warranted. The ARC Spec, through the unifying framework it defines, establishes the means by which systems with different buses may participate in the standard.

It is possible to have a “bus-less” system. By definition, this means that the platform does not have an ARC specified bus, but an internal non-standard bus defined by the platform vendor.

2.5 Firmware

System firmware provides an execution environment that supports initial program load and program execution. This environment includes a set of system files, the data structures inherited by loaded programs, a set of standard functions for accessing system resources, and means for passing input arguments and the system environment to loaded programs. See Chapter 4, “Platform Firmware,” for more detail.

2.6 System Interface Definitions

ARC compliant systems are implemented by combining a number of different types of subsystems. The need for shrink-wrap operating environments requires that operating environments not explicitly depend upon specific underlying hardware implementations of subsystems. Therefore, additional architectural elements in the form of standard interfaces exposed to operating environments (such as operating systems, their installers or loaders, and drivers) are defined. These interfaces are described below. Please refer to Figure 2-2 and Figure 2-3 for the following definitions.

The two most important interfaces we will look at here are when the machine is operating in Firmware State. These interfaces are:

1. **Utility/Loader <-> FW** Consists of the FW API that is available to the utility software. Since FW is developed by the hardware provider, this is platform specific.
2. **HW <-> Connector** Consists of cabling and other hardware attachments.

Platform Hardware 3

This chapter provides a detailed description of the required and recommended properties of each of the hardware components defined by the ARC Spec configurations. We will also look at the formats of the different media types for this specification.

3.1 System Configurations

The component list is divided into four categories. These categories define the different system configurations and their add-on capabilities.

1. Server System

The minimum amount of hardware that will support a running system. This feature list is included in all other configurations. Also note that while storage functionality or capability must be provided, the standard does not require any specific implementation of the storage capability. For example, storage capability may be realized through attached fixed disk devices or over a network by means such as virtual disk techniques.

- MIPS CPU/FPU
- Memory
- Storage
- Timers
- System Load (Boot) Capability
- Console
- CD-ROM accessibility

Note – Windows NT will require keyboard and video to run. Therefore it does not support this configuration.

---

2. Desktop System

   Includes the Server configuration plus the extra items needed for a desktop workstation system.

   - Keyboard
   - Pointing device
   - Video
   - Audio

3. Optional Hardware

   All other items that can be added onto the Server and Desktop configurations. These items are not required, but if they are used must conform to the ARC Spec.

   - Floppy Drive
   - Serial Port(s)
   - Parallel Port(s)
   - SCSI
   - Network Device(s)

4. Additional Hardware

   Items that are not required or included in the ARC Spec. They are either proprietary or covered in Peripheral Addenda.

3.2 Server System Configuration

The following functions are required on all ARC-compliant systems.

In the following sections, specific requirements for standard system elements are given. These requirements address functional capabilities required, and, in certain cases, some or all of the interface requirements in terms of connectors, signals, and protocols.

The provisions of this section of the specification are intended to establish a baseline capability appropriate for typical or mainstream use. It is recognized that for certain classes of systems, for example, portables and laptops, the most appropriate technologies may not allow satisfaction of some of these requirements. On the one hand, in such cases, it is desirable for applications to still work, although operation may be more complex and difficult. On the other hand, mainstream operational ease and capability should not be compromised by establishing the least common denominator technologies/capabilities as the baseline for application developers. This standard thus allows flexibility, which assures the broadest possible range of systems with the broadest possible support for industry standard applications.

This section defines a set of system elements that are standard across all compliant implementations. The objectives in defining such a set of elements are twofold. First, this set of system elements defines the interface to the system as seen by system and application software. As such, these standards define the common target for developers, (for example, a standard keyboard and a standard minimum video).

Second, by defining a guaranteed minimum set of standard elements (that is, a set of elements that are always present), system and application software developers know what base level of functionality they can assume to always be present.

Consequences of Non-Compliance

Across a broad spectrum of systems and system developers, there may be circumstances where absolute compliance with the standards set forth herein is not appropriate. It is not the purpose of this architecture to preclude variance in such cases. However, under such circumstances, the burden of noncompliance must be borne by the system developer. That is, developers of compliant systems can assume that compliance will enable full participation in the environment created by the standard with no extraordinary effort on their part. In the case of partially compliant systems, full participation in the environment may not be achieved, or achievement thereof may require specific effort on the part of the system developer, to mitigate the effects of noncompliance (such as definition and development of drivers and special services or development of special OS adaptations).

Elements are generally specified as part of the minimum capabilities because of their role in contributing to the development of uniform approaches to widely used capabilities and leveraging of the established personal computer infrastructure. These reasons are noted here and not repeated in each section. Additional objectives peculiar to specific elements are noted, as appropriate.

Certain consequences of deviation from the guaranteed minimum capabilities are common across all elements and are noted here. For systems that conform to all interface characteristics, leverage of third party add-on devices and software is maximized. However, while use of nonstandard interfaces, either physically, logically, or electrically, reduces leverage, complete operational compatibility can be maintained. Deviations that reduce operational compatibility may maintain functional compatibility through adaptation of the methods used for operation (for example, key chords on one keyboard can be defined to perform the function of a single key on another system). In general, systems that do not provide at least functional capability will be subject to compatibility restrictions and will not be able to participate fully in the environment that develops around this standard.

3.2.1 Processor Unit

Processing subsystems consist of a processor, memory caches (internal, external, primary and secondary), and memory and the interconnect structures they share. Each unique type of processing subsystem must be assigned a unique type that can be determined from an entry in the configuration tree. Based on the type and associated key, additional processing subsystem information can be obtained. All ARC-compliant processing subsystems must meet the following requirements:

- ARC-compliant systems must implement the user mode instruction set defined by the MIPS I architecture. Since MIPS II is a superset of MIPS I, hardware platforms can be based on either the MIPS I or the MIPS II architecture. In either case, they must support both the base instruction set and the floating point instructions. Floating point exceptions must be precise.
- ARC-compliant systems must be based on processors whose system coprocessors are compatible with the system coprocessor implemented on either the R3000 for MIPS I based platforms or R4000 for MIPS II based platforms.
- Processors used in ARC-compliant implementations must not impose interlock or hazard avoidance requirements beyond those of the processor (R3000 or R4000) with which it is otherwise compatible in accordance with the above two requirements.
- The processor used must be determinable from the configuration tree.
- ARC-compliant processing subsystems must use little endian byte ordering for multibyte data.
- ARC-compliant processing subsystems must provide means to enforce strong ordering of load/store transactions on demand. That is, it must be possible to ensure that specific, individual transactions will be performed in a strongly ordered fashion. This does not imply mode setting, such as “strongly ordered mode” versus some other mode. Rather, strong ordering applies on a transaction basis, and then only to transactions specifically identified as needing to be strongly ordered. In those situations where strong ordering is required, the following conditions must be met:
  1. Writes must complete in the order in which they are executed.
  2. The size of operations must be preserved. That is, successive writes of one size must not be combined into a single transaction of a larger size. For example, four successive bytes stored to successive byte addresses must not be combined into a single write transaction of a word spanning the four addressed bytes.
  3. Successive writes to the same address must not be collapsed into a single write transaction.
  4. Readaround must be prevented. If a read follows a write, the read must not take place until the write has completed.

With respect to strong ordering requirements, the HAL must provide an interface that can be used by higher level code, such as the operating system or device drivers, when the situation requires that the above mentioned conditions be precluded. Developers of system software and device drivers must determine the situations where ordering must be guaranteed and use the defined HAL interface at these points. Hardware implementations themselves must not be assumed to guarantee strong ordering.

3.2.2 Floating Point Unit

Note that the processor requirement includes floating point capability. All ARC-compliant systems must provide the standard MIPS FPU functionality.

3.2.3 Cache

Processing subsystems may be implemented with various cache architectures. For example, some implementations may have separate secondary caches, others may have unified secondary caches, while still others may have no secondary cache at all. Similarly, the cache architecture implemented in specific realizations of the MIPS I and MIPS II architectures may vary. Operating systems must not make any assumptions regarding the presence or size of a cache. Firmware functions are defined in this base standard, and HAL functions are defined in the various operating system specifications which may be used to query the system configuration structure to determine the presence and characteristics of the caches implemented on a specific system. In addition, HAL functions are defined to perform such functions as cache flushing in a manner appropriate for a specific implementation transparently to the caller.

3.2.4 Memory

In addition to instruction execution and data representation, programs require other minimum processing subsystem resources. Systems must have a minimum of 16 MBytes of memory, with at least 8 Kbytes at physical address 0.

3.2.5 Timing Function Support

A mechanism is required that can be used by the HAL to implement micro level time delays. While the implementation of underlying hardware is not specified, the granularity of the delays achievable via the HAL function must not be less than 1 microsecond.

3.2.6 Real Time Clock

ARC-compliant systems must provide a real-time clock. The contents of the real time clock will be backed up by a battery. The format in which the real-time clock device maintains the current time is not specified. Rather, system firmware and HALs provide a function that returns the time in a specified format. The time is obtained and the format converted in whatever manner is appropriate. However, regardless of how the time is set and stored, the GetTime firmware must return the correct time in UTC form.

3.2.7 System Timer

ARC-compliant systems must provide a system timer which generates interrupts on a periodic basis. The interval between interrupts must be programmable to include at least intervals of 1.0000 millisecond and 10.0000 milliseconds.

3.2.8 Console

This is a character mode interface, which does not have support like the system console. NT does not support character mode consoles.

3.2.9 CD-ROM

To facilitate the distribution of software and to ease the installation of large complex programs, the preferred medium for distribution of software for ARC systems is CD-ROM. To assure that software distributed on the preferred medium will be readily installable on any ARC-compliant system, all ARC-compliant systems are required to have access to a CD-ROM device.The required access can be achieved through direct attachment or over a network (if it is supported by the firmware). Access to a CD-ROM device assures that software developers who choose to distribute their products on CD-ROM will be able to have their software installed without special consideration.

3.3 Desktop System Configuration

3.3.1 Keyboard

This section describes the keyboard requirements of ARC-compliant systems. Consequences that can be expected as a result of noncompliance with these requirements are also outlined.

Requirements

The keyboard standard specifies the following keyboard characteristics:

- Number of keys (including variations for internationalization)
- Keycap markings (including variations for internationalization)
- Placement of keys (including variations for internationalization)

This standard includes the concept of a system console input device. When the keyboard device has such functionality, the requirements are as defined in the section on system console and are not part of the keyboard requirement.

The standardization of the keyboard provides an established layout for software.

The keyboard standard specifies the following characteristics:

Table 3-1 Keyboard Standard

Characteristics	Standard Requirements
Keyboard layout	Personal Computer industry standard Enhanced keyboard (101-key) layout and National Enhanced keyboard (102-key) layout are supported.

The Enhanced Keyboard standard layout features a separate cursor control key cluster and 12 function keys (F1 to F12). The National Enhanced Keyboard (102-key) layouts support international languages.

3.3.2 Pointing Device

This section describes the pointing device requirements of ARC-compliant systems.

Requirements

The standardization of the pointing device interface allows for third party development of a wide variety of pointing devices. The pointing device characteristics covered by this specification include:

- Two dimensional positioning
- The number of discrete user-operable controls (buttons or equivalent)

The pointing device characteristics defined by this standard are as follows:

Table 3-2 Pointing Device Characteristics

Characteristics	Standard Requirements
2 Dimensional positioning	X-Y axis movement
Number of push-buttons	2 minimum

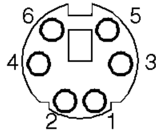


Figure 3-1 6 Position DIN connector for EISA Keyboard/Mouse Connectors

3.3.3 Video Subsystem

This section describes the video requirements of ARC-compliant systems. Consequences that can be expected as a result of noncompliance with these requirements are also outlined.

Requirements

The video characteristics defined by this standard are as follows:

- Minimum pixel counts (horizontal and vertical)
- Pixel aspect ratio
- Minimum numbers of bits per pixel and colors per palette

The program interface to the video subsystem is not specified as part of this standard. It is assumed that access to the video subsystem is through an implementation-dependent device driver.

The following base set of video characteristics is defined by this standard:

Table 3-3 Video Characteristics

Characteristics	Standard Requirements
Pixel counts	1024 x 768
Pixel aspect ratio	1:1 (Square Pixel)
Color scale/grayscale	256 out of 16.7 million color palette 256 grayscale levels

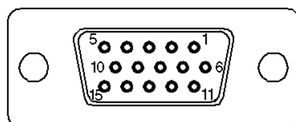


Figure 3-2 15-Pin D_SUB for EISA Video Connector

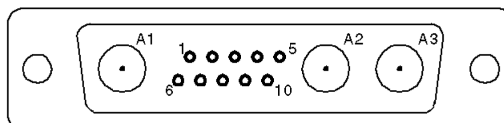


Figure 3-3 Combination Receptacle for EISA Video Connector

The program interface to the video subsystem is not specified as part of this standard. It is assumed that access to the video subsystem is through an implementation-dependent driver. While this standard embraces the notion of a system console, the display capabilities required to provide output console functions are not part of the video requirement. When output console functionality is realized by way of the video subsystem, the functionality required of firmware to map the output console onto the video subsystem is defined by the console requirement. Finally, some systems may require only a system console (for example, servers that have no reason to interface to users for any application). In such cases, the presence of a compliant video subsystem is not required and only the console functionality is required. Other systems, such as Windows NT, requires video to be present and will not run without it.

Failure to provide the required aspect ratio will result in distorted presentation such as malformed circles.

While systems and application software are expected to be developed so that they can adapt to different pixel counts, the establishment of minimum pixel counts allows developers to know what assumptions are appropriate with respect to screen presentations. Systems providing smaller pixel counts may lose some operational compatibility, that is, applications will operate correctly but the user may not be able to use them in the manner intended. (For example, limitations on the maintenance of multiple windows on smaller resolution displays may require users to alter their style of use.)

3.3.4 Audio

All ARC-compliant systems must be equipped with audio input and output capabilities. The functionality defined is the minimum required to meet minimum business audio needs The objective of this audio specification is to define the minimum audio functionality within a platform environment to meet the minimum audio needs. All ARC-compliant systems must provide support for accepting audio signals and converting these to one of several digital representations for processing or storage by programs. Similarly, all ARC-compliant systems must provide support for converting specified digital representations of audio signals to audio output.

The input/output frequency and sample format capabilities are shown in Table 3-4 and the platform audio interface must conform to the interface requirements shown inTable 3-5. The recommended monaural and stereo audio connectors are shown in Table 3-6.

Applications processing audio do so through an interface provided by an operating environment. The audio file or data formats processed by applications is defined by the audio Applications Program Interface (API) of each supported operating system. However, these formats imply the existence of certain characteristics in the data streams passing between device drivers for the audio subsystem and the operating system.

This specification requires that ARC-compliant systems support certain sets of characteristics. The required support may be achieved through any combination of hardware and software techniques. The requirement of this section are met if the device driver for the audio subsystem is capable of both accepting data in any of the required formats and providing data in any of the required formats.

Specific digital audio data formats required to be supported include:

Table 3-4 ARC System Audio Requirements

Audio Data	Standard Requirements
Input	Eight-bit, single channel, linear samples at 11.025 Khz, eight-bit, single channel, companded samples at 8 Khz.
Output	Eight-bit, single channel, linear samples at 11.025 and 22.05 Khz, eight-bit, single channel, companded samples at 8 Khz.

Table 3-5 Audio Interface Electrical Requirements

Jacks	Type	Impedance	Voltage
Microphone	3.5mm Mono	50 KOhm	77.5 mV RMS max
Line In	3.5mm Mono	10 KOhm	5.6 V p-p max., 2 V RMS max.
Headphone	3.5mm Mono	32 Ohm	100 MW Min.
Line Out	3.5mm Mono	500 Ohm	5.6 V p-p max., 2 V RMS max.

Table 3-6 Recommended Audio Connectors

Audio Connector	Example Part Number	Diagram
Monaural	Shogyo International Corp. TSH-3523A	Signal; Ground; 3.5 mm
Stereo	Shogyo International Corp. TSH-3523	Right Channel; Left Channel; 3.5 mm; Ground

Note that the minimum requirement is for mono support. However, to avoid incompatibilities with vendors who choose to provide stereo support, recommended stereo jacks and connectors are also shown.

ARC systems must provide at least one pair of audio jacks where the pair is either one microphone-in jack and one headphone-out jack, or one line-in jack and one line-out jack.

Additional audio functionality beyond the standard may be implemented on ARC-compliant systems.

Note – This standard defines sample format and sample frequency requirements only for ARC-compliant systems, not for ARC platforms. The intent of this distinction is that while application level software which accesses the system audio capabilities must always see a system that conforms to the sample format and frequency minimum requirements, the platform hardware need not directly implement these capabilities. For example, a platform might not implement 8 Khz companded eight-bit input but might implement only 11.025 Khz eight-bit linear input. This hardware capability, when supported with an audio device driver that compressed both the bandwidth and the dynamic range to 8 Khz companded input, would satisfy the standard's requirements.

Rationale – The flexible audio standards defined by this standard reflect the view that while it is necessary to guarantee that application software can rely on the presence of audio capabilities, it is not necessary that all platforms supply high quality audio. Speech grade audio (eight-bit, 8 Khz companded) audio is adequate for many contexts while higher grade audio is required for others. The need to insulate application software from the hardware capabilities without forcing it to assume the least common denominator is the reason the audio requirements are defined as they are.

Needless to say, platforms with minimal audio hardware that rely on software signal processing to implement required audio capabilities may not be capable of delivering quality audio for some of the required audio formats.

3.4 Optional Hardware

This section enumerates the types of hardware that are considered optional. The following devices, if provided by an ARC-compliant system, must be provided in a specified manner:

The following devices are not required but if they are present, some of their properties are covered by the ARC standard:

3.4.1 Floppy Drive

The device uses the industry standard for personal computers.

3.4.2 Serial Ports

The characteristics established by this standard for serial ports include:

- Modes of operation
- Speeds supported
- External connector, signals and signal characteristics

Developers may choose to meet the requirement for a system console by providing for the attachment of terminal devices (or terminal emulators) via a serial port. In this case, the console functionality required is defined by the console requirement and is not a port requirement. As such, it is specified in another section.

The characteristics established by this standard for the serial interface include:

Table 3-7 Serial Interface Characteristics

Characteristics	Standard Requirements
Standard interface	EIA-232
Speeds supported	19.2K, 9600, 4800, 2400, 1200, 600, 300, 110 baud
Protocol	Asynchronous
Connector	Subminiature D, 9-pin, plug. The pinout of the connector is shown in Appendix B.

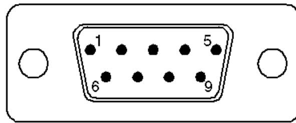


Figure 3-4 9-Pin D-SUB for Asynchronous Serial

The serial interface is signal compatible with EIA-232 with a 9-pin D connector as shown in Figure 3-4.

Developers may choose to meet the requirement for a system console by providing for the attachment of terminal devices (or terminal emulators) via a serial port. In this case, the console functionality required is defined by the console requirement and is not a port requirement.

3.4.3 Parallel Port

The characteristics established by this standard for parallel ports include:

- 8-bit bidirectional operation
- External connector, signals and signal characteristics

The characteristics established by this standard for the parallel interface include:

Table 3-8 Parallel Interface Characteristics

Characteristics	Standard Requirements
Standard	Centronics
Protocol	Centronics 8-bit bidirectional
Connector	Subminiature D, 25-pin, receptacle. The pinout of the connector is shown in Appendix B.

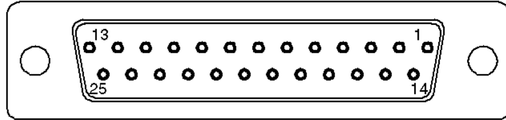


Figure 3-5 25-Pin D-SUB for Parallel Connector

3.4.4 SCSI Interface

Small Computer System Interface (SCSI) is an ANSI standard specification for a peripheral bus and command set. SCSI specification defines a high-performance peripheral interface that distributes data among the peripherals independently of the host.

Systems may be required to support the attachment of a wide variety of peripherals such as CD-ROMs, storage subsystems, tape backup systems and other peripherals. The prescribed method is a SCSI interface. The characteristics established by this standard for the SCSI port include:

- The SCSI protocol
- The external connector
- The signals and signal characteristics associated with SCSI
- The command set

The characteristics established by this standard for the SCSI interface include:

Table 3-9 SCSI Interface Characteristics

Characteristics	Standard Requirements
SCSI protocol	ANSI standard X3.131-1990 (Revision 10c)
SCSI command set	ANSI standard X3.131-1990 (Revision 10c)
SCSI connector	ANSI standard X3.131-1990 (Revision 10c) Shielded Connector Alternative 1 - A cable High-density single-ended 50-pin.The pinout of the connector is shown in Appendix B.

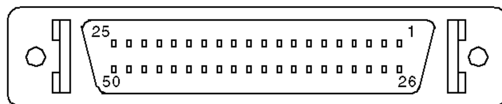


Figure 3-6 50-Pin Single Ended SCSI Connector

All SCSI devices attached to ARC Standard Specification systems must support all functions included in the SCSI common command set CCS as they apply to that device.

3.4.5 Network Interface

ARC systems that provide an interface to either an Ethernet or Token Ring network must meet the specifications below. This neither requires nor precludes multiple network interfaces.

Ethernet

If an Ethernet connection is provided, it must provide for programmable selection of operation according to either the Ethernet or IEEE 802.3 standard at device initialization.

The possible media connectors are as follows:

Table 3-10 Possible Ethernet Media Connectors

Specification	Type	Connector
10Base 2	thin Ethernet	BNC
10Base-T	UTP Ethernet	RJ-45 (ISO 8877)
10Base 5	thick Ethernet	AUI

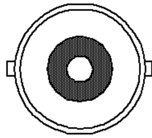


Figure 3-7 Ethernet BNC Jack

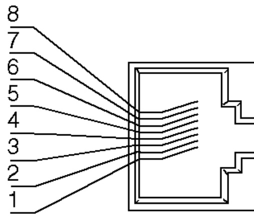


Figure 3-8 RJ-45 Connector

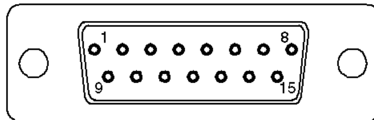


Figure 3-9 15-Pin D-SUB receptacle for Ethernet AUI Connector

Token Ring

If an ARC-compliant system provides a Token Ring, it must provide connectivity to a Token Ring network in accordance with the IEEE 802.5 "Token Ring Access Method and Physical Layer Specifications."

The possible media connectors are as follows:

Table 3-11 Possible Token Ring Media Connectors

Specification	Connector
STP (shielded twisted pair)	Subminiature D, 9-pin
UTP (unshielded twisted pair)	RJ-45

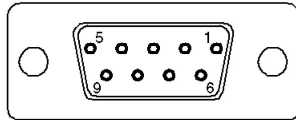


Figure 3-10 9-Pin D-Sub plug connector for Token Ring

3.5 Additional Hardware

Additional components are components that are not required. They are either proprietary or covered by Peripheral Attachment System Addenda fit into the standard.

3.6 Media Formats

To support the interchange of media and exchangeable devices, this standard defines system partition formats for all media supporting system load operations (bootstrap). In addition, basic requirements to ensure physical readability of media and devices are also defined.

In general, the external interfaces of compliant systems are defined by industry standards outside of this specification or are referenced as part of the standard applicable to a specific device (such as a network interface). However, specification of certain standards is shown below.

1. Media Formats for System Load
2. Diskette formats
3. CD-ROM System Loading
4. Fixed Disk Storage Devices Partitioning
5. Network System Load Protocols
6. Data Interchange Standards

3.6.1 Media Formats for System Load

To facilitate system load from removable media and interchangeable devices, the structure of system load information on any such medium or device must be defined within the standard. Devices for which system load requirements are established include diskettes, fixed disk devices (SCSI or other), and CD-ROM devices.

To facilitate system load from a remote device (that is, over a network or communications port), the remote system load protocols are included in the standard.

3.6.2 System Partition Formats

Any device or medium from which system load is to be performed must contain a system partition. The sections below specify the structures of the system partitions for the bootable devices encompassed by this standard. System partitions are based on a number of different file systems, for example, FAT, ISO 9660, and others, depending upon the device. Within a file system, directory and file names may be case insensitive (FAT), or case sensitive (ISO 9660). System firmware, other than file systems, is case sensitive and never alters the case of a file or directory name. Similarly, different file systems impose different constraints on directory and file-name length and structure. Inconsistent use of case and/or structure can result in file names that cannot exist on some system partitions. Therefore, files that are intended to be placed in system partitions on any arbitrary device should be named according to the most restrictive requirements.

3.6.3 Diskettes (5 1/4-inch and 3 1/2 inch)

Diskettes follow the standard formatting conventions for personal computer systems. As such, they consist of a single partition, with the file system in that partition being the FAT file system. The system partition for bootable diskettes is contained within this single FAT partition. Within the system partition, the same conventions as applicable to fixed disks apply. That is, the directory structure, and names for components stored in that directory structure, are the same as for fixed disk devices.

Note that the standard personal computer format for bootable floppies contains code in sector 0 that is specific to the personal computer architecture. Diskettes that are bootable in an ARC context do not use this information. Further, ARC-compliant systems must not alter or otherwise corrupt this information. Alterations to this information will interfere with the readability of diskettes on personal computers. Diskettes formatted for some other file system will not, in general, be usable as system load devices on ARC-compliant systems.

Note – Although there are many sources of written information on FAT, no suitable formal specification is available for reference. One suitable source of information on the current implementation of FAT is *Microsoft MS-DOS Programmer's Reference, Version 5*.

3.6.4 CD-ROM

The CD-ROM standard for system load operations is based on the ISO 9660 standard with the same conventions as established for disks applicable within the CD-ROM system load partition as defined by ISO 9660.

The system partition on CD-ROM is in the partition described by the Primary Volume Descriptor for the CD volume. Within that partition, the structure of information is based on the file system defined by ISO 9660 with the following additional restrictions.

- The standard system partition structure is contained in a subdirectory of the root directory named SYSTEMPARTITION.
- Path component and file names within the SYSTEMPARTITION subdirectory must follow the 8.3 and case conventions of the FAT file system.

3.6.5 Disk Storage Devices

Disk storage devices follow the partitioning conventions defined by the personal computer standard. That is, the physical location and logical structure of partitioning information on fixed disks follows the standards for the PC, including extensions currently under definition. Within this framework, new type codes are defined, one of which uniquely identifies ARC Standard Specification system partitions.

Within the system partition, the file system employs the standard FAT file system disk structures and formats as defined by the FAT file system, as implemented in DOS version 3.0 and later. ARC systems should be able to read disks formatted with either 12 or 16 bit FAT entries but should create only 16 bit FAT entries when formatting disks. As with diskettes, there is no suitable formal specification for FAT and fixed disk partitioning that can be cited here. As above, a suitable source of information on the relevant implementations of FAT is *Microsoft MS-DOS Programmer's Reference, Version 5*.

3.6.6 Network

ARC-compliant systems may support system load procedures over a network based upon either the bootp/tftp protocols or the DLC/RIPL protocol. In both cases, an intelligent system load server is assumed to complete the dialog. This includes supporting a system partition construct. In the network case, the load server maintains, at least conceptually, a unique directory for each load client, organized as would be a system partition local to the client.

This section is optional and should be considered as a recommendation.

3.6.7 Data Interchange

To exchange data stored on removable media, the format and structure of information on the media must be known. Therefore, specification of the structure of information at physical read-and-write level is included in this standard. The layering of file structures on top of the physical structure is file-system dependent and is not specified. The requirement is to assure that media or devices can be physically read and written on any ARC-compliant system. This enables the file system to operate on whatever is read or written according to its own needs. Specific media standards are provided in Section 3.6 for the following media:

- Diskettes (5 1/4" and 3 1/2")
- CD-ROM devices
- SCSI fixed disk storage devices

3.7 Processing Subsystem

In addition to instruction execution and data representation, programs require minimum processing subsystem resources. Therefore, within any processing subsystem, the following are required:

- All processing subsystems must provide a minimum of 16 Mbytes of contiguous physical memory. The physical memory must include at least 8 Kbytes at physical address 0. The remainder of the 8 Mbytes must be contiguous and addressable via kseg0/kseg1 addresses (it must be accessible without mapping). It is acceptable for there to be two pages starting at 0 (the 8 Kbytes required above) and a 16 Mbyte minus 8 Kbyte contiguous region to start at an address above 0x2000. That is, the required 8 Kbytes starting at 0 need not be contiguous with the remainder of the required 16 Mbytes. If an implementation achieves physical page 0 and 1 by an aliasing technique, i.e. by hardware mapping of these addresses to the first two pages of the actual physical memory, the address range corresponding to the aliased space must have an associated memory descriptor showing these pages to be bad or unusable.
- The initial environment in which programs loaded by system load procedures execute must contain at least 4 Mbytes of free, contiguous physical memory space (including the space occupied by the program itself).

  Other aspects of the processing subsystem (for example, cache organization) are not specified by the standard. Rather, variations in implementation outside of the areas standardized are hidden by the hardware abstraction layer (HAL).

The ultimate achievement of the objective of a shrink-wrap environment depends on additional factors beyond the scope of this document. Specifically, system coprocessors and the functions they provide are directly visible to both programs that run on top of the standard system firmware interface and to operating systems.

The differences among system coprocessors in different implementations are not abstracted (for example, the TLB and status registers). Changes in system coprocessors from those defined by the R3000 and R4000 may introduce incompatibilities with existing operating environments. Compatibility with the system coprocessors of the R3000 or the R4000 must be maintained to maintain full binary compatibility with shrink-wrap operating systems.

It is the responsibility of the designers of future MIPS architecture derivative processors to maintain this compatibility. Note that unless these constructs are exposed by the standard Application Binary Interface for an operating system, well-behaved applications do not touch them, and application compatibility is not affected. In general, operating environments do not expose these constructs.

3.7.1 Related Consequences

Processing systems that implement a different instruction set or big endian byte ordering will be incompatible and will not execute standard (shrink-wrap) applications or operating systems correctly. Such systems are considered outside of this standard.

Operating environments that are otherwise compliant, but expose the system coprocessor details directly to applications, will not be able to assure application compatibility across all ARC-compliant platforms.

Processing systems that impose requirements on instruction interlocking or which introduce hazards beyond those defined for the R3000 for user mode execution will, in general, be incompatible with shrink-wrap software designed to comply with the ARC Standard Specification. Processing systems implementing R3000-compatible or R4000-compatible coprocessors 0 and which introduce coprocessor 0 hazards beyond those of the R3000 or the R4000, respectively, will, in general, be incompatible with shrink-wrap operating environments designed to comply with the ARC Standard Specification.

3.8 Peripheral Attachment Subsystems (I/O Bus)

This section describes the requirements of the peripheral attachment subsystems (I/O bus). Consequences that can be expected as a result of noncompliance with these requirements are also discussed.

3.8.1 Requirements

Each different bus type must have a unique type identifier. System firmware supports a set of standard device I/O functions independent of the I/O topology. Each I/O topology must provide means for low-level device-specific services necessary to implement the high-level device-independent functions. In addition, each topology must define the base functionality and external interfaces for each of the system elements. Addenda specifying these requirements for the EISA bus are separate from this document. Products requiring other buses (or no external bus at all) can either assume one of these models or can develop similar models for their respective characteristics.

Rationale – This standard does not prescribe a specific peripheral attachment subsystem as the required I/O bus. Rather, this standard defines a framework that may incorporate multiple I/O buses to meet a wide range of market needs. In addition to the base functionality described in this document, I/O topology-specific requirements are defined in separate addenda. These requirements allow systems sharing the same I/O topology to share a common set of peripherals and add-on devices. The commonality of devices and peripherals, along with common applications and operating systems, allows for full compatibility of products sharing an I/O topology. With this approach, a fully compatible set of multivendor products based on a given I/O topology, can be developed to target customers sharing similar requirements.

3.8.2 Related Consequences

Systems developed with buses other than those described here will require development of equivalent device I/O services, device functionality and interfaces as appropriate for such I/O topology. Systems with dissimilar I/O buses will generally not be able to share add-on devices, peripherals or associated functions such as drivers, diagnostics and configuration capabilities.

Platform Firmware 4

This chapter describes in detail all basic firmware functions and data structures.

4.1 Firmware Conventions

The ARC Specification defines a set of firmware functions that must be provided by all ARC platforms. The interfaces to these functions are defined in the sections that follow, using ‘C’ language conventions to specify the number and type of arguments.

4.1.1 Calling Procedures

Firmware functions are called indirectly through a transfer vector. The transfer vector is described in Section 4.3.7, “The Firmware Function Vector,” on page 96.

The arguments to individual firmware functions are defined in the sections below. However, certain conventions apply to all functions. These include:

- Where a function takes a memory address as an argument (either directly or as a component of a structure), the address must be a kseg0/kseg1 address.
- Where an element of a structure is defined by an enumerated type, the size of the corresponding element of the structure is 32 bits, unless expressly stated otherwise.

In the specifications that follow, the notation follows ANSI ‘C’ conventions. Note that the names given to the functions, arguments and standard data structures are given for descriptive convenience and are not themselves part of the specification.

A standard 'C' prototype is shown for each function or procedure, along with any associated data structures or constants. The following conventions are used:

- All variables are of type ***CHAR***, ***SHORT***, ***LONG***, ***UCHAR***, ***USHORT***, and ***ULONG***. Each type has the following characteristics:

```
CHAR        8 bits     signed
SHORT       16 bits    signed
LONG        32 bits    signed
UCHAR       8 bits     unsigned
USHORT      16 bits    unsigned
ULONG       32 bits    unsigned
```

- Variable-length strings are always terminated with a null (0x00) character.
- The length of a variable-length string is the number of characters in the string excluding the terminating null character.
- In this specification, NULL is equivalent to 0.

The functions provided by the firmware interface are assumed to be operating in a single processor. For multiprocessor systems, during the reset state a processor must be chosen to run the platform firmware. All other processors must remain quiescent while the system is in the firmware state. The method for determining the processor to run the platform firmware is implementation-dependent.

Parameter Passing

Arguments are passed to standard system functions following the conventions defined by the MIPS 'C' Language compiler.

Status Codes

This specification uses status codes as defined by international standard ISO/IEC 9945-1 (POSIX). Each status code mnemonic from that standard is assigned a specific numeric code in the table below. The table below defines the status codes that must be available. Additional status codes are allowed, but are implementation-specific.

Table 4-1 Status Codes

Mnemonic	Status Code	Meaning
[ESUCCESS]	0	No error
[E2BIG]	1	Argument list too long
[EACCES]	2	Permission denied
[EAGAIN]	3	Resource temporarily unavailable
[EBADF]	4	Bad file descriptor
[EBUSY]	5	Resource busy
[EFAULT]	6	Bad address
[EINVAL]	7	Invalid argument

Table 4-1 (continued)

Mnemonic	Status Code	Meaning
[EIO]	8	Input/output error
[EISDIR]	9	Is a directory
[EMFILE]	10	Too many open files
[EMLINK]	11	Too many links
[ENAMETOOLONG]	12	Filename too long
[ENODEV]	13	No such device
[ENOENT]	14	No such file or directory
[ENOEXEC]	15	Execute format error
[ENOMEM]	16	Not enough space
[ENOSPC]	17	No space left on device
[ENOTDIR]	18	Not a directory
[ENOTTY]	19	Inappropriate I/O control operation
[ENXIO]	20	Media not loaded
[EROFS]	21	Read-only file system

4.1.2 Memory Utilization

While in the firmware state, memory is organized into regions of an integral number of memory pages of 4,096 bytes. Platform firmware maintains a list of memory descriptors which describe all memory regions.

Note – A page size of 4,096 bytes is used as a standard unit of measure for memory by platform firmware but system software may use any page size supported by the processor.

Memory regions are classified depending on how they are used. There is an ***ExceptionBlock*** region which is used for exception vectors, fast trap handlers, etc. There is a ***SystemParameterBlock*** region which contains the firmware transfer vector.

Platform firmware may allocate ***FirmwareTemporary*** or ***FirmwarePermanent*** memory regions. Firmware regions are used only by the firmware. ***FirmwareTemporary*** regions need not be preserved when the system is in the program state, while ***FirmwarePermanent*** regions should be preserved by the system when it is in the program state.

A ***BadMemory*** region is a region of memory that failed memory diagnostics.

A ***FreeMemory*** region is a region of good memory not assigned to any particular purpose.

A ***LoadedProgram*** region is a region of memory into which the firmware has loaded a program as part of a software load operation. A ***FreeContiguous*** region of memory is a region immediately below and contiguous to a ***LoadedProgram*** region. The ***FreeContiguous*** region contains the loaded program's stack area.

The ***ExceptionBlock*** and ***SystemParameterBlock*** regions have specified addresses. The ***ExceptionBlock*** region begins at physical address 0x0 and the ***SystemParameterBlock*** region begins at physical address 0x1000. All other memory regions may begin and end at any page with the restriction that regions may not overlap.

While in the firmware state, the following memory requirements must be met:

- Memory regions must lie wholly within or without the kseg0/kseg1 processor address space. That is, it may be necessary to split a large region of memory into two regions, one inside and one outside the kseg0/kseg1 address space.
- At least one 4-megabyte ***FreeMemory*** region in kseg0/kseg1 must be available.

4.1.3 Stack and Data Addressability

ARC platform firmware loads program code into the top of a kseg0/kseg1 ***FreeMemory*** region of at least 4 megabytes in size. The ***FreeMemory*** memory region is then replaced with a ***LoadedProgramRegion*** and a ***FreeContiguous*** region, with the former immediately following the latter. The ***LoadedProgramRegion*** is allocated so as to contain the program and data portions of the program as specified by the program object file. The ***FreeContiguous*** region is used as the stack area for the loaded program. The initial stack pointer for the loaded program is set to point to the first byte following the ***FreeContiguous*** region.

4.1.4 Object Formats

The format for programs loadable by platform firmware is a subset of the MIPS COFF format, which is defined in the *MIPS Assembly Language Programmer's Guide,* Chapter 9, "Object File Format." Specifically, the following requirements for the executable image must be met:

- The File Header Magic Field must be MIPSELMAGIC 0x162 (little endian).
- The File Header Flags must have F_EXEC set (file is executable). The only other flags which can be set are F_LNNO (line numbers stripped) and F_LSYMS (local symbols stripped).
- The Optional Header Magic Field must contain the value OMAGIC 0407 (octal) impure format. (Text is not write-protected or shareable; the data segment is contiguous with the text segment.)
- Global pointers must not be used.
- Shared Libraries must not be used.
- A loadable image must include fixup information unless it is entirely position independent, that is, can be based at any virtual address.
- Only local relocations are supported, i.e. external relocations (based on a symbol table) are not supported.

4.2 The Firmware Environment

The ARC Spec defines a variety of blocks and data structures which partially define the execution environment visible to loaded programs when the system is in the firmware state. These data structures are discussed below.

4.2.1 Exception Block

The first page (4,096 bytes) of physical memory is reserved for exception processing. The MIPS hardware architecture requires that memory exist at physical page 0. Specific usage of page 0 across different MIPS processors is defined in the *MIPS R-Series Architecture Reference Manual.*

Note – It is required that physical memory exist for all of page 0. Hardware implementations that try to optimize memory system design by having the required 8 megabytes in one block somewhere above page 0 and only a single word register at each of the exception locations are not acceptable.

4.2.2 System Parameter Block

ARC platform firmware builds System Parameter Block (SPB) during system initialization. The structure of the System Parameter Block is defined in the figure below.

Field		Address
SPB Signature		0x1000
SPB Length		0x1004
Revision	Version	0x1008
Pointer to Restart Block		0x100C
Pointer to Debug Block		0x1010
GEVector		0x1014
UTLBMiss Vector		0x1018
Firmware Vector Length		0x101C
Pointer to Firmware Vector		0x1020
Private Vector Length		0x1024
Pointer to Private Vector		0x1028
Adapter Count		0x102C
Adapter 0 Type		0x1030
Adapter 0 Vector Length		0x1034
Pointer to Adapter 0 Vector		0x1038
⋮		0x103C

```
typedef struct {

  ULONG              SPBSignature;
  ULONG              SPBLength;
  USHORT             Version;
  USHORT             Revision;
  RESTARTBLOCK       *RestartBlock;
  DEBUGBLOCK         *DebugBlock;
  void               *GEVector;
  void               *ULTBMissVector;
  ULONG              FirmwareVectorLength;
  FIRMWAREVECTOR     *FirmwareVector;
  ULONG              PrivateVectorLength;
  void               *PrivateVector;
  ULONG              AdapterCount;
struct {
  ULONG              AdapterType;
  ULONG              AdapterVectorLength;
  void               *AdapterVector
} ADAPTERS []
}SPB;
```

Definition 4-1: SPB

Figure 4-1 System Parameter Block Structure

The System Parameter Block begins at physical address 0x1000.

SPBSignature must contain the value 0x53435241.

Version and **Revision** contain the values specified by the ARC Spec on which the implementation is based. For implementations based on this version of the specification, the value of **Version** is 1 and the value of **Revision** is 2. Subsequent releases of the ARC Spec will increase the **Version** by 1 and reset **Revision** to zero when major changes to the specification are made. Minor changes in the specification will be indicated by increasing the value of **Revision** alone.

Note – Revision is interpreted in units of one hundredths. A specification 1.23 would have a Version value of 1 and a Revision value of 23. This same notation applies to all instances of Version/Revision in other data structures.

SPBLength indicates the length of the System Parameter Block in bytes.

RestartBlock contains a pointer to a Restart Block. Operating systems may implement the capability to restart the system after a power disruption, using information in the Restart Block. If restart is not supported, this pointer is null. The structure of the Restart Block is defined below.

GEVector and **UTLBMissVector** are pointers to functions that handle general exceptions and UTLB miss exceptions. System firmware initializes these entries to point to the firmware's own handlers for these exceptions. Loaded programs may replace these values with their own handlers provided certain specific constraints are met. These constraints are addressed in Section 4.5, "Interrupts and Exceptions," on page 100.

DebugBlock may contain a pointer to a debug block. Its contents are not defined by this specification. Operating-system environments may support operation of the system under control of a debugger, and the debug block may be used for this purpose. If no debug block exists, the value of this pointer must be NULL.

FirmwareVectorLength contains the length of the firmware vector in bytes.

FirmwareVector contains the address of the firmware call vector.

PrivateVectorLength contains the length of a private firmware call vector in bytes.

PrivateVector contains the address of the private call vector. Additional vendor or implementation specific functions may be accessed via the private call vector.

AdapterCount contains the number of additional adapter-specific firmware vectors. Adapter-specific firmware vectors may be defined by addenda to this specification. Adapter-specific vector descriptions follow **AdapterCount**. Each descriptor consists of three fields: **AdapterType**, a unique numeric value assigned to the adapter by the relevant addenda; **AdapterLength**, the length of the adapter call vector in bytes; and **AdapterVector**, the address of the adapter call vector, which is a pointer to an array of vectors. These vectors are not in the SPB, just the pointers for the various buses.

4.2.3 Restart Block

A Restart Block provides a means of returning to the program state following power-on or system reset, without the need to “reboot” the platform. The structure of the Restart Block is defined by the Figure 4-2.

RSTBlock Signature	
RSTB Length	
Revision	Version
Pointer to Next RBlock	
Restart Address	
Boot Master ID	
Processor ID	
Boot Status	
Checksum	
Save Area Length	
Saved State Area	

Figure 4-2 RestartBlock

```
typedef struct {
  ULONG              RSTBSignature;
  ULONG              RSTBLength;
  USHORT             Version;
  USHORT             Revision;
  RESTARTBLOCK       *NextRSTB;
  void               *RestartAddress;
  ULONG              BootMasterID;
  ULONG              ProcessorID;
  ULONG              BootStatus;
  ULONG              Checksum;
  ULONG              SaveAreaLength;
  ULONG              SavedStateArea[];
} RESTARTBLOCK;
```

Definition 4-2: RESTARTBLOCK

RSTBSignature identifies a valid block as a Restart Block; it must have the value 0x42545352.

RSTBLength contains the overall length of the Restart Block in bytes.

Version and **Revision** contain values indicating the version and revision of the code that established the Restart Block.

NextRSTB points to another Restart Block. In multiprocessor systems, each processor has a Restart Block. If this is the last block, or the only block, this value is null.

RestartAddress specifies the address of a restart entry point in the operating system. It is filled in by operating system if it supports restart capabilities; otherwise, it is set to NULL.

BootMasterId identifies the processor controlling the boot process.

ProcessorID identifies the processor to which this Restart Block pertains. The contents of **BootMasterId** and **ProcessorId** correspond to the entries contained in the **Key** elements of the component structures for the corresponding processors in the configuration data structure. (See Section 4.2.5, “System Configuration Data,” on page 57).

BootStatus reflects the current state of processor activity and is defined below.

SavedStateArea contains operating-environment dependent-state information.

Checksum contains a value such that the sum of all elements of a Restart Block, including the saved state area, is 0. **Checksum** is originally computed when a Restart Block is written. In the event of a restart, a Restart Block is considered valid only if the checksum computed at the time of the restart is zero.

BootStatus is a 32-bit container with bits 0 through 6 defined by Figure 4-3. Bits 7 through 31 are reserved and must be 0. All status bits of **BootStatus** are initially set to zero, except for **ProcessorReady** which is set to one if the processor is available for use. In the discussion below, *set* means given a value of 1, while *reset* or *cleared* means assigned a value of 0.

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Processor Ready	Power Fail Finished	Power Fail Started	Restart Finished	Restart Started	Boot Finished	Boot Started

Figure 4-3 Boot Status Bits

BootStarted is set when firmware begins the normal boot process and is cleared by the loaded program. If this bit is already set when a boot process is begun, automatically initiated boot attempts are aborted. This is to prevent the possibility of an infinite loop during boot.

BootFinished is cleared by the firmware when it begins a bootstrap process and is set immediately before transferring control to the loaded program.

RestartStarted is set when the firmware begins a restart process and is cleared by the restart code that receives control.

RestartFinished is cleared when the firmware begins a restart operation and is set by the code that receives control.

PowerFailStarted is set by the power-fail state-save routine when it begins execution.

PowerFailFinished is set by the state-save routine when all state has been saved.

ProcessorReady is set by firmware when the processor has successfully completed power-on diagnostics and is ready to run system software.

Restart Procedure

For platforms and operating systems that support power-fail restart, the power-fail/power-on procedure is as follows:

1. When a power-fail occurs, the operating system passes control to code which sets the **PowerFailStarted** bit, saves the state of the machine in the **SavedStateArea**, writes the **RestartAddress**, sets the **PowerFailFinished** bit, and then computes and writes the **Checksum**.
2. When a reset occurs, the firmware checks for a valid Restart Block by looking for a non-null pointer in the **SystemParameterBlock** and a correct **RestartBlockSignature** and **Checksum** in the Restart Block.

3. If a valid Restart Block is found, the **BootStatus** bits are checked.
4. If both **PowerFailStarted** and **PowerFailFinished** are set and **RestartStarted** is clear, **RestartStarted** is set, and control is passed to the address in **RestartAddress**. This code will restore the saved state, set **RestartFinished**, and return control to the operating system.
5. If any of the required conditions are not met, the firmware reverts to a normal reset sequence.

4.2.4 Environment Variables

Certain system parameters are maintained by platform firmware in global system environment variables. Standard environment variables specify default values used for initializing the system, loading software, and recording the system time zone.

Environment variables are represented by strings of the form ***name=value***, where name is the ***name*** of the variable and ***value*** is its corresponding value. Environment variable names are not case sensitive, but their values are. Multiple values may be stored in environment variables, each value being separated from the previous value by a semicolon and the entire string being null terminated.

Some global environment variables may be maintained in non-volatile memory. Setting such a variable causes the value in non-volatile memory to be updated.

Additional environment variables beyond those specified here are allowed, but are implementation specific.

The values of all visible environment variables are included in a program's environment table (see Section 4.4, "Loaded Program Conventions," on page 97) when it is loaded. Modification of this table by the program has no effect on the global values maintained by system firmware. Likewise, modification of the global environment table by the **SetEnvironmentVariable** function has no effect on the contents of the program's environment table. Similarly, the **GetEnvironmentVariable** function returns the value maintained by system firmware, not the value in the program's environment table.

Console Initialization Environment Variables

ConsoleIn	The default pathname for the console input device.
ConsoleOut	The default pathname for the console output device.

When the system is initialized, platform firmware opens the device specified by **ConsoleIn** as the console input device and the device specified as **ConsoleOut** as the Console output device.

Software Loading Environment Variables

SystemPartition	The default path for the system partition.
OSLoader	The default path for an operating-system loader program.
OSLoadPartition	The default pathname of the partition containing the program to be loaded by the operating-system loader.
OSLoadFilename	The default filename of the program the operating system loader is to load.
LoadIdentifier	An ASCII string that may be used to associate an identifier with a set of load parameters.
OSLoadOptions	The default options to be passed to the operating system loader program.
AutoLoad	The default setting for automatic operating system load. If not set to "Yes," then it is assumed to be set to "No."

The **AutoLoad** environment variable controls the default load action of platform firmware. If **AutoLoad** equals yes, firmware will initiate an automatic load operation. If **AutoLoad** does not equal yes, or if **AutoLoad** does not exist as an environment variable, firmware enters the interactive firmware mode.

The **SystemPartition**, **OSLoader**, **OSLoadFilename**, **OSLoadPartition**, and **OSLoadOptions** environment variables specify the default arguments passed to a program loaded by system bootstrap firmware. **LoadIdentifier** may contain an identifier or name by which the set of values in the load-related environment variables can be referred to.

Some implementations may support the concept of alternate automatic-load paths so that an automatic software load operation can be directed to try more that one load path. In this case, more than one value may be stored in the **SystemPartition**, **OSLoader**, **OSLoadFilename**, **OSLoadPartition**, **OSLoadOptions**, and **LoadIdentifier** environment variables, with values separated by semicolons. When attempting an automatic software load, the load is attempted with the first set of values, then second set of values, and so on. The above sets of environment variables are traversed in parallel; that is, the first entry (value) of each environment variable is associated with the first entry of all of the other environment variables; the second entry of one, with the second entry of all of the others, and so on. An environment variable need not have a value for every alternate load. This case is indicated by an empty value string, that is, adjacent semicolons with no intervening text. The **LoadIdentifier** environment variable is provided so that each set of values may be given a name by which a set of values may be consistently referred to across different environments (e.g. operating-system loaders or installers, firmware-based utilities, etc.). For NT, OSLoadFilename points to the root of the NT tree and not the kernel file.

Time Zone Environment Variable

TimeZone	The local time zone.

The **TimeZone** environment variable contains a specification of the local time zone. The time zone information is as specified by international standard ISO/IEC 9945-1 (POSIX) for the POSIX **TimeZone** environment variable using the *std offset dst offset,rule* form.

Firmware Search Path Environment Variable

FWSearchPath	Specifies the set of system partitions to search looking for loadable system components

FWSearchPath defines the set of system partitions in which system firmware and loaded programs should search for loadable system components. The value of **FWSearchPath** is one or more pathnames for system partitions separated by semicolons. System firmware searching for loadable firmware or firmware upgrades should search the system partitions identified in the value string for candidate components. System installation and configuration utilities should similarly look at each of the partitions listed as possible suppliers of firmware components, drivers, and firmware upgrades.

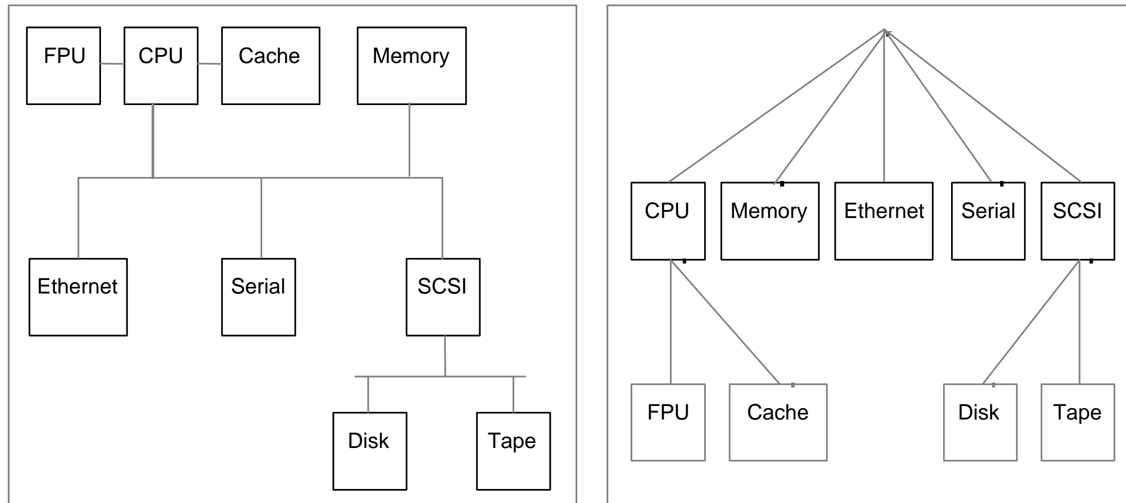
4.2.5 System Configuration Data

ARC-platform firmware maintains a collection of configuration data containing component and topology information about the system. This body of configuration data is referred to collectively as the system configuration data structure. When the platform is in the firmware state, this data may be accessed using the functions defined in Section 4.3.3, “Configuration Functions,” on page 79.

Rationale – A goal of the ARC architecture is to simplify system software installation and loading. By requiring that platforms maintain configuration information and provide standard functions to access this information, the need for human involvement in the system configuration process is reduced.

Note – The means by which ARC-platform firmware obtains the configuration information is not part of this specification, nor does this specification require that the configuration information be complete. In some instances, full configuration information may not be directly available to the platform firmware, but must be obtained through the execution of a configuration utility program or, in some instances, must be obtained by querying the operator. Platform implementers are encouraged to utilize the **AddChild** and **SaveConfiguration** routines defined in Section 4.3.3, “Configuration Functions,” on page 79 to incorporate this information into the platform configuration data structure, but this may not always be possible, and, in some instances, the operating system may be obliged to collect this information itself.

The system configuration data structure is organized as a tree of component structures. Each node contains descriptive information about a system component. Connections between nodes indicate connections or dependencies between components, referred to as system *topology*. For example, consider the system block diagram shown in Figure 4-4(a) and the corresponding configuration data structure in Figure 4-4(b).



(a) System Block Diagram

(b) System Configuration Data Structure

Figure 4-4 Example System

Rationale – The representation of the system configuration as a tree structure is a compromise between utility and simplicity. A tree structure is open ended and can be easily traversed one node at a time. Processing configuration data one node at a time is seen as desirable both for firmware clients and for the firmware itself.

While much of the configuration data serves mainly to identify components, paths between nodes of the tree also present a reasonable representation of dependencies or relationships between components.

On the other hand, a tree structure cannot adequately represent all the connections or dependencies of systems with crossbar buses or even systems with dual-ported peripherals connected to two different controllers. However, a tree structure is adequate for most ARC systems, and complicating the data structure to deal with all possible physical configurations seems unwarranted. An adequate representation of a system configuration does not require that every topological feature of the system be indicated.

The tree structure of the configuration data structures not exposed directly to firmware clients and the actual internal representation of this structure are not defined by this specification. Component data may be accessed one node at a time using the configuration query functions defined in Section 4.3.3, “Configuration Functions,” on page 79. The format of the COMPONENT data structure is shown Figure 4-5.

31 16	15 0	
Class		000
Type		004
Flags		008
Revision	Version	00C
Key		010
Affinity Mask		014
Configuration Data Size		018
Identifier Length		01C
Pointer to Identifier		020

```
typedef struct {
  COMPONENT_CLASS    Class;
  COMPONENT_TYPE     Type;
  COMPONENT_FLAGS    Flags;
  USHORT             Version;
  USHORT             Revision;
  ULONG              Key
  ULONG              AffinityMask;
  ULONG              ConfigurationDataSize
  ULONG              ;
  CHAR               IdentifierLength;
} COMPONENT;         *Identifier;
```

Definition 4-3: COMPONENT

Figure 4-5 COMPONENT Data Structure

Component Class and Type

Seven classes of components are defined. Within most classes, a variety of types is defined. The general properties of any component may be inferred from the component's class and type. The class and type of a component are found in the Class and Type fields of the component structure, respectively.

The following component classes are defined:

```
typedef enum{
   SystemClass              = 0,
   ProcessorClass           = 1,
   CacheClass               = 2,
   AdapterClass             = 3,
   ControllerClass          = 4,
   PeripheralClass          = 5,
   MemoryClass              = 6
}COMPONENT_CLASS;
```

Definition 4-4: COMPONENT_CLASS

SystemClass: The system class is the class of the root configuration component. It serves to identify the type of system and to act as the root of the configuration tree. Within the system class, only one type is defined:

```
typedef enum{
   ARC                      = 0,
   :
} COMPONENT_TYPE;
```

Definition 4-5: COMPONENT_TYPE

ProcessorClass: The processor class indicates a computing element, either a central processing unit, or a floating point unit. Within the processor class, the following types are defined:

```
typedef enum{
   :
   CPU                      = 1,
   FPU                      = 2,
   :
} COMPONENT_TYPE;
```

Definition 4-6: COMPONENT_TYPE (continued)

CacheClass: The cache class indicates a memory cache component. Within the cache class, the following types are defined:

```
typedef enum{
   :
   PrimaryICache            = 3,
   PrimaryDCache            = 4,
   SecondaryICache          = 5,
   SecondaryDCache          = 6,
   SecondaryCache           = 7,
   :
} COMPONENT_TYPE;
```

Definition 4-7: COMPONENT_TYPE (continued)

AdapterClass: The adapter class indicates a peripheral attachment adapter. An adapter is used to provide access to a heterogeneous collection of peripheral controllers. Within the adapter class, the following types are defined:

```
typedef enum{
   :
   EISAAdapter              = 8,
   TCAdapter                = 9,
   SCSIAdapter              = 10,
   DTIAdapter               = 11,
   MultiFunctionAdapter     = 12,
   :
} COMPONENT_TYPE;
```

Definition 4-8: COMPONENT_TYPE (continued)

The type codes **EISAAdapter**, **TCAdapter**, **SCSIAdapter**, and **DTIAdapter** denote an EISA bus adapter, a TURBOchannel bus adapter, a SCSI bus adapter, and an AccessBus adapter, respectively. The **MultiFunctionAdapter** code is used to indicate an adapter that is not a standard adapter type. For example, a combination module containing both a disk and network controller would have a type of **MultiFunctionAdapter**.

ControllerClass: The controller class indicates a peripheral controller. A controller is used to provide access to a homogeneous collection of peripherals. Within the controller class, the following types are defined:

```
typedef enum{
   :
   DiskController           = 13,
   TapeController           = 14,
   CDROMController          = 15,
   WORMController           = 16,
   SerialController         = 17,
   NetworkController        = 18,
   DisplayController        = 19,
   ParallelController       = 20,
   PointerController        = 21,
   KeyboardController       = 22,
   AudioController          = 23,
   OtherController          = 24,
   :
} COMPONENT_TYPE;
```

Definition 4-9: COMPONENT_TYPE (continued)

The controller type code ***OtherController*** denotes a controller type not otherwise explicitly defined.

PeripheralClass: The peripheral class indicates a peripheral device. Within the peripheral class, the following types are defined:

```
typedef enum{
   :
   DiskPeripheral          = 25,
   FloppyDiskPeripheral    = 26,
   TapePeripheral          = 27,
   ModemPeripheral         = 28,
   MonitorPeripheral       = 29,
   PrinterPeripheral       = 30,
   PointerPeripheral       = 31,
   KeyboardPeripheral      = 32,
   TerminalPeripheral      = 33,
   OtherPeripheral         = 34,
   LinePeripheral          = 35,
   NetworkPeripheral       = 36,
   :
} COMPONENT_TYPE;
```

Definition 4-10: COMPONENT_TYPE (continued)

The peripheral type code ***OtherPeripheral*** denotes a peripheral of a type not otherwise explicitly defined.

MemoryClass: The memory class indicates a memory unit. A memory unit implements a range of memory cells for an ARC system. Within the memory class, only one type is defined:

```
typedef enum{
   :
   MemoryUnit              = 37
} COMPONENT_TYPE;
```

Definition 4-11: COMPONENT_TYPE (continued)

The system configuration, and therefore the configuration data structure, may contain multiple instances of these component classes, except for the system class itself. There may be only one instance of a system class component in the configuration.

Note – If new component classes or types are added to this specification, new classes will be assigned class code values beginning with 7, and new types will be assigned type code values beginning with 38. Note that because new type codes may be added in the future, it is invalid to assume that components of a given class will necessarily have type codes clustered together.

Component Flags

Seven component flags are defined:

```
typedef enum{
   Failed                  = 1,
   ReadOnly                = 2,
   Removable               = 4,
   ConsoleIn               = 8,
   ConsoleOut              = 16,
   Input                   = 32,
   Output                  = 64
} COMPONENT_FLAG;
```

Definition 4-12: COMPONENT_FLAG

Failed: This flag bit is set if the component failed diagnostic testing. If this bit is set, the device will generally be disabled. However, this specification imposes no diagnostic testing requirements, so it is invalid to assume that a component is operating correctly if this bit is not set.

Readonly: This flag bit is set if peripheral component may be read but not written.

Removable: This flag bit is set if the component is a removable media peripheral such as a tape or a floppy disk.

ConsoleIn: This flag bit is set if the component is a controller or peripheral supported by the platform firmware as a system console input device.

ConsoleOut: This flag bit is set if the component is a controller or peripheral supported by the platform firmware as a system console output device.

Input: This flag bit is set if the component is a controller or peripheral used as an input device.

Output: This flag bit is set if the component is a controller or peripheral used as an output device.

Table 4-2 indicates which flag bits are appropriate for components of various class and types. Note the following:

- ***Fail*** is appropriate for all components.
- ***Readonly*** and ***Removable*** are appropriate only for peripheral classes.
- ***ConsoleIn*** and ***ConsoleOut*** may be set only if ***In*** and ***Out***, respectively, are set as well.

Table 4-2 Component Flag Bit Usage

Class/Type	Flag Bits						
	Fail	**RO**	**Rem**	**CIn**	**COut**	**In**	**Out**
System/ARC	√						
Processor/*	√						
Cache/*	√						
Memory/*	√						
Adapter/*	√						
Controller/Disk	√					√	√
Controller/Tape	√					√	√
Controller/CDROM	√					√	
Controller/WORM	√					√	√
Controller/Serial	√			√	√	√	√
Controller/Network	√					√	√
Controller/Display	√				√		√
Controller/Printer	√						√
Controller/Pointer	√					√	
Controller/Keyboard	√			√		√	
Controller/Audio	√					√	√
Controller/Other	√			√	√	√	√
Peripheral/*	√	√	√	√	√	√	√

Fail = Failed CIn = ConsoleIn Out = Output
RO = ReadOnly COut = ConsoleOut
Rem = Removable In = Input

Component Version and Revision

The component **Version** and **Revision** fields of the component data structure indicate the version and revision level of the release of the ARC Spec on which the component data structure is based.

Component Key

Associated with each component is a **Key** field. The meaning of the **Key** field depends on the class and type of the component.

System: The value of the **Key** field must be zero.

Processor/CPU: The **Key** field must contain the processor number.

Processor/FPU: The **Key** field must contain the FPU coprocessor number.

Cache: The **Key** field contains the following:

31 24	23 16	15 0
Refill Size	Line Size	Cache Size

Refill Size = number of cache lines transferred in a cache refill
Line Size = $\log_2$(line size in bytes/tag)
CacheSize = $\log_2$(cache size in 4,096 pages)

Memory: The **Key** field contains the memory module number. Memory module numbers are platform specific and are not defined by this specification.

Adapter: The **Key** value of an adapter is interpreted in the context of its parent. If the parent is an adapter covered by an addendum, **Key** is defined by the addendum. Otherwise, **Key** contains a zero based instance count of the parents adapter children. That is, if the parent has three adapter children, the first would have a **Key** value of zero, the second a value of one, and the third a value of two.

Controller: The **Key** value of a controller is interpreted in the context of its parent. If the parent is an adapter covered by an addendum, **Key** is defined by the addendum. If the parent is a SCSI adapter, **Key** contains the SCSI ID of the controller. Otherwise, **Key** contains a product specific value.

Peripheral: The **Key** value of a peripheral is interpreted in the context of its parent. If the parent is a controller descended from an adapter covered by an addendum, the value of **Key** is defined by the addendum. If the parent is a SCSI adapter, the value of **Key** is the SCSI unit number of the peripheral. Otherwise **Key** contains a zero based instance count of the children of the parent controller.

Affinity Mask

Components may be organized into groups using the **AffinityMask** field. The contents of **AffinityMask** are viewed as a bitvector of 32 bits, where each bit corresponds to an affinity group. Components that have the same bit set in **AffinityMask** share some platform specific affinity, perhaps superior performance when used together. In a multiprocessor system, some adapters or controllers may only be accessible from particular processors and the **AffinityMask** value is used to associate the adapters or controllers with the processors that can access them. There is no restriction on the number of affinity groups to which a component may belong.

For memory units, the **AffinityMask** field is explicitly defined to indicate that in some sense, the memory is *nearer* to processors in the same affinity group than it is to processors in other affinity groups.

Configuration Data Size

Associated with each component is optional, component-specific data. The number of bytes of this additional configuration data is specified in the **ConfigurationDataSize** field. If the value of this field is zero, no additional data is available. This additional configuration data may be accessed using the **GetConfigurationData** routine described in Section 4.3.3, "Configuration Functions," on page 79. The format of this additional configuration data is described in Section 4.2.6, "Additional Configuration Data," on page 67.

Component Identifier

Associated with each component is a class and type specific component identifier string. The length of the string in bytes is in the field **IdentifierLength** and the address of the string is in the field **Identifier**. If present, the identifier string contains ISOLatin1 characters. Valid values of the identifier string for each class and type of component are:

System: The identifier string is the registered identifier of the platform vendor followed by a hyphen and the model name by which the product is known.

Processor/CPU: The identifier consists of a string in the form <*CPU vendor ID*>-<*product ID*>. Allowed product ID's are **R3000**, **R4000** and **R4400**.

Processor/FPU: The identifier consists of a string in the form <*FPU vendor ID*>-<*product ID*>. Allowed processor ID's are **R3010** and **R4000FPU**.

Cache: No identifier is defined. The value of **IdentifierLength** must be zero.

Memory: No identifier is defined. The value of **IdentifierLength** must be zero.

Adapter/EISA: The identifier string must be ***EISA***.

Adapter/TC: The identifier string must be ***TC***.

Adapter/Multifunction: If the adapter is a child of an adapter for which there is an addendum, the identifier is specified by the addendum. Otherwise the identifier is product specific.

Controller/SCSI: The identifier is the concatenation of the vendor and the product name fields returned by a SCSI Inquiry command sent to the controller.

Controller/Other: If the controller is the child of an adapter for which there is an addendum, the identifier is specified by the addendum. Otherwise, the identifier is product specific.

Peripherals: No identifier is defined. The value of **IdentifierLength** must be zero.

System Topology Constraints

The configuration data structure must conform to the following rules:

- The root of the tree must be the system component.
- CPU components are children of the system component.
- Private caches are children of the processors to which they belong.
- Shared caches are peers of the processors among which they are shared. The affinity mask indicates which caches are associated with which processors.
- FPU components are always the children of CPU components.
- Memory components are children of the system component.
- Adapter components are children of either other adapter components or of the system component.
- Peripheral components are children of controller components.
- Buses connected embedded devices are represented by multifunction adapter components.
- Children of controllers are generally of the same peripheral type but not always. For example, a disk controller may have both ***DiskPeripheral*** and ***FloppyDiskPeripheral*** components as children.

4.2.6 Additional Configuration Data

All configuration data is stored in a structure called a Partial Resource List. This variable-length structure that can be used to describe the I/O Port addresses, interrupts, memory addresses, and DMA channels used by the device. In addition, this structure can contain vendor and product names and serial numbers. Other, free-form device-specific information can be included in the configuration data immediately after the Partial Resource List.

The Partial Resource List has the following form:

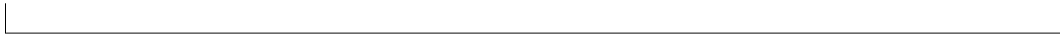
```
typedef struct _CM_PARTIAL_RESOURCE_LIST {
  USHORT Version;
  USHORT Revision;
  ULONG Count;
  CM_PARTIAL_RESOURCE_DESCRIPTO PartialDescriptors[1];
}CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;
```

Version and Revision contain the values specified by the ARC specification on which the implementation is based. Count is the number of Partial Resource Descriptors that are contained in the structure, starting with PartialDescriptor[0].

A Partial Resource Descriptor has the following definition:

```
//
// Defines the Type in the RESOURCE_DESCRIPTOR
//

typedef enum _CM_RESOURCE_TYPE {
 CmResourceTypeNull = 0, // Reserved
 CmResourceTypePort,
 CmResourceTypeInterrupt,
 CmResourceTypeMemory,
 CmResourceTypeDma,
 CmResourceTypeDeviceSpecific,
 CmResourceTypeVendor,
 CmResourceTypeProductName,
 CmResourceTypeSerialNumber
} CM_RESOURCE_TYPE;

//
// There can only be *1* DeviceSpecificData block. It must be
// located at the end of all resource descriptors in a full
//descriptor block.
//

typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
 UCHAR Type;
 UCHAR ShareDisposition;
 USHORT Flags;
 union {

 //
 // Range of port numbers, inclusive. These are physical, bus
 // relative.
 //

 struct {
 PHYSICAL_ADDRESS Start;
 ULONG Length;
 } Port;

 //
 // IRQL and vector.
 //

 struct {
 ULONG Level;
 ULONG Vector;
 ULONG Reserved1;
 } Interrupt;

 //
 // Range of memory addresses, inclusive. These are physical, bus
 // relative.
 //
```

```
struct {
PHYSICAL_ADDRESS Start; // 64 bit physical addresses.
ULONG Length;
} Memory;

//
// Physical DMA channel.
//

struct {
ULONG Channel;
ULONG Port;
ULONG Reserved1;
} Dma;

//
// Vendor string.
//

struct {
CHAR Vendor[12];
} Vendor;

//
// Product name string.
//

struct {
CHAR ProductName[12];
} ProductName;

//
// Serial Number string.
//

struct {
CHAR SerialNumber[12];
} SerialNumber;

//
// Device Specific information defined by the driver.
// The DataSize field indicates the size of the data in bytes. The
// data is located immediately after the DeviceSpecificData field
// in the structure.
//

struct {
ULONG DataSize;
ULONG Reserved1;
ULONG Reserved2;
} DeviceSpecificData;
} u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR, *PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```



Type is a CM_RESOURCE_TYPE which defines the type of resource which is described. ShareDisposition describes if the resource is exclusive to the device (e.g. local memory), exclusive to the driver (e.g. a DMA channel), or shareable (e.g. a shared interrupt line). The interpretation of Flags is determined by the resource type.

Each Port descriptor describes a range of I/O port (register) addresses, beginning at physical address Start and having a range of Length bytes. Each Interrupt descriptor describes a local bus interrupt line, of local interrupt level Level and vector Vector. Each Memory descriptor describes a range of memory addresses, beginning at physical address Start and having a range of Length bytes. Each DMA descriptor describes the use of DMA channel number Channel, at port location Port.

Note that the meaning of the words Interrupt, Level, Vector, Channel, and Port are bus dependent, and not all these parameters will be used for all buses.

The Vendor, ProductName, and SerialNumber descriptors contain zero terminated ASCII strings which identify the vendor, product name, and or serial number of the associated device. If the string is exactly 12 characters, the terminator can be omitted. Multiple descriptors can be used to build strings of longer than 12 characters.

Device specific data can be included directly after the Resource Descriptor List by including a resource descriptor of type CmResourceTypeDeviceSpecific. DataSize determines the size of the device-specific data, which starts at location PartialDescriptors[Count].

The defines for the ShareDisposition and Flags variables are as follows:

```
//
// Defines the ShareDisposition in the RESOURCE_DESCRIPTOR
//

typedef enum _CM_SHARE_DISPOSITION {
 CmResourceShareUndetermined = 0,
 CmResourceShareDeviceExclusive,
 CmResourceShareDriverExclusive,
 CmResourceShareShared
} CM_SHARE_DISPOSITION;

//
// Define the bit masks for Flags when type is CmResourceTypeInterrupt
//

#define CM_RESOURCE_INTERRUPT_LEVEL_SENSITIVE 0
#define CM_RESOURCE_INTERRUPT_LATCHED 1

//
// Define the bit masks for Flags when type is CmResourceTypeMemory
//

#define CM_RESOURCE_MEMORY_READ_WRITE 0
#define CM_RESOURCE_MEMORY_READ_ONLY 1
#define CM_RESOURCE_MEMORY_WRITE_ONLY 2

//
// Define the bit masks for Flags when type is CmResourceTypePort
//

#define CM_RESOURCE_PORT_MEMORY 0
#define CM_RESOURCE_PORT_IO 1
```

4.2.7 Devices, Partitions, Files, and Path Specifications

ARC systems may incorporate a variety of peripheral devices. Some devices are essential for ARC platforms, namely those needed to act as the system console and those from which software may be loaded.

Block storage devices (disks, floppy disks, CDROMs and WORMs) may be organized into partitions (see Section 3.6.2, “System Partition Formats,” on page 42). ARC firmware supports access to such devices in partitioned mode, where file addresses are relative to the beginning of a partition and are restricted to that partition; and in non-partitioned mode, where file addresses are relative to the beginning of the volume and may reference any part of the volume.

Block storage devices may also be file structured as well as partitioned. ARC firmware supports file structured access to FAT volumes or partitions, and to ISO9660 volumes. Support for additional types of file systems is allowed, but is implementation specific.

Path Specifications

Devices, device partitions, and files are denoted textually by path specifications. A *path specification* is an ISOLatin1 character string with the following format:

```
<path>              ::= <adapter>...<controller><peripheral>
                        [<partition>][<filepath>]
<adapter>           ::= <adapter mnemonic>(<key>)
<controller>        ::= <controller mnemonic>(<key>)
<peripheral>        ::= <peripheral mnemonic>(<key>)
<partition>         ::= partition([<number>])
<protocol>          ::= tftp() | ripl() | console(0) | console(1)
<filepath>          ::= <file system specific name>
```

Valid adapter, controller, and peripheral mnemonics are shown in Table 4-3.

Table 4-3 Path Mnemonics

Class	Type	Mnemonic
Adapter	EISA	eisa
Adapter	TC	tc
Adapter	SCSI	scsi
Adapter	DTI	dti
Adapter	MultiFunction	multi
Controller	Disk	disk
Controller	Tape	tape
Controller	CDROM	cdrom
Controller	WORM	worm
Controller	Serial	serial
Controller	Network	net
Controller	Display	video
Controller	Parallel	par
Controller	Pointer	point
Controller	Keyboard	key
Controller	Audio	audio
Controller	Other	other
Peripheral	Disk (rigid)	rdisk
Peripheral	FloppyDisk	fdisk
Peripheral	Tape	tape
Peripheral	Modem	modem
Peripheral	Monitor	monitor
Peripheral	Printer	print
Peripheral	Pointer	pointer
Peripheral	Keyboard	keyboard
Peripheral	Terminal	term
Peripheral	Line (comm)	line
Peripheral	Network	network
Peripheral	Other	other

Note – Although ARC platform firmware need not support input/output functions for all the adapter, controller, peripheral combinations shown in Table 4-3, some implementations may choose to do so, and for that reason, mnemonics are defined for all combinations.

The initial portion of a path specification, up to and including the peripheral specifier, specifies a top-down traversal of the system configuration data structure, uniquely identifying a peripheral device. This traversal is restricted to passing through adapter, controller, and peripheral nodes only.

For example, consider a platform with a single integral SCSI adapter, to which is connected a SCSI disk controller with a SCSI ID of 3, and one disk attached to the controller. The path specification for the disk would be:

```
scsi(0)disk(3)rdisk(0)
```

Following the peripheral specification, the optional partition specification serves to indicate that the device is to be accessed as a partitioned device where the <key> value indicates the partition number. If the <key> field is zero, or if the partition specifier is absent, the device is to be accessed as an unpartitioned device. If the device is not a block-structured device, the partition specifier must be absent.

For example, if the disk described in the preceding example had 4 partitions, the path specification for the first partition would be:

```
scsi(0)disk(3)rdisk(0)partition(1)
```

Following the optional partition specifier is an optional protocol specifier. For network devices, the ***tftp*** or ***ripl*** protocol specifier indicates which network bootstrap protocol is to be used for network based bootstraps. For console devices, the ***console*** protocol specifier indicates which console protocol is to be employed, ***console(0)*** indicating that the basic 8-bit console protocol is to be employed and ***console(1)*** indicating that the 16-bit Unicode console protocol is to be employed. Additional protocol specifiers for network and console devices, as well as other devices, may be available but are implementation specific.

The final portion of the path specification is the optional file specifier. The format of the file specifier portion is file-system specific.

If the partition in the preceding example is a FAT file system partition, then the path specification for the file ***\OS.DIR*** is:

```
scsi(0)disk(3)rdisk(0)partition(1)\OS.DIR
```

4.2.8 System Partition

Every source from which an ARC system may load programs must have a system partition. Primarily, items to be found in the system partition are programs such as operating-system loader and installer programs, configuration utilities, and so on. But the system partition may also include code that extends the platform's ROM-based firmware, program code that extends (or implements) ARC firmware device I/O functionality, or other programs or collections of data.

Rationale – The presence of a standard system partition is intended to be an enabling feature of ARC systems. The presence of a file-structured storage system during system installation time or system bootstrap time is a significant enabling feature. For example, during system installation, an operating system installer may copy vendor-provided HALs and device drivers, along with a platform independent operating system image to the system partition which at boot time, an operating system loader binds together and invokes. Alternatively an installer may construct and store an image of a configured operating system image in the system partition which is then booted directly. Either approach, as well as others, is possible.

Note – Note that at present, this specification does not define network-based, file-structured input/output as fully as for disk and other block storage devices. This limits network-based system partitions relative to disk and other block storage based system partitions. This asymmetry may be addressed in a future version of this specification.

The following requirements for the system partition must be met:

- For each device, the system partition must conform to the requirements established for system partitions on that device or class of devices, as set forth in Section 3.6.2, “System Partition Formats,” on page 42.
- Programs that are to be loaded from the system partition must conform to the object format requirements set forth in Section 4.1.4, “Object Formats,” on page 50.
- Program code that implements addendum-specific functionality must conform to the program conventions by the addendum.

Within the system partition, the following base directory tree is defined by this specification:

```
\OS
    \<Standard OS-specific identifier>
\<vendor>
\UTIL
```

Any files stored in the system partition that are associated with a specific operating system must be placed in the subdirectory for that operating system (for example, \OS\NT). The contents and structure of this directory are defined by the specifications for the operating system.

Any platform-related files that are specific to a system vendor (such as loadable extensions to system firmware or loadable extensions to device firmware) are placed in a \<vendor> subdirectory where <vendor> is the vendor’s ARC ID. This is the same ID as returned as the first component of the **GetSystemId** function. The contents and structure of this directory is defined by the vendor.

Standard utilities whose location must be known to platform firmware are placed in the \UTIL subdirectory.

Addenda to this specification may define additional branches off of the root. The content and structure of this directory is specified by the addendum. Adapter-specific branches defined by addenda have the root directory name:

```
\<adapter mnemonic>
```

where <adapter mnemonic> is the mnemonic assigned to that adapter for pathname specifications.

Pathnames referring to files in system partitions may be formed with either backslash (“\”) or forward slash (“/”) as path-component separators.

4.3 Standard Firmware Functions

The ARC Spec defines numerous functions that are callable by loaded programs. Functions are defined for loading programs, terminating programs, accessing the system configuration data, performing Input/Output, accessing global environment variables, obtaining a unique system identifier value, querying the system memory configuration, and requesting time and date data.

Except for the program termination functions, these functions may be called only if the system is in the firmware state, that is, the exception vectors that were in use when the firmware loaded the program are unchanged, all permanent and temporary firmware memory regions are unmodified, and no accesses have been made to system or device control or data registers by the program. (See Section 4.5, “Interrupts and Exceptions,” on page 100 for a limited escape from these restrictions.)

4.3.1 Program Loading

The following programloading functions are defined:

- **Load**, used to load programs into memory;
- **Invoke**, used to invoke a previously loaded program; and
- **Execute**, used to load and then automatically execute a program.

Load()

```
LONG Load(
       CHAR     *Path,
       ULONG    TopAddr,
       ULONG    *ExecAddr,
       ULONG    *LowAddr);
```

The **Load** function reads the program specified by **Path** into memory immediately below the memory address specified by **TopAddr. Load** stores the execution address of the program in ***ExecAddr** and the low address of the program in ***LowAddr**. The program to be loaded must be in MIPS COFF format as described in Section 4.1.4, “Object Formats,” on page 50, and is relocated as needed during the load process. **Load** sets the program’s bss region to 0.

Status codes returned by **Load** are:

`[ESUCCESS]`	The load operation was successful.
`[EIO]`	An I/O error occurred while the program was being loaded.
`[EMFILE]`	Too many file descriptors are in use.
`[ENAMETOOLONG]`	The filename portion of the path is too long.
`[ENODEV]`	Device portion of path is ill-formed or device doesn't exist.
`[ENOENT]`	The named file does not exist.
`[ENOEXEC]`	The file specified is not executable.
`[ENOMEM]`	There is not enough memory to load the specified program.

Invoke()

```
LONG Invoke(
        ULONG    ExecAddr,
        ULONG    StackAddr,
        ULONG    Argc,
        CHAR     *Argv[],
        CHAR     *Envp[]);
```

The **Invoke** function invokes a previously loaded program. **Invoke** calls the program whose execution address is specified by **ExecAddr**. Before calling the program, **Invoke** sets the stack pointer to **StackAddr**. The program is invoked with the argument count Argc, the argument list **Argv**, and the environment **Envp** as described in Section 4.4, "Loaded Program Conventions," on page 97. Unless one of the termination functions is explicitly called, an invoked program returns to the calling program.

Note – It is assumed that an invoked program returning to a caller has not corrupted the state of the platform firmware or the invoking program. Because any corruption of this state will probably cause the system to fail in an unpredictable manner, such returns should be used with caution.

Any files that are open when **Invoke** is called remain open for the invoked program. Any file descriptors closed by an invoked program are closed for all of the invoked program's ancestors.

If the invoked program returns, its status is returned by **Invoke**. If an invalid address is specified for **ExecAddr** or **StackAddr**, `[EFAULT]` is returned by **Invoke**.

Execute()

```
LONG Execute(
       CHAR     *Path,
       ULONG    Argc,
       CHAR     *Argv[],
       CHAR     *Envp[]);
```

The **Execute** function reads the program specified by **Path** into memory and then invokes it. The program is invoked with the argument count **Argc**, the argument list **Argv**, and the environment **Envp** as describe in Section 4.4, “Loaded Program Conventions,” on page 97. If the loaded program returns, control returns to the platform firmware, not to the caller.

Note – It is assumed that a loaded program returning to the platform firmware has not corrupted the state of the platform firmware. Because any corruption of this state will probably cause the system to fail in an unpredictable manner, such returns should be used with caution.

Execution of the **Execute** function causes the firmware to reinitialize its memory descriptors and memory regions as required. In particular, any existing ***LoadedProgram*** and ***FreeContiguous*** memory regions are returned to the list of ***FreeMemory*** regions before the program is loaded.

Any files that are open when **Execute** is called remain open for the invoked program. The program to be loaded must be in MIPS COFF format as described in Section 4.1.4, “Object Formats,” on page 50 and is *relocated* as needed during the load process.

If the invoked program returns, its status is returned; otherwise, the status codes returned by **Execute** are:

`[EIO]`	An I/O error occurred while the image was being loaded.
`[EMFILE]`	Too many file descriptors are in use.
`[ENAMETOOLONG]`	The filename portion of the path is too long.
`[ENODEV]`	Device portion of path is ill-formed or device does not exist.
`[ENOENT]`	The named file does not exist.
`[ENOEXEC]`	The file specified is not executable.
`[ENOMEM]`	There is not enough memory to load the specified image.

4.3.2 Program Termination

The ARC Spec defines five program termination functions:

- **Halt** exits the program and enters the halt state.
- **PowerDown** exits the program and powers down the system.
- **Restart** performs the equivalent of a power on reset.
- **Reboot** attempts to reboot the system.
- **EnterInteractiveMode** terminates the executing program and enters interactive mode.

None of these functions returns a value.

Halt()

```
VOID Halt(VOID);
```

The **Halt** function exits the program and enters a halt state. In this state, the system is quiescent and the only exit from this state is a reset or by an implementation specific exit.

PowerDown()

```
VOID PowerDown(VOID);
```

The **PowerDown** function is identical to the **Halt** function with the additional feature that, if a system is so equipped, power to the system is removed. If power removal is not supported, this function is identical to **Halt**.

Restart()

```
VOID Restart(VOID);
```

The **Restart** function performs the equivalent of a power-on reset.

If platform firmware detects the presence of a valid Restart Block, the firmware will resume execution at the location specified in the Restart Block. Note that this is performed without modifying any non-firmware memory regions.

If no valid Restart Block is found, normal power-on firmware processing is performed. If the value of **Autoload** is ***yes***, the firmware will perform the default system load sequence.

Reboot()

```
VOID Reboot(VOID);
```

The **Reboot** function attempts to reboot the system with the same parameters used for the previous system load sequence. If the parameters cannot be reproduced, the default system load sequence is performed.

Rationale – Interactive firmware may provide a means for the system operator to specify non-default parameters when the system is loaded. By retaining these parameters in FirmwarePermanent memory, the firmware is normally able to reproduce these parameters for subsequent invocations of Reboot. However, the contents of the FirmwarePermanent memory may be corrupted during operation of the program calling Reboot. In such a case the request could either be rejected or the default load sequence could be performed. Either could be the correct action for some cases. Performing the default load sequence is seen as the best action for most cases.

EnterInteractiveMode

```
VOID EnterInteractiveMode(VOID);
```

The **EnterInteractiveMode** function performs the equivalent of **Restart**, but ignores the presence of valid Restart Blocks and acts as if the value of the **Autoload** environment variable is ***no***. This causes the system to enter firmware interactive mode.

4.3.3 Configuration Functions

The following configuration query functions are defined:

- **GetChild**, **GetParent**, and **GetPeer**, used to traverse the system configuration data structure and obtain component information.
- **AddChild**, used to add entries to the system configuration data structure.
- **DeleteComponent**, used to delete entries from the system configuration data structure.
- **GetComponent**, used to obtain component information for an input/output device.
- **GetConfigurationData**, used to obtain additional component-specific configuration data.
- **SaveConfiguration**, used to save the system configuration data structure in non-volatile memory.

GetChild(), GetParent(), GetPeer()

```
COMPONENT *GetChild(COMPONENT *Current);
COMPONENT *GetParent(COMPONENT *Current);
COMPONENT *GetPeer(COMPONENT *Current);
```

GetChild, **GetParent**, and **GetPeer** are used to traverse the system configuration database described in Section 4.2.5, “System Configuration Data,” on page 57.

GetChild(NULL) returns a pointer to the component structure for root node of the configuration data structure, that is, the system itself. **GetChild(Current)** returns a pointer to the structure of the child of **Current**, if one exists; otherwise, it returns NULL.

GetParent(Current) returns a pointer to the structure for the parent of **Current** if one exists; otherwise, it returns NULL.

GetPeer(Current) returns a pointer to the structure for the next peer of component **Current**, if one exists; otherwise, it returns NULL.

For each function, **Current** must be the value returned from a previous call of **GetChild**, **GetParent**, or **GetPeer**; for **GetChild**, NULL is allowed.

Programs may make copies of component structures but pointers to copies of the structures may not be used as arguments to these routines.

AddChild()

```
COMPONENT *AddChild(
        COMPONENT *Current,
        COMPONENT *Template,
        VOID *ConfigurationData);
```

AddChild creates a deep copy of the structure pointed to by **Template** in firmware-managed space and makes it a child of **Current**. If the **ConfigurationDataLength** field of the structure pointed to by **Template** is not NULL, the function also creates a configuration data structure in its own space and copies the configuration data pointed to by **ConfigurationData**.

AddChild affects only the current system configuration data structure. It does not update any non-volatile storage used to retain this information when the system is in the Off state.

This function returns a pointer to the new child if the addition is successful; otherwise it returns NULL.

DeleteComponent()

```
LONG DeleteComponent(COMPONENT * ComponentToDelete);
```

DeleteComponent removes the component pointed to by **ComponentToDelete** from the configuration data structure. Only components with no children may be deleted. If the specified component has children, the delete fails. If the delete succeeds, this function returns [ESUCCESS]; otherwise, it returns [EINVAL], indicating either that **Component** does not point to a valid configuration component or that the component has children.

GetComponent()

```
COMPONENT *GetComponent(CHAR *Path);
```

GetComponent returns a pointer to the component structure corresponding to the path string pointed to by **Path**. **Path** is a pointer to a null-terminated string containing the specified path.

In interpreting the path string, only those elements of the pathname that correspond to elements of the configuration data structure are used. That is, this function ignores elements of the pathname following the <peripheral> element, if any are present. For example, if a pathname includes a partition identifier, unit identifier, protocol specifier and so forth, these elements of the pathname are ignored. Since this function does not use pathname elements after the <peripheral> element, successful execution of this function does not imply that an **Open** with the same pathname would necessarily succeed.

If the relevant portion of **Path** is invalid, or if a component structure does not exist, this function returns NULL.

GetConfigurationData()

```
LONG GetConfigurationData(
       VOID *ConfigurationData,
       COMPONENT *Component);
```

The **GetConfigurationData** function is used to get additional data about the component pointed to by **Component**. **GetConfigurationData** stores component-specific information in the buffer pointed to by **ConfigurationData**. If successful, this function returns [ESUCCESS]; otherwise, it returns [EINVAL], indicating either that no data is available or that **Component** does not point to a valid configuration component.

The size of the buffer needed to hold the configuration data returned by is indicated in the **ConfigurationDataSize** field of the component structure pointed to by **Component**. A **ConfigurationDataSize** of 0 indicates that no configuration data is available. A call to **GetConfigurationData** in this case fails and returns [EINVAL]. Component configuration data, when provided, consists of a standard header and component specific details; see Section 4.2.6, “Additional Configuration Data,” on page 67 for further information.

SaveConfiguration()

```
LONG SaveConfiguration(void);
```

SaveConfiguration causes the current system configuration data structure to be written to non-volatile storage. If the function succeeds, it returns [ESUCCESS]; otherwise it returns [ENOSPC], indicating insufficient space in the medium used to store the configuration data. ARC systems are not required to provide non-volatile storage to store the configuration data base. If a system provides no storage for this purpose, **SaveConfiguration** returns [ENOSPC].

Rationale – For some systems, firmware can obtain all configuration information by "inspection." These systems may have no need to retain configuration information across power cycles. For other systems however, it may be necessary to query the operator for some configuration information or perhaps run a configuration utility program. For these systems, it is desirable to save the configuration information in non-volatile storage so that it is retained across power cycles.

Note – On systems which do not provide some form of non-volatile storage for configuration data, components may be added to, accessed, and removed from the configuration tree built by system firmware, but will not be saved across power cycles, since **SaveConfiguration** will always fail.

4.3.4 Input/Output Functions

The following input/output functions are defined:

- **Open**, used to open a file.
- **Close**, used to close a file.
- **Read**, used to read data from a file.
- **Write**, used to write data to a file.
- **Seek**, used to alter the current read or write position in a file.
- **Mount**, used to mount and dismount removable media.
- **GetFileInformation**, used to get information about a file.
- **SetFileInformation**, used to set file information.
- **GetDisplayStatus**, used to get current screen attributes.
- **TestUnicodeCharacter**, used to check for the existence of a valid glyph corresponding to a Unicode character.

Except where stated below, the Input/Output functions defined in this section operate on both files and raw devices, including character devices, block devices, and network devices.

Open()

```
LONG Open(
       CHAR *Path,
       OPENMODE OpenMode,
       ULONG *FileID);
```

The **Open** function opens the device or file specified by **Path** for the type of access specified by **OpenMode**.

If **Open** succeeds, it returns an int in **FileID**, sets the file's byte offset to 0, and returns [ESUCCESS]. Otherwise, it returns an error status code. The **FileID** is used as a file descriptor for the **Close**, **Read**, **GetReadStatus**, **Write**, **Seek**, **GetFileInformation**, and **SetFileInformation** functions. **Open** must return the smallest value possible in **FileID** that does not correspond to another open device or file.

The **Open** function implies an automatic mount operation for removable media devices. If the mount fails, the **Open** will fail and return [ENXIO] status.

Platform firmware must permit a minimum of 20 files to be in use simultaneously. The same entity (device, partition, or file) may be opened multiple times, but the results produced by concurrent access to a device and partitions or files on that device, or to a partition and files within that partition, are undefined.

```
typedef enum {
   OpenReadOnly,
   OpenWriteOnly,
   OpenReadWrite,
   CreateWriteOnly,
   CreateReadWrite,
   SupersedeWriteOnly,
   SupersedeReadWrite,
   OpenDirectory,
   CreateDirectory
} OPENMODE;
```

Definition 4-13: OPENMODE

OpenMode specifies how the device, file or partition is to be opened. **Open** will fail if **OpenMode** is inconsistent with device capabilities, e.g. if an attempt to open for writing is made to a read-only device. **Open** will fail if a protocol is specified and the device does not support the protocol specified.

The ***OpenReadOnly***, ***OpenWriteOnly***, and ***OpenReadWrite*** modes open a device or existing file for the specified access. These are the only open modes that are valid for devices. All other open modes are valid only for files. If the device or file does not exist, or if the specified file is a directory, an error code is returned.

The ***CreateWriteOnly*** and ***CreateReadWrite*** modes create a file with the specified access. If the file exists, an error code is returned.

The ***SupersedeWriteOnly*** and ***SupersedeReadWrite*** modes either open or create a file, depending upon whether the file already exists. If the file exists, it is truncated to zero. If the file is a directory, an error code is returned. When a file is created, only the ***ArchiveFile*** attribute flag is set.

The ***OpenDirectory*** mode opens an existing directory for read-only access. If the file is not a directory, an error code is returned. Specifying the ***CreateDirectory*** mode causes the creation of a directory for read-only access, with the ***directorial*** and ***ArchiveFile*** attribute flags set. There is no supersede mode for directories.

The following error status codes are returned by **Open**:

`[EACCES]`	The named file is read-only and **OpenMode** indicates write access; or **OpenMode** indicates a create operation, and the file exists.
`[EINVAL]`	The **OpenMode** parameter indicates an invalid mode.
`[EISDIR]`	**OpenMode** is not either ***OpenDirectory*** or ***CreateDirectory***, and the specified file is a directory.
`[EMFILE]`	Too many file descriptors are in use.
`[ENAMETOOLONG]`	The filename portion of the path specified is too long.
`[ENODEV]`	The device portion of the pathname is invalid.
`[ENOENT]`	The named device or file does not exist; or the named file does not exist, and **OpenMode** specifies read access.
`[ENOSPC]`	There is no room on the disk for the new file or directory.
`[ENOTDIR]`	**OpenMode** specifies ***OpenDirectory*** or ***CreateDirectory*** and the specifies file is not a directory.
`[EROFS]`	The named device is read-only, and **OpenMode** specifies write access.
`[ENXIO]`	Media is not loaded on the device specified by the device portion of the specified path (removable media devices only).

Close()

```
LONG Close(ULONG FileID);
```

The **Close** function closes the device or file specified by **FileID**. If the close operation succeeds, [ESUCCESS] is returned; otherwise, [EBADF] is returned, indicating an invalid file descriptor.

Read()

```
LONG Read(
        ULONG FileID,
        VOID *Buffer,
        ULONG N,
        ULONG *Count);
```

The **Read** function reads data from the device or file specified by **FileID** into memory, starting at the address in **Buffer**. If **N** is 0, no bytes are read. If **N** is greater than 0, no more than **N** bytes of data are read. If the read operation succeeds, the actual number of bytes read is stored in ***Count**, and the byte offset is updated accordingly. If a call to **Read** with **N** > 0 succeeds and sets ***Count** equal to 0, end of file has been reached.

From a firmware perspective, all read operations are treated as byte reads. If the device represented by **FileID** is a block device and the calling program wants to perform block oriented operations, it may do so by assuring that byte counts supplied as arguments to **Read** are integer multiples of the device block size. **Read** operations with byte counts that are not multiples of a block device's block size are valid and return the next **N** bytes in sequence, treating data from the device as a sequential byte stream.

If the device represented by **FileID** is a network device, then up to **N** bytes are read from the current network packet and placed in **Buffer**. If there are fewer than **N** bytes remaining in the current network packet, the bytes actually available are placed in **Buffer**, and **Count** is set to the number of bytes placed in the buffer. No data will be taken from the next packet. If there is no current network packet, the **Read** will not return until either a packet arrives or an error occurs.

Rationale – Some software may need to perform input and output directly with a network device and perform its own protocol processing. This feature simplifies the task of recognizing packet boundaries.

Status codes returned by **Read** are:

`[ESUCCESS]`	The read operation was successful.
`[EBADF]`	The file descriptor specified by **FileID** is invalid.
`[EIO]`	An input error occurred during the read.

GetReadStatus()

```
LONG GetReadStatus(ULONG FileID);
```

The function **GetReadStatus** determines if any bytes would be returned if a read operation were performed on **FileID**. If one or more bytes would be returned, [ESUCCESS] is returned; otherwise, an error status code is returned.

This function may be used to determine if a **Read** would block. If the device represented by **FileID** is a network device, **GetReadStatus** will fail if there is no current packet from which one or more bytes can be returned.

Status codes returned by **GetReadStatus** are:

`[ESUCCESS]`	One or more bytes would be returned.
`[EBADF]`	The file descriptor specified by **FileID** is invalid.
`[EAGAIN]`	No bytes are currently available.

Write()

```
LONG Write(
       ULONG FileID,
       VOID *Buffer,
       ULONG N,
       ULONG *Count);
```

The **Write** function writes data from memory, starting from the address in **Buffer** to the device specified by **FileID**. The number of bytes to write is specified by **N**. If the **N** bytes are actually written or if **N** is 0, the operation succeeds. Upon successful completion, **Write** stores the number of bytes written in ***Count**; otherwise, an error status code is returned. If **Write** cannot write **N** bytes, then ***Count** is set to the number actually written. Upon completion of a successful write, the byte offset is updated accordingly; otherwise, the byte offset is left unchanged.

If the device represented by **FileID** is a block device, the calling program can perform block oriented operations by restricting **N** to integer multiples of the device block size. **Write** operations with byte counts that are not multiples of a block device's block size are valid and write the next **N** bytes in sequence, treating the device as a sequential byte-oriented device.

If the device represented by **FileID** is a network device, the **N** data bytes are all encapsulated in a single network packet. If **N** bytes cannot be placed into a network packet, **Write** returns [ENOSPC], places the number of bytes that could be written in ***Count**, and returns without sending the packet. The format of the data within the packet is network and protocol specific and is not addressed in this specification.

Status codes returned by **Write** are:

`[ESUCCESS]`	The write operation was successful.
`[EBADF]`	The file descriptor specified by **FileID** is invalid for this write operation, or the network does not support write.
`[EIO]`	An output error occurred during the write.
`[ENOSPC]`	The device is full, or the amount of data was too large to fit into a network packet.

Seek()

```
LONG Seek(
       ULONG FileID,
       LARGEINTEGER *Position,
       SEEKMODE SeekMode);
```

The **Seek** function changes the byte offset associated with the device, partition, or file specified by **FileID**. **Position** is a 64-bit signed quantity specified by the LARGEINTEGER structure. The 64-bit value is obtained by concatenation of **HighPart** with **LowPart**.

```
typedef struct{
   ULONG                  LowPart;
   LONG                   HighPart;
} LARGEINTEGER;
```

Definition 4-14: LARGEINTEGER;

```
typedef enum {
   SeekAbsolute,
   SeekRelative
} SEEKMODE;
```

Definition 4-15: SEEKMODE

SeekMode indicates the type of positioning to be performed. If **SeekMode** is ***SeekAbsolute***, the offset is set to **Position**. If **SeekMode** is ***SeekRelative***, the value of **Position** is added to the offset associated with the specified **FileID** creating a new offset, using 64-bit signed addition.

If the file specified by **FileID** has the attribute ***DirectoryFile***, a **Seek** with **SeekMode** of ***SeekAbsolute*** and a position of 0 causes the directory entry pointer to be reset to the first entry in the directory.

For block devices, positioning on block boundaries may be achieved by performing seeks with offsets that are multiples of the block size.

For network devices, **SeekMode** is ignored, and the offset is taken to be a count of input packets to ignore. The data in packets that are ignored is discarded.

For disk files, the offset may be increased or decreased.

Error status codes returned by **Seek** are:

[ESUCCESS]	The seek operation was successful.
[EBADF]	The file descriptor specified by **FileID** is invalid.
[EINVAL]	The **SeekMode** specified is invalid or the offset specified is beyond the end of the file or device.
[EIO]	An error occurred while the network device was waiting for the prescribed number of packets to arrive.

Mount()

```
LONG Mount(
       CHAR *Path,
       MOUNTOPERATION Operation);
```

The **Mount** function is used to load and unload media for devices that support removable media.

```
typedef enum {
   LoadMedia,
   UnloadMedia
} MOUNTOPERATION;
```

Definition 4-16: MOUNTOPERATION

Operation indicates whether media is to be loaded or unloaded. If **Operation** is ***LoadMedia***, media for the device specified by **Path** is loaded. If **Operation** is ***UnloadMedia***, the media is unloaded. The **Mount** function returns [ESUCCESS] if the operation succeeds; otherwise, an error status code is returned.

Mount fails if **Path** specifies a file, a network device, or if **Path** specifies a device that does not support removable media. Opening a device that supports removable media automatically causes a media load attempt, and a **Mount** of such a device prior to **Open** is not required. Note, however, that **Close** does not perform unload media operation.

Error status codes returned by **Mount** are:

[EBADF]	The file descriptor specified by **FileID** is invalid.
[EINVAL]	The path specified is ill-formed.
[ENOENT]	The named device does not exist.
[ENXIO]	Media is not loaded on the device specified by the device portion of the specified path.

GetFileInformation()

```
LONG GetFileInformation(
       ULONG FileID,
       FILEINFORMATION *Information);
```

The **GetFileInformation** function returns an information structure about the specified file or partition. The format of the information structure is shown in Definition 4-17.

```
typedef struct{
   LARGEINTEGER            StartingAddress;
   LARGEINTEGER            EndingAddress;
   LARGEINTEGER            CurrentAddress;
   CONFIGTYPE              Type;
   ULONG                   FileNameLength;
   UCHAR                   Attributes;
   CHAR                    Filename[32];
} FILEINFORMATION;
```

Definition 4-17: FILEINFORMATION

For files, **StartingAddress** is zero and **EndingAddress** is the size of the file, in bytes. For partitions, **StartingAddress** and **EndingAddress** are the start and end position of the partition in the form of byte offsets from the start of the disk. **GetFileInformation** is invalid for devices.

CurrentAddress contains the current file offset into the file or partition.

The **Type** field is the type of the device on which the file or partition resides.

For files, **FileNameLength** contains the length, in bytes, of the filename, and Filename is the name of the file. **Filename** is a null-terminated string and includes any separator characters that appear as part of the name (for example, the dot in "FILE.EXT"). For partitions, **FileNameLength** is zero and **Filename** is NULL.

For files, **Attributes** contains file attribute flags.

```
typedef enum{
   ReadOnlyFile =          1,
   HiddenFile =            2,
   SystemFile =            4,
   ArchiveFile =           8,
   DirectoryFile =         16,
   DeleteFile =            32
} FILEATTRIBUTES;
```

Definition 4-18: FILEATTRIBUTES;

SetFileInformation()

```
LONG SetFileInformation(
       ULONG FileID,
       ULONG AttributeFlags,
       ULONG AttributeMask);
```

The **SetFileInformation** function allows a file's attribute flags to be modified. **SetFileInformation** is only valid for files. Each attribute flag is set or cleared depending on the **AttributeFlags** parameter if the corresponding bit is set in the **AttributeMask** parameter. For example, a file can be set to read-only by setting the ***ReadOnlyFile*** flag in both the **AttributeFlags** and **AttributeMask** parameter. The ***DirectoryFile*** flag is ignored for **SetFileInformation** operations.

A file can be marked for deletion by setting the ***DeleteFile*** flag in both the **AttributeFlags** and **AttributeMask** parameters. After a file has been marked for deletion, **Close** is the only valid operation that can be successfully performed on the specified **FileID**. Directory files that are not empty, or data files that are marked read-only, cannot be deleted.

This function returns the following status codes:

`[ESUCCESS]`	The attribute flags were set successfully.
`[EACCES]`	The **DeleteFile** attribute is being set, and the file is a directory which is not empty or a read-only file.
`[EBADF]`	**FileID** is an invalid descriptor.

GetDisplayStatus()

```
DISPLAY_STATUS *GetDisplayStatus(ULONG FileID);
```

The **GetDisplayStatus** function returns a pointer to the display status structure shown in Definition 4-19. This function returns NULL if the device associated with **FileID** is not a display device or is otherwise an invalid file descriptor.

```
typedef struct{
   USHORT                  CursorXPosition;
   USHORT                  CursorYPosition;
   USHORT                  CursorMaxXPosition;
   USHORT                  CursorMaxYPosition;
   UCHAR                   ForegroundColor;
   UCHAR                   BackgroundColor;
   UCHAR                   HighIntensity;
   UCHAR                   Underscored;
   UCHAR                   ReverseVideo;
} DISPLAY_STATUS;
```

Definition 4-19: DISPLAY_STATUS

The **CursorXPosition** and **CursorYPosition** fields contain the current text cursor position for the display device. The **CursorMaxXPosition** and **CursorMaxYPosition** fields contain the maximum text cursor positions for the display device. **ForegroundColor** and **BackgroundColor** contain the current foreground and background color codes, respectively, for the display. **HighIntensity**, **Underscored**, and **ReverseVideo** are set (nonzero) or clear (zero) depending on whether subsequent characters will be written in a highlighted, underscored, or reverse video mode, respectively.

Note – This structure is not updated automatically by subsequent output to the display device. If additional output is performed, **GetDisplayStatus** must be called again to get the current display status information.

TestUnicodeCharacter()

```
LONG TestUnicodeCharacter(
        ULONG FileID,
        USHORT UnicodeCharacter);
```

The **TestUnicodeCharacter** function is used to test whether or not the display driver associated with **FileID** is capable of rendering the Unicode character in **UnicodeCharacter**.

This routine returns the following status codes:

`[ESUCCESS]`	The character in **UnicodeCharacter** can be rendered.
`[EBADF]`	The file descriptor specified by **FileID** is invalid.
`[EINVAL]`	The character in **UnicodeCharacter** cannot be rendered.

4.3.5 Environment Functions

Two environment functions are defined:

- **SetEnvironmentVariable**, used to set the value of a global environment variable.
- **GetEnvironmentVariable**, used to get the value of a global environment variable.

SetEnvironmentVariable()

```
LONG SetEnvironmentVariable(
        CHAR     *Name,
        CHAR     *Value);
```

SetEnvironmentVariable sets the value of a global environment variable. **Name** is a pointer to a string containing the name of the variable. **Value** is a pointer to a string containing the value to be set. If the function succeeds, it returns [ESUCCESS]; otherwise, it returns [ENOSPC], indicating insufficient space where the environment variables are stored. The string pointed to by **Name** is not case sensitive. The string pointed to by **Value** is case sensitive.

GetEnvironmentVariable()

```
CHAR *GetEnvironmentVariable(CHAR *Name);
```

GetEnvironmentVariable is used to obtain the current value of a global environment variable. **Name** is a pointer to a string containing the name of the variable. If the variable exists, this function returns a pointer to a string containing its value. Otherwise, the NULL pointer is returned. The string pointed to by **Name** is not case sensitive.

4.3.6 Miscellaneous Functions

The following miscellaneous functions are defined:

- **GetSystemID**, used to get a unique system identifier structure.
- **GetMemoryDescriptor**, used to access system memory descriptors.
- **GetTime**, used to get the date and time.
- **GetRelativeTime**, used to measure time intervals.
- **GetDirectoryEntry**, used to read file directories.
- **FlushAllCaches**, used to flush the contents of all memory caches.

GetSystemId()

```
SYSTEMID *GetSystemId(void);
```

The **GetSystemId** routine returns a pointer to a system identification structure containing information used to uniquely identify each ARC system. The 16-byte structure value may be used as a key for security or license purposes.

```
typedef struct{
   CHAR VendorId[8];
   UCHAR ProductId[8];
} SYSTEMID;
```

Definition 4-20: SYSTEMID

VendorId is an 8-byte ASCII string that uniquely identifies each ARC system vendor. **VendorId** values are registered with the ACE Initiative through MIPS Technologies, Inc. To register a **VendorId**, contact:

MIPS Technologies, Inc.
2071 N. Shoreline Blvd.
Mountain View, CA 94039-7311

ProductId, is an 8-byte field whose contents uniquely identify each ARC system produced by the vendor identified in **VendorId**. The value in **ProductId** is assigned by the system vendor and, subject to the size uniqueness constraints, may be assigned in any manner.

GetMemoryDescriptor()

```
MEMORYDESCRIPTOR   *GetMemoryDescriptor(
       MEMORYDESCRIPTOR * Current);
```

GetMemoryDescriptor returns a pointer to a memory descriptor structure. **GetMemoryDescriptor(NULL)** returns a pointer to the first memory descriptor. **GetMemoryDescriptor(Current)** returns a pointer to the memory descriptor after the memory descriptor whose address is **Current**. If there are no more descriptors or if **Current** is invalid, NULL is returned. The value of **Current** must be either NULL or the value returned by a previous call of **GetMemoryDescriptor**.

The format of a memory descriptor structure is shown in Definition 4-22.

```
typedef enum{
   ExceptionBlock,
   SystemParameterBlock,
   FreeMemory,
   BadMemory,
   LoadedProgram,
   FirmwareTemporary,
   FirmwarePermanent,
   FreeContiguous
} MEMORYTYPE;
```

Definition 4-21: MEMORYTYPE

```
typedef struct{
   MEMORYTYPE                Type;
   ULONG                     BasePage;
   ULONG                     PageCount;
} MEMORYDESCRIPTOR;
```

Definition 4-22: MEMORYDESCRIPTOR

The **Type** field indicates the type of memory region described.

The **BasePage** field indicates the page number of the beginning of the memory region. Pages are assumed to be 4,096 bytes in size.

The memory descriptors returned by this routine describe the state of system memory as seen by the firmware after it last performed an initial program load or executed an **Execute**.

GetMemoryDescriptor may return a pointer to a temporary memory descriptor structure. If so, a subsequent call to **GetMemoryDescriptor** will overwrite the previous copy. Programs that need to maintain the data provided by **GetMemoryDescriptor** should make a copy of the data in their own memory.

GetTime()

```
TIMEINFO *GetTime(void);
```

The **GetTime** function returns a pointer to a structure containing the current time. The format of this structure is shown in Definition 4-23. In this structure, **Year** contains the current year; **Month** contains a value between 1 and 12; **Day** contains a value between 1 and 31; **Hour** contains a value between 0 and 23; **Minute** and **Seconds** contain values between 0 and 59; and **Milliseconds** contains a value between 0 and 999.

```
typedef struct{
   USHORT                Year;
   USHORT                Month;
   USHORT                Day;
   USHORT                Hour;
   USHORT                Minutes;
   USHORT                Seconds;
   USHORT                Milliseconds;
} TIMEINFO;
```

Definition 4-23: TIMEINFO

ARC-compliant systems maintain Universal Coordinated Time. Local time may be derived using the time zone information in the **TimeZone** environment variable.

No function for setting time is defined. Functions to set the time are implementation-specific.

GetRelativeTime()

```
ULONG GetRelativeTime();
```

The **GetRelativeTime** function returns the time in seconds since some arbitrary starting point. This function is provided to enable timing of intervals without requiring conversion from the form provided by **GetTime**.

GetDirectoryEntry()

```
LONG GetDirectoryEntry(
      ULONG    FileID,
      DIRECTORYENTRY *Buffer,
      ULONG    N,
      ULONG    *Count);
```

The **GetDirectoryEntry** function is used to read directory entries from a system partition directory file specified by **FileID** The first call with a valid **FileID** after opening a directory reads one or more entries, beginning with the first entry. Repeated calls read subsequent entries.

A maximum of **N** entries are read into the location specified by **Buffer**. The actual number of entries read is returned in ***Count**.

A **Seek** with a **SeekMode** of ***SeekAbsolute*** with an offset of 0, performed on an open directory **FileID**, causes the entry pointer to be reset to the first entry.

The format of a directory entry is shown in Definition 4-24.

```
typedef struct{
   ULONG                    FileNameLength;
   ULONG                    FileAttribute;
   ULONG                    FileName[32];
} DIRECTORYENTRY;
```

Definition 4-24: DIRECTORYENTRY

FileNameLength contains the length of the file name for the file corresponding to the directory entry. **Filename** is a string containing the name of the file, including any separator characters (for example, the dot in “FILE.EXT”).

FileAttribute contains file flag bits. (See Definition 4-18.)

GetDirectoryEntry returns the following error status codes:

`[ENOTDIR]`	An attempt has been made to read past the last directory entry in the directory file.
`[EBADF]`	**FileID** is not a valid file descriptor, or does not correspond to an open directory file.

FlushAllCaches()

```
VOID FlushAllCaches(VOID);
```

The **FlushAllCaches** routine forces the contents of memory caches to be made consistent with memory.

4.3.7 The Firmware Function Vector

All firmware functions are called indirectly through the firmware vector. The firmware vector, whose address is held in the System Parameter Block, is an array of pointers to the entry points of the standard firmware routines. The contents of the firmware vector are shown in Table 4-4.

Table 4-4 Firmware Vector

Firmware Function	Vector Offset	Vector Index
AddChild	0x34	14
Close	0x60	25
DeleteComponent	0x38	15
EnterInteractiveMode	0x1C	8
Execute	0x08	3
FlushAllCaches	0x88	35
GetChild	0x28	11
GetComponent	0x3C	16
GetConfigurationData	0x30	13
GetDirectoryEntry	0x58	23
GetDisplayStatus	0x90	37
GetEnvironmentVariable	0x78	31
GetFileInformation	0x80	33
GetMemoryDescriptor	0x48	19
GetParent	0x2C	12
GetPeer	0x24	10
GetReadStatus	0x64	27
GetRelativeTime	0x54	22
GetSystemID	0x44	18
GetTime	0x50	21
Halt	0x0C	4
Invoke	0x04	2
Load	0x00	1
Mount	0x74	30
Open	0x5C	24
PowerDown	0x10	5
Read	0x64	26
Reboot	0x18	7
Restart	0x14	6
SaveConfiguration	0x40	17
Seek	0x70	29
SetEnvironmentVariable	0x7C	32
SetFileInformation	0x84	34
TestUnicodeCharacter	0x8C	36
Write	0x6C	28

4.3.8 Platform-Specific Firmware Functions

An ARC platform may provide firmware functions that are not defined by the ARC Spec. These functions are accessed indirectly via the private firmware vector. The address and size of the private vector is in the System Parameter Block.

4.3.9 Adapter-Specific Firmware Functions

Addenda to the ARC Spec may define additional firmware functions. These addenda-specific functions are accessed indirectly via an addendum-specific vector. The type, address, and size of each addendum-specific vector is in the System Parameter Block.

4.4 Loaded-Program Conventions

Programs are loaded by platform firmware automatically following a power-on or reset condition; in response to an operator command in interactive firmware mode; or by an already loaded program calling a program loading function. Loadable programs must reside in a system partition, and the program image must conform to the object file requirements specified in Section 4.1.4, “Object Formats,” on page 50.

When programs are loaded, whether automatically by platform firmware, in response to an operator command, or as the result of an **Execute** function, firmware adjusts the memory descriptors as described in Section 4.3.1, “Program Loading,” on page 75, for the **Execute** function. Programs executing while the system remains in the firmware state are recommended to restrict their use of memory to the ***LoadedProgram*** and ***FreeContiguous*** memory regions assigned to them by the platform firmware.

Note – When programs are loaded and invoked via the **Load** and **Invoke** functions, the memory descriptors are not modified. Programs loaded via these two functions are presumed to be “managed” by the program that loaded them, and such programs are also recommended to restrict their use of memory to the ***LoadedProgram*** and ***FreeContiguous*** regions originally assigned by the platform firmware.

The entry point of programs loaded by platform firmware is assumed to have a function prototype of the form:

```
LONG Main(LONG Argc, CHAR *Argv[], CHAR *Envp[]);
```

where the arguments **Argc** and **Argv** conform to common ‘C’ language usage: **Argv** is a vector of pointers to null terminated argument strings; **Argc** is the number of entries in the vector; and **Envp** is a vector of pointers to null terminated global environment variable strings. The NULL pointer is a valid entry in **Argv**, but a NULL pointer terminates the list of pointers in the **Envp** vector.

The value of **Argv[0]** is set to the full pathname of the program being loaded.

When a program is loaded automatically, additional arguments corresponding to the effective values of the **OSLoader**, **SystemPartition**, **OSLoadFilename**, **OSLoadPartition**, **LoadIdentifier**, **OsLoadOptions**, **ConsoleIn**, and **ConsoleOut** environment variables are passed as *name=value* strings. The order of these arguments is not specified.

Note – Environment variables may contain multiple values separated by semicolons. The values passed as arguments to a loaded program are the actual values used for the load attempt.

For example, suppose the global environment variables had the values indicated as follows:

OSLoader=scsi(1)disk(3)rdisk(3)partition(2)\OS\ARCOS\LOADER
SystemPartition=`scsi(1)disk(3)rdisk(3)partition(2)`
OSLoadFilename=`\KERNELS\FRED`
OSLoadPartition=`scsi(1)disk(1)rdisk(2)partition(1)`
LoadIdentifier=`Scenario1`
OsLoadOptions=`SECURE`
ConsoleIn=`key(0)keyboard(0)console(1)`
ConsoleOut=`video(0)monitor(0)console(1)`
AutoLoad=`yes`
TimeZone=`CST6`
FWSearchPath=

If the loader program is invoked automatically, the following arguments will be passed:

```
Argc = 0

Argv[0] -> "scsi(1)disk(3)rdisk(3)partition(2)\OS\ARCOS\LOADER"
Argv[1] -> "OsLoader=scsi(1)disk(3)rdisk(3)partition(2)\OS\ARCOS\LOADER"
Argv[2] -> "SystemPartition=scsi(1)disk(3)rdisk(3)partition(2)"
Argv[3] -> "OSLoadFilename=\KERNELS\FRED"
Argv[4] -> "OSLoadPartition=scsi(1)disk(1)rdisk(2)partition(1)"
Argv[5] -> "LoadIdentifier=Scenario1"
Argv[6] -> "OsLoadOptions=SECURE"
Argv[7] -> "ConsoleIn=key(0)keyboard(0)console(1)"
Argv[8] -> "ConsoleOut=video(0)monitor(0)console(1)"

Envp[0] -> "OSLoader=scsi(1)disk(3)rdisk(3)partition(2)\OS\ARCOS\LOADER"
Envp[1] -> "SystemPartition=scsi(1)disk(3)rdisk(3)partition(2)"
Envp[2] -> "OSLoadFilename=\KERNELS\FRED"
Envp[3] -> "OSLoadPartition=scsi(1)disk(1)rdisk(2)partition(1)"
Envp[4] -> "LoadIdentifier=Scenario1"
Envp[5] -> "OsLoadOptions=SECURE"
Envp[6] -> "ConsoleIn=key(0)keyboard(0)console(1)"
Envp[7] -> "ConsoleOut=video(0)monitor(0)console(1)"
Envp[8] -> "AutoLoad=yes"
Envp[9] -> "TimeZone=CST6"
Envp[10] -> "FWSearchPath="
```

Rationale – The general use of **Argc** and **Argv** and **Envp** follows common ‘C’ programming practices, but passing arguments in keyword format does not. The keyword format was chosen to make the arguments self-identifying and also to allow additional, non-standard arguments to be passed, while avoiding any difficulties that would arise from a positional approach.

If the program is loaded by the **Load** or **Execute** functions, the caller specifies the arguments passed to the loaded program. It is recommended that caller and interactive firmware-constructed argument lists follow the same convention as defined for automatic program loads.

4.5 Interrupts and Exceptions

As stated in Section 2.5, “Firmware,” on page 26, when an ARC system is in the firmware state, the firmware is responsible for handling any and all interrupts or exceptions that might arise as well as all system Input/Output. A system passes from the firmware state to the program state when the a loaded program assumes responsibility for handling interrupts and exceptions, takes control of memory management, and all Input/Output functions, and so on.

However, the ARC Spec specifies enough about how interrupts and exceptions are to be handled in the firmware state to allow loaded programs to take over control of exceptions and interrupts, but still be able to return control to the firmware. A loaded program can use this mechanism to take over control of the machine in stages.

Invoking Exception Handlers

All interrupts and exceptions cause traps to fixed locations in the first page of memory, as specified by the MIPS processor architecture. While in the firmware state, the primitive code invoked there must call the appropriate handlers specified in the **GEVector** and **UTLBMissVector** fields of the System Parameter Block. The handler addresses must be fetched from System Parameter Block in the ***kseg0*** address space. When the handler routine is called, the return address must be saved in processor register **K1**. On return from the handler routine, the code must perform a return from exception with **K0** containing the return address. The following MIPS assembly language code fragment illustrates one implementation:

Code Example 4-1 Exception Handling

```
lui     K0,(0x80000000 | 0x1000 + SPBEntryOffset) >>16
ori     K0,(0x80000000 | 0x1000 + SPBEntryOffset) & 0xffff
lw      K0,0(K0)
nop
jalr        K1,K0
nop
j       K0
c0_rfe
```

where ***SPBEntryOffset*** is the offset of either the **GEVector** or **UTLBMissVector** in the System Parameter Block. Firmware must initialize these fields to point to exception handlers before any exceptions may be processed.

Exception Handler Routines

Exception handler routines invoked via the preceding mechanism must meet the following constraints:

- On entry they must save all state. They may not assume that the stack pointer points to a usable stack area.
- Before returning, they must restore all state and place the return address in **K0**, restore **K1** to the value it had on entry, and return to the caller with a jump to **K1**.

These constraints imply that the tails of the exception handlers look like:

Code Example 4-2 Restoring Registers

```
/*restore all registers */
K0 <- return from exception destination (EPC)
K1 <- contents of K1 at entry to handler
    j   K1
```

Loaded Program Access to Exceptions

While in the firmware state, loaded programs may take control of exceptions and memory management, while preserving the ability to invoke firmware routines by meeting the following constraints:

- The loaded program must save the firmware-supplied exception-handler routine addresses and store the addresses of its own handlers in the System Parameter Block.
- If a loaded program manipulates the system coprocessor, it must install its own exception handlers, and it must preserve the entire state of the system coprocessor established by the firmware before making any change to the System Parameter Block or any system coprocessor register.
- If a loaded program has installed its own handlers and manipulates the system coprocessor, it must restore all state of the system coprocessor to the state established by the firmware before it may call any firmware function (including firmware exception handlers). In essence, loaded programs must assure that their use of the system coprocessor and their handling of exceptions is totally invisible to system firmware.
- ***FirmwareTemporary*** memory regions must not be modified.
- Device control and data registers must not be accessible by the loaded program.
- If firmware runs with interrupts enabled, then the loaded program must be able to disable interrupts.

Eventually the loaded program abandons these constraints and assumes complete control of the platform. When it does so, the system enters the program state.

System Console 5

All ARC Spec compliant systems must include at least one device capable of serving as a console input device and one device serving as a console output device.

Console input and output provide the means for firmware functions (and programs loaded by firmware) to present messages to an operator and obtain input from an operator. The console capability may be provided by multiple devices; in that case, means are defined for specifying the device being used. The goal is to allow for simple localization of ARC hardware, utilities, and operating systems, but keep compatibility with most existing ARC hardware and software.

Console capability is defined in terms of

1. Console functionality.

2. Console operations.

Two levels of console functionality are specified:

1. UNICODE This is the highest level of functionality and support. It consists of all features in the Basic Console, with additional support for different languages.

2. Basic This level of functionality is comparable to an intelligent terminal. All ARC-compliant systems must support at least this level of functionality.

5.1 Functionality

5.1.1 Basic Console Input

The base ARC Spec defines the basic terminal level console functionality as providing the following.

- Console input and output is text (character) based. The Basic Console does not provide support for bit maps on output or pointing devices on input.

- Returns the ISO Latin1 character set plus a set of control sequences for function keys and other special keys. Function and other special keys are presented on input as escape sequences as defined in ANSI X3.64. Receipt of any of these escape sequences on the console input device is to be interpreted as depression of a key or combination of keys on the keyboard which is defined to have the corresponding interpretation.
- The input data stream from a console input device is based on the same character encodings that are defined for the industry standard personal computer keyboard.

The specific control sequences required are as follows:

Table 5-1 Function Key Control Sequences

Control Sequence*	Interpretation
CSI A	Move cursor up 1 row.
CSI B	Move cursor down 1 row.
CSI C	Move cursor right 1 column.
CSI D	Move cursor left 1 column.
CSI H	Home
CSI @	Insert
CSI P	Delete
CSI K	End
CSI ?	Page Up
CSI /	Page Down
CSI O P	Function 1
CSI O Q	Function 2
CSI O w	Function 3
CSI O x	Function 4
CSI O t	Function 5
CSI O u	Function 6
CSI O q	Function 7
CSI O r	Function 8
CSI O p	Function 9
CSI O M	Function 10
CSI O A	Function 11
CSI O B	Function 12

Note – *The white space in these sequences is for ease of reading and does not represent a blank in the sequence.

In the above input sequences, the control sequence introducer (CSI) is the character code with the hexadecimal value 9B. Compliant systems are not required to support the two character sequence Esc[.

5.1.2 UNICODE Console Input

The UNICODE Console becomes a UNICODE version of the Basic Console. For current implementations, all that is required is to zero extend the ASCII characters that are currently generated. New implementations can provide different mappings from keystrokes to characters, including an input method editor, if required (an interactive editor that aids the creation of a single character from multiple keystrokes, including the selection of one of several alternatives, in some cases).

FAT filenames are ASCII, so every implementation must have a method of input for the UNICODE characters U+0020 to U+007E.

The function keys are represented as UNICODE control sequences, as defined above. A Unicode control sequence is a sequence of UNICODE characters that begins with the Unicode CSI character (U+009b). Note that the Unicode control sequences are the same as their ASCII counterparts in the Basic Console, except that the 8-bit ASCII characters are converted to 16-bit Unicode characters with the upper 8-bits zero.

5.1.3 Basic Console Output

- The standard character set for text output is the ISO Latin1 character set. For line drawing, this changes to ISO Latin 1 characters 0x20 to 0x7E, plus Code Page 437 characters 0xB3 to 0xDA. This parallels the required characters for the UNICODE console.
- The base ARC specification also defines the output control sequences for screen control. This is the same as the input with the addition of the control sequences that are for color support.

Console output functions must meet the following requirements:

- The console presentation space is a minimum of 25 lines of 80 characters each. Lines are not automatically wrapped. That is, output in excess of the line length is lost. TABS are preset at every eighth column, starting with column 9, and are fixed.
- Devices opened as a console output device are required to accept streams of characters consisting of standard text characters and embedded control sequences. Such devices must process embedded control sequences to effect formatting of text. In particular, the following control sequences, a subset of ANSI standard X3.64, must be supported.

The tables below identify the subset of ANSI X3.64 to be supported. The brief descriptions are provided only for convenience. In all cases, the precise definition of the control sequence is as defined in ANSI X3.64.

In general, these sequences are defined for an 8-bit environment. Where a graphic exists for a character code, that graphic is used. Where necessary for clarity, the *column/row* notation of X3.64 is used.

Table 5-2 Control Sequences

Esc Sequence(1)	Description
CSI J	Erases from the cursor to the end of screen, including the position where the cursor resides.
CSI 0 J	Same as above.
CSI 1 J	Erases from the beginning of the screen up to and including the position where the cursor resides.
CSI 2 J	Erases the entire screen and places the cursor in the upper left position.
CSI K	Erases from the cursor to the end of the line, including the position where the cursor resides.
CSI 0 K	Same as above.
CSI 1 K	Erases from the beginning of the line up to and including the position where the cursor resides.
CSI 2 K	Erases the complete line and places the cursor at the leftmost position.
CSI row;col H(2)	Move cursor to row row and column col.
CSI n A(2)	Move cursor up n rows.
CSI n B(2)	Move cursor down n rows.
CSI n C(2)	Move cursor right n columns.
CSI n D(2)	Move cursor left n columns.
CSI H	Home
CSI @	Insert
CSI P	Delete
CSI K	End
CSI ?	Page Up
CSI /	Page Down
CSI O P	Function 1
CSI O Q	Function 2
CSI O w	Function 3
CSI O x	Function 4
CSI O t	Function 5
CSI O u	Function 6
CSI O q	Function 7
CSI O r	Function 8
CSI O p	Function 9
CSI O M	Function 10
CSI O A	Function 11
CSI O B	Function 12
CSI n m or CSI n;...n m	Select graphic rendition. All following characters in the data stream are rendered according to the parameters until the next SGR.When used with multiple parameters, the parameters are applied in sequence. Parameters have the following meanings: 0-> all attributes off. 1-> Display at high intensity. 4-> Display underscored. 7-> Display negative image (reverse video). 10-> Primary font as designated by FNT. 1x(2) x th Alternate font as designated by FNT. 1 <= x <= 9

Note –
(1) The white space in these sequences is for ease of reading and does not represent a character (i.e., blank) in the sequence.

(2) Numeric values for row, col, n, and x are specified by strings of ASCII digits.

(3) For clarity, this character is specified using the column/row notation of ANSI X3.64; 2/0 represents the “blank” character, i.e., the character whose 8-bit code is 0x20 and is otherwise indistinguishable from white space.

The table below defines additional parameter values for the SGR (Select Graphics Rendition) control sequence that must be supported. These values are compliant with ISO Standard 6429.

Table 5-3 Additional SGR Control Sequences

Esc Sequence (1)	Description
CSI n m or CSI n;...n m	Select graphics rendition, additional parameter values for color support. Parameters have the following meanings:
	30 foreground black
	31 foreground red
	32 foreground green
	33 foreground yellow
	34 foreground blue
	35 foreground magenta
	36 foreground cyan
	37 foreground white
	40 foreground black
	41 background red
	42 background green
	44 background yellow
	44 background blue
	45 background magenta
	46 background cyan
	47 background white

Note – (1) The white space in these sequences is for ease of reading and does not represent a character (i.e., blank) in the sequence.

Programs that use the extended features defined for the enhanced character console must first determine that the device currently opened as the console output device supports the additional SGR parameter values defined above. Output of the additional control sequences to a console device that does not support this feature produces unpredictable results.

Programs may determine the level of functionality supported by the currently open console device by obtaining the configuration data structure for the device. Note that the environment variables **ConsoleIn** and **ConsoleOut** contain the pathname of the currently open console, and, therefore, the current console device and mode can be determined from these.

In addition to the multicharacter sequences above, the following single-character control functions must also be supported:

Table 5-4 SIngle Character Control Functions

Esc Sequence(1)	Mnemonic	Description
	Nul	Null – Ignored when received.
	Bel	Bell – Generates a bell tone.
	BS	Backspace – Moves cursor one character position to the left. If the cursor is at left margin, no action is taken.
	HT	Horizontal Tab – Moves cursor to next tab stop or to right margin if there is no tab stop. Tabs are fixed at every eighth column. Does not cause autowrap.
	LF	Linefeed – Causes a new line operation.
	VT	Vertical Tab – Processed as a LF.
	FF	Form Feed – Processed as a LF.
	CR	Carriage Return – Moves cursor to left margin on the current line.

Note – The system console is not required to implement any form of flow control for console input or output.

5.1.4 UNICODE Console Output

The UNICODE console is just a UNICODE version of the Basic Console, including UNICODE versions of all the control sequences.

Two Unicode code blocks are required to be present in all systems, so that a valid write to the output console device of a UNICODE character in one of these blocks will result in a recognizable glyph. These blocks are U+0020->U+007E ASCII, for the printing of FAT filenames, and U+2500->U+257F Form and Chart Components, for the printing of lines and borders.

All other Unicode code pages are optional. If a program tries to print a Unicode character that cannot be printed because the corresponding font is not present, then a suitable error character is displayed (e.g. a shaded box).

When a character is printed, the next cursor position is defined by the directionality of the character. The Unicode specification lists the directional properties of all Unicode characters.

The screen control sequences are the same as for the base specification, except that the corresponding Unicode characters are used.

5.2 Operational Characteristics

Console operations are performed using the standard device I/O functions (**Open/Close/Read/Write**), defined in a following section. However, there are certain semantics associated with these functions that are of particular significance to console operation. These are as follows:

- **FileID**s 0 and 1 are preopened by system firmware and correspond to the default console input and console output devices specified in the **ConsoleIn** and **ConsoleOut** environment variables.

The semantics of **Open** are defined to require that it return the lowest available **FileID**. This can be used to change the device associated with the preopened **FileID**s.

- Loaded programs may close either or both of these and open other qualified devices as console input or output devices.
- The system configuration tree includes flags associated with each device. Among the flags are those that indicate whether a device can be opened as a console input device and whether it can be opened as a console output device.

  A device that can be opened for console input or output is opened the same as any other device. That is, the console input and console output devices are opened by calling **Open** with a string based on the path through the configuration description structure, as defined in section Section 4.2.7, “Devices, Partitions, Files, and Path Specifications,” on page 71.

  A protocol specifier, **console(<key>)**, in the path of a device indicates that the input (**Read**) must be returned and output (**Write**) must be accepted in the format defined for the console mode specified by **<key>**.

  Attempts to **Open** a device with **console(<key>)** specified whose configuration flags do not indicate corresponding console functionality fail. Similarly, attempts to **Open** a device in a console mode not supported fail.

  A device supporting both console input and output is opened twice, once for input and once for output. A console device is closed by calling **Close** in the usual way.

Note – The console functionality may be realized in a wide variety of implementations. Possible implementations include internally supported keyboard/display devices; console emulation programs running in other computers connected across networks or serial links; and intelligent TTY devices, connected via communications ports. In the case of console emulation programs or external TTY devices, complete support for the functionality defined above is required. Devices that do not provide such support may not function properly as consoles.

Multiprocessor Platforms 6

This specification is based on the premise that application-level compatibility must be maintained with uniprocessor ARC systems. All ARC-compliant uniprocessor application binaries will run unchanged on ARC MP machines, and it will be possible for application developers to develop ARC MP applications which will run in binary form on uniprocessor systems.

It is possible, however, that certain poorly written uniprocessor applications which assume specific system behavior or which have inherent race conditions may have to be modified to run correctly on an ARC multiprocessor system.

6.1 MP Architecture Overview

The model for ARC multiprocessor systems encompassed by this specification is a homogeneous, symmetric, tightly coupled system in which all processors appear identical to software with respect to their instruction sets and all of system memory is directly addressable by all processors. These systems have a maximum of 32 processors.

The following system elements of an ARC-MP system are specified:

1. Processor Subsystem and Caches

2. I/O Subsystem

3. Interprocessor and I/O interrupts

4. Reset and Boot functions of system firmware

6.2 Processor Subsystem and Caches

6.2.1 Processor Instruction set

The CPUs in ARC-compliant MP machines are required to be based on the MIPS R4000MC instruction set which includes the MIPS I and II instructions, system co-processor instructions, and all privileged instructions. The R4000MC provides instructions for synchronization between multiple processors and for atomically updating memory locations. Restricting ARC-MP machines to using the R4000 instruction set ensures that there is a consistent way of doing these functions on all ARC-compliant MP systems, and the operating system is free to use these instructions directly without incurring the overhead of going through a HAL interface.[1]

6.2.2 User Application Portability Considerations

User applications that utilize the MIPS I instruction set only will operate on all ARC-compliant machines. User applications which require atomic operations must use an ACE OS vendor supplied synchronization library to operate on all ARC machines - both multiprocessor and single-processor.

6.2.3 Symmetry and Shared Memory

On ARC-MP machines, system memory is represented by a uniform global address space accessible from all CPUs. The processor subsystem is symmetric, in the sense that all processors have access to all of system memory at the same physical addresses, and they also have access to all the interprocessor interrupts.

6.2.4 Homogeneity of CPUs

All processors in an ARC-compliant MP system must be identical, as perceived by software, with the following exceptions: cache sizes and clock speeds. The number and format of translation look-aside buffers (TLBs) must be the same for all processors in a system box. All processors must execute the same instruction set with the same semantics. This implies that not only must all processors be of the same architecture, but that they must also be of the same revision level, at least as far as they appear to software. Machines in which differences in revisions or errata among processors are visible to software may not be able to run unmodified ARC software.

Homogeneity extends to the system co-processor (co-processor 0) and floating point unit (co-processor 1). All processors have floating point units per the base ARC specification.

[1] The Hardware Abstraction Layer (HAL) interface is specified by the ACE operating system provider.

6.2.5 Hardware-Enforced Cache Coherency

Processors must see a view of coherent memory which is always consistent regardless of any caching that may be provided (this does not include non-coherent cached mode). This consistency must be maintained by hardware and must be maintained at a granularity of one byte. To be clear, this consistency is required only for memory which has been marked coherent.

Software is responsible for flushing instruction caches after operations that change the instruction stream, so instruction caches need not snoop data writes. Software is also responsible for maintaining TLB consistency with main memory.

6.2.6 Cache Coherency During I/O Transfers

As in the base (single-processor) ARC specification, hardware is not required to maintain cache coherency during I/O data transfers. Operating systems are, however, required to handle such consistency and flush caches before such transfers, through well-defined and separate HAL interfaces, so that systems where hardware does provide cache coherency during I/O transfers can take advantage of the feature by simply defining these cache-flush routines as null procedures.

6.2.7 Atomic Writes

Aligned byte, half-word, word, and double-word reads and writes to both cached and uncached memory must be indivisible. So, for example, page table entry (PTE) updates could be done using aligned double-word writes, without having to acquire and release locks.

6.2.8 Strong Ordering

Strong ordering requirements are the same as in the standard ARC specification for single-processor machines.

6.2.9 Processor Identification

The system vendor will be required to provide an interface in the HAL (perhaps called "whoami") which must return a unique processor ID for each CPU.

The system designer may wish to provide hardware support for such an interface. The processor ID returned by such an interface will be the one that relates to:

- binding of I/O interrupts to that processor.
- sending interprocessor interrupts to that processor.
- the CPU number stored in the configuration database.

6.2.10 Timer Interrupts

Each CPU must receive hardware timer interrupts based on a real-time timer. Such interrupts may either be synchronous or asynchronous, that is, these interrupts do not need to arrive at all processors at the same time. This timer interrupt is an external interrupt and not the one generated by the R4000 on-chip count-compare registers.

6.2.11 Optional Powerfail Interrupt

A powerfail interrupt is optional, but if provided, this interrupt must go to all processors as the highest priority non-error interrupt (only machine-check and other error interrupts may have higher priority).

It is not required that the powerfail interrupt be a hardware interrupt, but hardware must provide the capability to generate this interrupt through software.

6.3 I/O Subsystem

6.3.1 Symmetry

I/O must be symmetric. That is, all I/O devices, including all I/O devices on I/O adapters, are seen at the same physical addresses by all CPUs; I/O interrupts are also symmetric, so that an interrupt from any of these I/O devices can be assigned to any of the processors.

6.4 Interprocessor and I/O interrupts

6.4.1 Interprocessor Interrupts

Every CPU must be able to interrupt any specific CPU on the system, including itself. There is no requirement that the recipient be able to identify the source of an inter-processor interrupt. IP interrupts are the only interrupts one processor sends to another, and there is no message passing associated with these interrupts. Any message passing must be done through shared memory.

Ideally, the hardware allows specifying that any one or a set (including all) of the processors in the machine receive the IP interrupt by writing a mask with a bit per processor. However, this function may be abstracted in the HAL and could be implemented by executing a loop that sends an interrupt to each processor one at a time.

6.4.2 Interprocessor Interrupt Priority

It must be possible to configure IP interrupts to be of higher priority than any I/O interrupts. It must also be possible to mask IP interrupts.

6.4.3 I/O interrupt Assignment

ARC-MP hardware must support the capability to assign I/O interrupts statically at initialization time. Hardware is not required to provide the capability to change assigned interrupts through software once the system is running; however, for dynamically loadable drivers, there must be a means to bind their interrupts to specific CPUs the first time the driver is loaded and initialized.

Dynamic interrupt distribution using various hardware schemes is permitted with the requirement that a given interrupt must be received and handled by exactly one CPU.[2]

Even if dynamic interrupt distribution is provided, capability must still be provided to disable the feature and assign I/O interrupts statically.

6.5 Boot and Reset functions

6.5.1 Boot Master CPU

On ARC-MP machines, there will be a vendor-specific way of selecting the boot master. The boot master then boots exactly as specified in the base ARC specification. CPUs which are not boot masters must remain in a quiescent state in which they do not interfere with the boot process and can be started by the boot master. Any CPU must be capable of accessing or modifying the time value stored in the Real Time Clock (RTC) available on the system.

6.5.2 Starting CPUs

An OS vendor-supplied HAL function will provide the interface to start selected CPUs in kernel mode, with interrupts disabled and at a physical PC address passed as an argument to the function. An OS vendor-supplied HAL routine is required to poll which CPUs are in the system and also their status.

[2] This is not to say that all interrupts should go to a single CPU, but rather to stress that an interrupt must not be handled by more than one CPU.

6.5.3 Program Termination Function Semantics for MP Machines

The system firmware functions — **Halt**, **Restart**, **Reboot** and **EnterInteractiveMode** defined in the base ARC specification — have the following semantics on ARC MP machines:

1. CPUs may be stopped by putting them into the halt state by calling the **Halt** firmware function.

2. A halted CPU will change state by the **Restart** or **Reboot** function.

3. **Reboot** or **Restart** called by any CPU invokes the restart sequence on all CPUs.

4. The **EnterInteractiveMode** firmware function is vendor-specific.

Part 2 — Developing Material

This section is the on-going developmental portion of the ARC Specification. This material is not considered as the last word for these topic areas. In addition, this material is not spread throughout the specification so there may be some redundancy of topic areas. This section exists because the following information was previously in Addenda form and has been moved into the main specification, but not become part of the Base Specification.

Network Bootstrap Protocol Mappings 7

To achieve the interoperability set forth in the ARC Spec objectives, ARC-compliant systems must be able to perform system load from a most boot servers. Network booting of ARC systems is not obliged to involve an ARC server. Therefor the mappings described in this chapter exist to facilitate booting from other systems.

Two protocols, BOOTP/TFTP and DLC RIPL, are defined to be included in the standard set of boot protocols supported by ARC-compliant systems. Compliant systems must support both protocols.

7.1 BOOTP/TFTP/UDP/IP/ARP Protocols

Compliance to the BOOTP/TFTP/UDP/IP/ARP Protocol stack, referenced as the BOOTP protocol stack, must encompass four areas:

1. Basic adherence to the referenced specifications for the protocol.

2. Consistent mapping of the ARC system services defined by the ARC System Interface into the usage of the various packet types defined by the referenced protocol specifications.

3. Clarifications to the protocol’s usage to alleviate any possible inconsistencies in the protocol’s implementation.

4. Requirements and implementation issues of BOOTP protocol boot servers.

7.2 Networked System Partition

To maintain a consistent mapping of ARC I/O services across block, character, and network devices, a System Partition abstraction for network-related file systems/protocols is necessary. This section describes the implementation of this System Partition mapping under both the BOOTP/TFTP and DLC RIPL protocols for ARC-compliant systems.

On block devices, such as fixed disks, the System Partition is a special area on the device media that contains a directory tree where ARC specific information and programs are stored. Likewise, a network boot server should include a directory tree that mirrors the structure of the directory tree on the System Partition of the fixed disk.

This “System Partition Directory Tree” can be anywhere in the server’s directory structure. Multiple System Partition Directory Trees can be included within the server’s directory tree to support more than one workstation. The embedded BOOTP/TFTP, and DLC RIPL protocols in ARC-compliant systems provide access to the System Partition Directory Tree on network boot servers.

7.2.1 BOOTP/TFTP Protocol References

The BOOTP Protocol is defined by two documents: RFC 951, “Bootpstrap Protocol (BOOTP),” September 1985; and RFC 1084, “BOOTP Vendor Information Extensions,” December 1988.

RFC 783 defines the TFTP Protocol. “The TFTP Protocol.” January 1980.

RFC 768 defines the UDP Protocol. “User Datagram Protocol.” August 1980.

RFC 791 defines the IP Protocol. “Internet Protocol, DARPA Internet Program Protocol Specification.” September 1981.

RFC 792 defines the ICMP Protocol. “Internet Control Message Protocol.” September 1981.

RFC 826 defines the ARP Protocol. “An Ethernet Address Resolution Protocol.” November 1982.

RFC 903 defines the RARP Protocol, which is also referenced here. “A Reverse Address Resolution Protocol.” June 1984.

ARP and IP protocol implementations over IEEE 802-type networks are described in RFC 1042, “A Standard for the Transmission of IP Datagrams over IEEE 802 Networks,” February 1988.

Various addresses, port numbers and other important protocol information is specified in RFC 1060, “Assigned Numbers,” March 1990.

The LLC protocol is formally defined in “IEEE Standards for Local Area Networks: Logical Link Control,” IEEE, New York, New York, 1985.

7.2.2 System Interface Mapping

Unlike standard, mass storage-based file systems, the BOOTP protocol does not fit beneath the System Interface defined by this specification in a straightforward manner. This section describes how ARC-compliant systems should implement the BOOTP protocol stack. The following descriptions apply only to network operations when the **Open** specifies the BOOTP protocol.

Open()

ARC systems are required only to implement input mode for BOOT/TFTP files. If an attempt is made to open a file for writing and the implementation does not support write mode, **Open** returns an [ENOFS] status code.

The protocol code should send a BOOTP request frame out onto the network using an IP broadcast address as the destination IP address. A zero IP source address should be used if the workstation does not know its own IP address. If no response is seen after a period of time (possibly adjustable), the protocol code should resend the BOOTP request frame until some suitable number of retries (possibly adjustable) has been performed. If the number of retries has been satisfied, the protocol code should return an [ENOENT] status code. Upon receiving a BOOTP response frame from a boot server, the protocol code should save the BOOTP reply packet in its entirety (including the MAC header).

If the filename given is a file and the **Open** flags are set to read-only, the protocol code should take the file name received in the BOOTP reply packet and issue a TFTP RRQ frame to the server specified by the IP address in the BOOTP reply packet to receive the first packet of file data. The protocol code should wait for the first TFTP DATA frame to be received from the boot server to verify that the file request is valid. After receiving the TFTP DATA frame, the protocol code should save the received data into an internal buffer and return from the **Open** request with a status code of [ESUCCESS]. If a TFTP ERROR packet is received, the protocol code should return the appropriate error to the caller.

If the filename given to the **Open** function is “BOOTPREPLY.PKT” and the **Open** flags are read-only, the protocol code should return a status code of [ESUCCESS] to the caller, if a BOOTP reply packet was saved (from a previous BOOTP transaction). If a BOOTP transaction has never taken place, the protocol code should return an [ENOENT] status code to the caller. All subsequent reads will cause the protocol code to read from its internal copy of the last BOOTP reply packet that it had saved internally.

Read()

Upon being invoked through its **Read** entry point, the protocol code should verify that the file handle is correct and a file has been opened. If the handle is invalid, the protocol code should return an [EBADF] status code. The protocol code should then check for file data from a previously received TFTP DATA frame. If enough data is available to satisfy the **Read** request, the protocol code should fill the **Read** buffer with data, reduce the data-available count, and return a status code of [ESUCCESS]. If the last TFTP DATA received contained less than 512 bytes of data, the protocol code should set the count field to 0 and set the status code to [ESUCCESS].

If not enough data is available to fill the entire **Read** request, the protocol code should fill the **Read** buffer with the available data (if any) and verify that another properly sequenced TFTP DATA frame has been received. If none is available, wait a small period of time (possibly adjustable) until one arrives or send another TFTP ACK frame to request the next file data block. If so, update any internal sequence numbers, send a TFTP ACK frame, and fill the **Read** buffer with the available data. If there is not enough data to satisfy the entire Read request, repeat this procedure.

When handling the **Read** request, if the protocol code determines that it has lost contact with the boot server (that is, a number of retries has been expended), the protocol code should return an [EIO] status code.

Write()

Since TFTP writes are handled in unpredictable ways by a variety of servers, this specification does not require writes to be supported. If not supported, the protocol code should return an [EROFS] status code.

Close()

Upon being invoked through its **Close** entry point, if the protocol code is processing write files, the protocol code should send a TFTP DATA frame to the server with any remaining unsent data (even if that is zero bytes) and wait for the TFTP ACK frame from the boot server. When the ACK frame is received, the protocol code should clear all internal queues, timers, and counters, and cease all communication with the boot server.

If the protocol code is processing read files and the end of file has already been already reached (it received a TFTP DATA frame with 0 to 511 bytes), the protocol code should clear all internal queues, timers, and counters and cease all communication with the boot server. If the end of file had not been reached, the protocol code should send a TFTP ERROR frame to the boot server with a status code of [ESUCCESS] and a message string of “ARC Close File” in the message-string field. It should then clear all internal queues, timers, and counters and cease all communication with the boot server.

GetReadStatus()

Upon being invoked through its **GetReadStatus** entry point, the protocol code should check its internal buffer to ensure that no data remains from the last TFTP DATA frame received. If there is remaining data, the protocol code should return to the caller with a status code of [ESUCCESS]. If the last received TFTP DATA frame contained less than 512 bytes, the protocol code should return an [EAGAIN] status code.

If no data is available, the protocol code should check to see if another properly sequenced TFTP DATA frame has been received. If not, it should wait a small period of time (possibly adjustable) until one arrives or send another TFTP ACK frame to ask for the next file data block. If so, update any internal sequence numbers, send a TFTP ACK frame, and return to the caller with a status code of [ESUCCESS].

When handling the **GetReadStatus** request, if the protocol code determines that it has lost contact with the boot server (that is, a number of retries has been expended), the protocol code should return an [EIO] status code.

Mount()

The BOOTP protocol does not support mounting files or devices; therefore, the protocol code should return an [EINVAL] status code.

Seek()

Upon being invoked through its **Seek** entry point, the protocol code should verify that the seek operation is to a valid file handle. If not, the protocol code should return an [EBADF] status code. The protocol code should be able to handle both absolute and relative seek requests by maintaining a file position variable.

If the file handle is valid and is forward of the current file position, the protocol code should check if the new location begins within the current data buffer of the last TFTP DATA frame received. If it does not, the protocol code should check to see if another properly sequenced TFTP DATA frame has been received. If not, it should wait a small period of time (possibly adjustable) until one arrives or send another TFTP ACK frame to ask for the next file data block.

If there is a TFTP DATA frame available, update any internal sequence numbers, send a TFTP ACK frame, and adjust the current file pointer location. If there was not enough data to satisfy the **Seek** request, repeat this procedure. If the end of the file has arrived and the new file position is beyond the end of file, the protocol code should set the file position to the byte beyond the last byte received and return the [EINVAL] return code.

If the file handle is valid and the new file location is behind current file position, the protocol should issue a TFTP ERROR packet to the server indicating an ARCSS close file (see the description of Close request handling). The protocol code should then send another TFTP RRQ packet to reopen the file from the beginning and repeat the procedure specified in the previous paragraph to seek to the proper file position desired by the user.

GetDirectoryEntry()

Since the TFTP protocol does not support directories, the protocol code should return the [EINVAL] status code to the caller.

7.2.3 Protocol Clarifications

To ensure that implementations of the BOOTP protocol stack are consistent among all ARC-compliant systems, several aspects of the BOOTP protocol suite must be clarified. This clarification is necessary for several reasons. First, the various RFCs that define the BOOTP protocol suite referenced here were written to allow the maximum amount of flexibility, thus allowing incompatible implementations of the protocol to exist. Furthermore, several fields in the defined packets are used to provide general information without specifying their particular use. This specification defines the contents of these fields to guarantee all ARC-compliant implementations are equivalent. This implementation will not affect the operation of currently existing BOOTP boot servers.

Token-Ring MAC Requirements

Token-ring network devices that implement the BOOTP protocol evoke two considerations. First, the IP and ARP protocol implementations must support source routing in the Token-ring MAC header. This support allows the system to communicate over source-routing bridges. Second, use of the LLC Class I protocol is required for all IEEE 802-type networks. The LLC code should sit under the IP and ARP protocols. For details on these topics, refer to RFC 1042.

If source routing is not supported, the system will be incapable of loading programs from BOOTP boot servers located on other LANs separated by source-routing bridges.

Ethernet MAC Requirements

When the BOOTP protocols are implemented on an Ethernet network device, the IP and ARP protocol implementations support the older Ethernet MAC header format and, optionally, the IEEE 802.3 MAC header format. (Most existing boot servers that use the BOOTP protocol suite still use the Ethernet MAC header format, so it is supported herein.) In addition, if IEEE 802.3 MAC header formats are supported, LLC Class I must also be supported as described in RFC 1042.

LLC Requirements

When the BOOTP protocols are implemented over an IEEE 802-type network, Class I LLC protocol must be used. The use of the LLC Class I protocol is specified in RFC 1042.

BOOTP Request Frame Requirements

The following paragraphs provide necessary clarifications on the usage of the fields of the BOOTP request frame.

htype Field: The htype field indicates the type of network hardware address used. This also implies the type of network hardware used for the read or write requests that the TFTP protocol code is preparing to perform. RFC 951 describes only the Ethernet hardware address type value. Additional hardware address type values are defined in RFC 1060, under the section entitled, “Address Resolution Protocol Parameters.” Another hardware type of note (other than the Ethernet type) is the IEEE 802 type, which can be used to describe both IEEE 802.3 Ethernet and Token-Ring hardware address types.

vend Field: This standard does not restrict the contents of the vend field. If the vend field is used, the protocol code should be cognizant of the fact that the remote boot server may not support the BOOTP request in the desired manner (that is, it may not support interpreting the values placed in the vend field).

BOOTP Response Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of the BOOTP response frame vend field.

vend Field: This standard places no requirements on the contents of the vend field. The protocol code should not be implemented in a manner that should require any information to be placed into the vend field by the BOOTP server.

TFTP RRQ Frame Requirements

The following paragraphs provide necessary clarifications on the usage of several of the TFTP RRQ frame fields.

Filename Field: The Filename field of a TFTP RRQ frame should contain the pathname, specified by the Root Marker in the System Partition Directory File, concatenated with the filename passed in the Open request.

Mode Field: The Mode field should be set to “octet,” signifying raw 8-bit transfers. The ARC Specification Standard only requires support for the “octet” mode of file transfer.

TFTP ERROR Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of several of the TFTP ERROR frame fields.

ErrorCode and ErrMsg Fields: The ErrorCode field is defined by RFC 783 to acknowledge any TFTP packet and to signify an error or alert condition to the opposite party of the TFTP connection. The values for this field are defined in RFC 783. One additional area to cover is when a Close request is processed by the protocol code in the workstation and the end of the file transfer has not been reached. In this case, the ARC-compliant protocol code should set the status code field to zero (0) and set the ErrMsg field to "ARC Close File."

ICMP Frame Requirements

The ICMP protocol, specified in RFC 972, must be implemented in an ARC-compliant system. The purpose of the ICMP protocol is to provide greater error information regarding network communications problems at the IP layer of the protocol suite. The following paragraphs provide the necessary clarifications on the usage of several of the ICMP frame fields.

Type: An ARC-compliant system should be capable of receiving and processing ICMP packets with the type field set to the following:

ICMP Message Type	Type Code
Destination Unreachable	3
Redirect	5
Echo	8
Time Exceeded	11
Parameter Problem	12

An ARC-compliant system should be able to generate ICMP packets with the type field set to the following:

ICMP Message Type	Type Code
Echo Reply	0

ARP Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of several of the ARP frame fields.

ar$hrd Field: The ar$hrd field indicates the type of network hardware address used. This also implies the type of network hardware used for the read or write requests that the TFTP protocol code is preparing to perform. RFC 826 describes only the Ethernet hardware address type value. Additional hardware address type values are defined in RFC 1060 under the section entitled, "Address Resolution Protocol Parameters." Another hardware type to note is the IEEE 802 type, which can be used to describe both IEEE 802.3 Ethernet and Token-Ring hardware address types.

7.2.4 Server Considerations

When considering the implementation of the protocols to support remote file loading from a network, the server implementation must be considered as well as the workstation implementation. The following paragraphs describe several server issues which affect the ARC definition of the BOOTP protocol implementation.

BOOTP vs. RARP

Both the BOOTP and RARP protocols are prevalent in existing networks. However, ARC-compliant workstations are only required to support the BOOTP protocol suite; therefore, a server must support the BOOTP protocol to support all ARC-compliant workstations. A server that only supports RARP will only be able to boot workstations that have optionally supported the RARP protocol.

LLC Support

BOOTP servers, as well as RARP servers, must support the LLC implementation as laid out in RFC 1042 if operation over standard IEEE 802-type networks is desired.

Filename Support

All BOOTP and TFTP servers that wish to support loading files to ARC-compliant systems must support the usage of filenames in the filename fields of the BOOTP request/response and TFTP RRQ frames.

7.3 IBM DLC RIPL/LLC Protocols

Compliance to the IBM DLC RIPL/LLC Protocol stack, referenced as the RIPL protocol stack, must encompass four areas:

1. Basic adherence to the referenced specifications for the protocol.
2. Consistent mapping of the ARC-system services defined by the ARC-System Interface into the usage of the various packet types defined by the referenced protocol specifications.
3. Clarifications to the protocol’s usage to alleviate any possible inconsistencies in the protocols implementation.
4. Requirements and implementation issues of RIPL protocol boot servers.

7.3.1 Protocol References

The IBM DLC RIPL Protocol for Token-ring is defined in Appendix B of the "IBM Token-ring Network Remote Program Load User's Guide," Third Edition.

The IBM DLC RIPL Protocol for Ethernet is defined in Appendix B of the "IBM Ethernet Networks User's Guide," First Edition. The differences between the Token-ring and Ethernet versions of this protocol are minimal; therefore, their implementation under ARC should be considered the same protocol.

The LLC protocol is formally defined in "IEEE Standards for Local Area Networks: Logical Link Control," IEEE, New York, New York, 1985.

7.3.2 System Interface Mapping

Unlike standard mass storage-based file systems, the RIPL protocol does not fit beneath the System Interface defined by this specification in a straightforward manner. This section describes the implementation of the RIPL protocol stack for ARC-compliant systems. The following descriptions apply only to network operations when the **Open** specifies the RIPL protocol.

Open()

Upon being invoked through its **Open** entry point, the protocol code should verify that the **Open** request flags are set for read-only because the RIPL protocol only supports reading files. If not set correctly, the protocol code should return an [EROFS] status code.

If the **Open** flags are set to read-only, the protocol code should then send a FIND request with the filename in the filename field to establish a connection to the server. Upon receiving the FOUND frame from the boot server, the protocol code should then take the concatenated filename and issue a SEND.FILE.REQUEST frame to the boot server to receive the first packet of file data.

The protocol code should wait for the first FILE.DATA.RESPONSE frame to be received from the boot server to verify that the file request is valid. After receiving the FILE.DATA.RESPONSE frame, the protocol code should save the received data into an internal buffer and return from the **Open** request with a status code of [ESUCCESS]. If a LOAD.ERROR packet is received or the server does not respond to the FIND frame, the protocol code should return the appropriate error to the caller.

Read()

Upon being invoked through its **Read** entry point, the protocol code should verify that the file handle passed to it corresponds to a file opened with an open request. If not, the protocol code should return an [EBADF] status code. The protocol code should then check for available file data from a previously received FILE.DATA.RESPONSE. If enough data is available to satisfy the **Read** request, the protocol code should fill the **Read** buffer with data, reduce the data available count, and return a status code of [ESUCCESS].

If not enough data is available to fill the entire Read request, the protocol code should fill the Read buffer with the available data (if any) and check to see if another properly sequenced FILE.DATA.RESPONSE frame has been received. If not, it should wait a small period of time (possibly adjustable) until one arrives or send another SEND.FILE.REQUEST to ask for the next file data block.

If so, update any internal sequence numbers, send a SEND.FILE.REQUEST frame if the ACK bit was set in the flags field of the just received FILE.DATA.RESPONSE frame, and fill the Read buffer with the available data. If there was not enough data to satisfy the entire Read request, repeat this procedure.

When handling the Read request, if the protocol code determines that it has lost contact with the boot server (e.g. a number of retries has been expended), the protocol code should return an [EIO] status code.

Write()

The RIPL protocol does not support writing files to boot servers; therefore, the protocol code should return an [EROFS] status code.

Close()

If the protocol code was performing read file processing, it should clear all internal queues, timers, and counters, and cease all communication with the boot server.

GetReadStatus()

Upon being invoked through its **GetReadStatus** entry point, the protocol code should check its internal buffer for data remaining from the last FILE.DATA.RESPONSE frame received. If data is available, the protocol code should return to the caller with a status code of [ESUCCESS]. If the last FILE.DATA.RESPONSE frame had the EOF flag set, the protocol code should return an [EAGAIN] status code.

If no data is available, the protocol code should check for another properly sequenced FILE.DATA.RESPONSE. If none is available, it should wait a small period of time (possibly adjustable) until one arrives or send another SEND.FILE.REQUEST frame to ask for the next file data block. The protocol code should then update any internal sequence numbers, send a SEND.FILE.REQUEST frame (if necessary) to acknowledge the received data, and return to the caller with a status code of [ESUCCESS].

When handling the **GetReadStatus** request, if the protocol code determines that it has lost contact with the boot server (that is, a number of retries has been expended), the protocol code should return an [EIO] status code.

Mount()

The RIPL protocol does not support mounting files or devices; therefore, the protocol code should return an [EINVAL] status code.

Seek()

Upon being invoked through its **Seek** entry point, the protocol code should verify that the file handle corresponds with an opened file. If not, the protocol code should return an [EBADF] status code.

If the file handle is valid and the new file location is forward of the current file position, the RIPL protocol code should check if the new location begins within the current data buffer of the last FILE.DATA.RESPONSE frame received. If it is not, the protocol code should check to see if another properly sequenced FILE.DATA.RESPONSE frame has been received. If none has been received, it should wait a small period of time (possibly adjustable) until one arrives or send another SEND.FILE.REQUEST to ask for the next file data block.

If there is a FILE.DATA.RESPONSE available, the RIPL protocol code should update any internal sequence numbers, send a SEND.FILE.REQUEST frame if the ACK bit was set in the flags field of the just-received FILE.DATA.RESPONSE frame, and adjust the current file pointer location. If there was not enough data to satisfy the **Seek** request, repeat this procedure. If the end of file is detected before the seek operation is satisfied, the protocol code should set the current file position to the byte after the last byte received and return the [EINVAL] status code.

If the file handle is valid and the new file location is before the current file position, the protocol code should perform the procedures to close the file and then use the procedures to reopen the same file. The protocol code may then use the procedures described in the previous paragraphs to seek to the proper file position.

7.3.3 Protocol Clarifications

To ensure that implementations of the RIPL protocol stack are consistent among all ARC-compliant systems, several aspects of the RIPL protocol must be clarified. Since the RIPL protocol specifications referenced here were written to allow the maximum amount of flexibility, some incompatible implementations of the protocol may exist. Furthermore, several fields in the defined packets are used to provide general information without specifying their particular use. This specification defines the contents of those fields to guarantee compatibility among all ARC-compliant implementations.

Finally, several fields defined in the IBM DLC RIPL protocol were specified to carry IBM PC and compatible hardware information. This information is irrelevant for the target systems of this specification; therefore, these fields are redefined to provide the proper information for ARC-compliant systems. This implementation will not affect the operation of existing RIPL boot servers.

Token-Ring MAC Requirements

When the RIPL protocol is implemented over a Token-ring network device, the RIPL and LLC protocol implementation must support source routing in the Token-ring MAC header. This support will allow the system to communicate over source-routing bridges.

Systems that do not support this feature will be incapable of loading programs from RIPL boot servers located on other LANs separated by source routing bridges.

Ethernet MAC Requirements

When the RIPL protocol is implemented over an Ethernet network device, the RIPL and LLC protocol implementation must use the IEEE 802.3 format MAC header format, not the Ethernet MAC header format.

Systems that do not support this feature will be incapable of loading programs from most standard RIPL boot servers.

FIND Frame Requirements

Several of the fields in the FIND frame header are either defined for the personal computer environment or are ambiguously defined by the referenced specifications. The following paragraphs provide necessary clarifications on the usage of these fields.

Machine_Conf and Equip_Flags Fields: These two fields are defined by the RIPL specifications to be filled in with values returned by the PC BIOS INT 15 and INT 11 calls respectively. Since the configurations of ARC-compliant machines vary widely, it is not practical to reproduce this information for an ARC system. Therefore, these two fields are combined into a single 10-byte field representing the 8-byte Vendor ID returned by the ARCSS GetSystemId function call. The format of this field is 2 bytes of 0xFF, followed by the 8-byte Vendor ID. The 2 bytes of 0xFF can be used by a boot server to differentiate this RIPL request from RIPL requests originating from industry standard personal computers.

This modification to the FIND frame should not cause compatibility problems, since no known RIPL servers today use this field.

Memory_Size Field: The Memory_Size field is defined by the RIPL specifications to be filled in by a value returned by the PC BIOS INT 12 call. Since the PC BIOS INT 12 call returned only available memory below the 1-Mbyte boundary, it does not provide the type of information that is needed by ARC-compliant systems, which provide at least 4 Mbytes of memory, for downloading files from the boot server. Therefore, this field is redefined to be the amount of contiguous memory available to stand-alone programs (applications and OS loaders) in the ARC-compliant system, specified in 1 Kbyte amounts. If there are 64 Mbytes or more of available memory, this field should be set to 0xFFFF. The value placed into this field may be obtained through the **GetMemoryDescriptor** ARC function call.

This modification to the FIND frame should not cause any compatibility problems, since no known RIPL servers use this field.

Conn_Class Field: The Conn_Class field is defined by the RIPL specifications to be the type of RIPL service supported by the workstation. For the purposes of this specification the following shall apply:

- Token-ring – the RIPL implementation must support at a minimum setting bit 0, “Class I” support. Bits 1-3 may optionally be set. These bits are defined in the Token-ring RIPL specification listed in the reference section.
- Ethernet – the RIPL implementation may optionally set bits 0-1. These bits are defined in the Ethernet RIPL specification listed in the reference section.

Remote_Program_Load_EC Field: The Remote_Program_Load_EC Field is defined to be a 10-byte field that specifies the current revision level of the RIPL protocol stack code. For the purposes of this specification, this field will contain the 8-byte Product ID returned from the ARCSS **GetSystemId** call followed by 2 bytes of specification reference. The format is as follows:

8-byte Product ID | ARC RIPL Level (0 for this specification)

File_Name_Header and File_Name Fields: The File_Name field is defined by the RIPL specifications to be the name of the file to download from the boot server or a special data field if the first byte has a value greater than 0x80. For all ARC-compliant systems, the File_Name fields should always contain the name of the file or directory to be opened. The first byte of the file name should always be a value between 0x00 and 0x7F. In addition, the File_Name_Header field defines the length of the filename specified in the File_Name field.

The RIPL specifications state that the File_Name field should be no longer that 80 (decimal) bytes long. To maintain compatibility with existing RIPL boot servers, an ARC-compliant implementation of the RIPL protocol stack should limit filename sizes to 80 or fewer characters. This restriction is, however, imposed on the filename passed as part of the path parameter during an **Open** call. Because the filename-length byte within the File_Name_Header field is a full byte, this specification only limits the filename to 251 bytes in length; however, implementations that allow the length of filename to exceed 80 (decimal) bytes may not be able to load files properly from existing RIPL boot servers.

The File_Name field should be filled with the filename portion of the path specified during the **Open** call to the RIPL protocol stack code. The filename may be either a filename or a directory name. Server implications of this feature of the protocol are specified in the section discussing server considerations.

FOUND Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of several of the FOUND frame fields.

Conn_Class Field: When expecting and receiving a FOUND frame, the RIPL protocol stack code should verify that the connection class supported by the server matches one of the bits set in the Conn_Class field of the FIND frame sent to the RIPL boot server.

Source_Addr and RSAP Fields: The RIPL protocol stack code should use the Source_Addr and RSAP fields in the FOUND frame to build the MAC and LLC headers for SEND.FILE.REQUEST frames going to that RIPL boot server.

SEND.FILE.REQUEST Frame Requirements

Several of the fields in the SEND.FILE.REQUEST frame header are either defined for the personal computer environment or are ambiguously defined by the referenced specifications. The following paragraphs provide necessary clarifications on the usage of these fields.

RSAP Field: The RIPL specification states that the workstation should set the RSAP field to the value found in the RSAP field in the received FOUND frame. However, current implementations have this field set to the same SAP value that the workstation used to receive FOUND frames. Therefore, it is recommended that this field be set to the SAP value to which the RIPL protocol stack code wishes to receive FILE.DATA.RESPONSE frames.

Machine_Conf and Equip_Flags Fields: These two fields are defined by the RIPL specifications to be filled in with values returned by the PC BIOS INT 15 and INT 11 calls, respectively. Since the configurations of ARC-compliant machines vary widely, it is not practical to reproduce this information for an ARC system. Therefore, these two fields are combined into a single 10-byte field representing the Vendor ID as returned by the ARC **GetSystemId** function call. The format of this field is 2 bytes of 0xFF, followed by the 8-byte Vendor ID. The 2 bytes of 0xFF can be used by a boot server to differentiate this RIPL request from RIPL requests originating from industry standard personal computers.

This modification to the SEND.FILE.REQUEST frame should not cause any compatibility problems, since no known RIPL servers use this field.

Memory_Size Field: The Memory_Size field is defined by the RIPL specifications to be filled in by a value returned by the PC BIOS INT 12 call. Since the PC BIOS INT 12 call returns only available memory below the 1-Mbyte boundary, it does not provide the type of information that is needed by ARC-compliant systems, which provide at least 4 Mbytes of memory, for downloading files from the boot server. Therefore, this field is redefined to be the amount of contiguous memory available to stand-alone programs (applications and OS loaders) in the ARC-compliant system, specified in 1-Kbyte amounts. If there are 64 Mbytes or more of available memory, this field should be set to 0xFFFF. The value placed into this field may be obtained through the **GetMemoryDescriptor** ARC function call.

This modification to the SEND.FILE.REQUEST frame should not cause compatibility problems, since no known RIPL servers use this field.

Remote_Program_Load_EC Field: The Remote_Program_Load_EC Field is defined to be a 10-byte field that specifies the current revision level of the RIPL protocol stack code. For the purposes of this specification, this field will contain the 8-byte Product ID returned from the ARCSS GetSystemId call, followed by 2 bytes of specification reference. The format is as follows:

8-byte Product ID | ARC RIPL Level (0 for this specification)

File_Name_Header and File_Name Fields: The File_Name field is defined by the RIPL specifications to be the name of the file to download from the boot server. It may also be a special data field if the first byte has a value greater than 0x80. For all ARC-compliant systems, the File_Name fields should always contain the name of the file or directory to be opened. The first byte of the filename should always be a value between 0x00 and 0x7F. In addition, the File_Name_Header field defines the length of the filename specified in the File_Name field.

The RIPL specifications state that the File_Name field should be no longer that 80 (decimal) bytes long. To maintain compatibility with existing RIPL boot servers, an ARC-compliant implementation of the RIPL protocol stack should limit filename sizes to 80 or fewer characters. This restriction is, however, actually imposed on the filename passed as part of the path parameter during an **Open** call. Because the filename-length byte within the File_Name_Header field is a full byte, this specification only limits the filename to 251 bytes in length; however, implementations that allow the length of filename to exceed 80 (decimal) bytes may not be able to load files properly from existing RIPL boot servers.

The File_Name field should be filled in with the filename portion of the path specified during the **Open** call to the RIPL protocol stack code. The filename may be either a filename or a directory name. Server implications of this feature of the protocol are specified in the section discussing server considerations.

FILE.DATA.RESPONSE Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of several of the FILE.DATA.RESPONSE frame fields.

Flags Field: The following are how each of the flag bits should be handled by ARC-compliant RIPL protocol stack:

LOCATE_ENABLE – This bit should always be ignored by an ARC-compliant RIPL protocol stack since all loadable files should be relocatable.

XFER_ENABLE – This bit should always be ignored by an ARC-compliant RIPL protocol stack since it is the system firmware's responsibility to establish the entry points to the loaded file.

ACK_REQUEST – An ARC-compliant RIPL protocol stack should always use the ACK_REQUEST bit to determine if repeated SEND.FILE.REQUEST frames need to be sent to acknowledge FILE.DATA.RESPONSE frames.

Locate_Addr Field: This field should be ignored by all ARC-compliant RIPL protocol stacks.

Xfer_Addr Field: This field should be ignored by all ARC-compliant RIPL protocol stacks.

File_Data Field: All ARC-compliant RIPL protocol stacks should perform data blocking on incoming file data to match incoming read requests with the associated file data from incoming FILE.DATA.RESPONSE frames.

LOAD.ERROR Frame Requirements

The following paragraphs provide the necessary clarifications on the usage of several of the LOAD.ERROR frame fields.

Error_Code Field: If the Error_Code field is 0x0002 (File_not_found), an ARC-compliant RIPL protocol stack should return an [EEXIST] status code on the current open request or the next read request.

If the Error_Code field is 0x0000 (Restart), an ARC-compliant RIPL protocol stack should reset its sequence number and start rereading the file from the server until the proper location in the file is reached to continue handling read requests.

PROGRAM.ALERT Frame Requirements

ARC-compliant RIPL protocol stacks are not required to send PROGRAM.ALERT frames. The RIPL protocol code should return an error from the **Open** or **Read** call. If PROGRAM.ALERT frames are sent to the RIPL boot server, the RIPL protocol code should do so for a limited period of time before returning control to the system firmware.

7.3.4 Server Considerations

When considering the implementation of the protocols to support remote file loading from a network, the server implementation must be considered as well as the workstation implementation. The following paragraphs describe several server issues which affect the ARC definition of the RIPL protocol implementation.

Minimum Conn_Class Support: A RIPL server that is to support remote loading to ARC-compliant systems must support a minimum connection class (specified in the Conn_Class field). This support varies slightly between Token-ring and Ethernet.

For Token-Ring, the RIPL server must at least support a connection class of Class I. The server may optionally support Class II, broadcast to functional address, and broadcast to group address. However, the server must support requests coming from workstations that only support Class I.

For Ethernet, the Ethernet RIPL specification implies that only Class I is supported; therefore, the RIPL server must support Class I (there is not bit to set or reset). The RIPL server may optionally support broadcast to group address or functional address (both are the same as multicast address).

Class I support in both the Token-ring and Ethernet cases implies that the RIPL server supports sending FOUND, FILE.DATA.RESPONSE, and LOAD.ERROR frames to the workstation's individual hardware address (not broadcast or multicast). The RIPL server may only send these frames to a multicast address if the workstation has the corresponding bit set in the Conn_Class field of the FIND frame.

Frame Reception: RIPL servers must be capable of receiving RIPL frames on the RIPL functional address (0xC00040000000) for Token-ring and on the RIPL multicast address (0x030002000000) for Ethernet.

Machine_Conf, Equip_Flags, and Memory_Size Field Support: Optionally, RIPL servers may use the Machine_Conf, Equip_Flags, and Memory_Size fields of received FIND, SEND.FILE.REQUEST, and PROGRAM.ALERT frames. It is not required. If used, the RIPL server should be able to distinguish PC workstations from ARC-workstations. All RIPL servers implemented today do not use these fields.

File Name Support: All RIPL servers that wish to support loading files to ARC-compliant systems must support the usage of filenames in the File_Name field of the FIND, SEND.FILE.REQUEST, and PROGRAM.ALERT frames.

Glossary

Abstraction A representation of an entity that is disassociated from any specific instance of that entity.

Adapter A defined means for attaching a wide variety of optional devices to a system. As used herein, it is synonymous with bus.

Advanced Computing Environment (ACE) A consortium to design and promote the next generation of desktop PCs.

Advanced RISC Computing (ARC) A specification for a RISC-based desktop system.

Application Binary Interface (ABI) A standard interface based on the executable binary image of a compiled program. Allows programs to be run on different machines without having to recompile them.

Application Programming Interface (API) The set of system calls available to a program. The source code of an application calls this to standardize on the program/OS interface.

Bus A defined means for attaching a wide variety of optional devices to a system. As used herein, it is synonymous with adapter.

Central Processing Unit (CPU) The chip which is the control center for determining what work is to be performed and actually performing the work.

Control Sequence Introducer A character sequence which is recognized by the console that allows console behavior to be altered.

Electronically Programmable Read-Only Memory (EPROM) A chip that can have software burned into it and not lose the information when power is turned off.

Extended Industry Standard Architecture (EISA) An extension to the ISA bus that allows for 32-bit cards and higher bus speeds.

Firmware The instructions/programs that normally reside in an EPROM, which are used to boot a machine.

Firmware State The state of the computer when it is being run by the firmware routines.

Floating Point Unit (FPU) The chip which executes instructions for processing floating point numbers.

Hardware The physical components that make up a computer.

Hardware Adaptation Layer (HAL) Interface between the system hardware and the operating system environment.

Hardware Abstraction Layer See Hardware Adaptation Layer.

Homogeneous Multiprocessor Architecture All processors are physically the same, down to the make and model. For example, each CPU in the system must be a MIPS 3000A.

Industry Standard Architecture (ISA) The de facto standard system bus used in today's desktop PCs.

MIPS Manufacturer of R4000 CPU.

Multiprocessor A computer which uses more than one CPU for processing work.

NT Microsoft operating system named for "New Technology." Based on multiuser, multiprocessor object-oriented design.

Operating System (OS) The software that controls the operating of the computer, for accessing files, running programs, and managing resources.

Program State The state when the computer is being run by the operating system.

Real Time Clock (RTC) The system clock that can measure time in the real world.

Reduced Instruction Set Computer (RISC) A CPU instruction set that is designed to have a minimal number of instructions that run quickly.

Shared-Memory Multiprocessing When multiple processors share the same physical memory.

Shrink-Wrap Software Mass produced and distributed software that executes ,without modification, on any platform from any vendor in a target class of systems.

Symmetric Multiprocessing Architecture (SMP) All processors see the same view of the world; i.e. they all see the same physical memory, peripherals, etc. This simplifies and unifies the software that runs on each CPU.

System Load The process of loading, from some media, the operating system so that the system will run and perform work.

System Partition A segmented portion of the boot hard drive that stores the operating system software.

Tightly Coupled Multiprocessing Architecture All processors see and access the same physical address space. All information within memory is global to each CPU.

Translation Look-aside Buffer (TLB) A mechanism the operating system can use to gain access to firmware routines while the machine is in the program state.

Topology The manner of configuration or arrangement of components making up a system.

Uniprocessor A computer which has only one CPU to perform work.

Unix Multiuser multiprocessing operating system.

Index

A

ABI *See* Application Binary Interface
ANSI Standard X3.131-1990, 39
ANSI X3.64, 104
Application Binary Interface, 23
Architecture
    Components of, 19
Asynchronous Serial Port *See* Serial Interface
Audio Port
    Requirements of, 35

B

BootMasterId, 53
BOOTP, 116
    Close, 119
    Filename Support, 124
    GetDirectoryEntry, 121
    GetReadStatus, 120
    LLC Support, 124
    Mount, 120
    Open, 118
    Read, 119
    Response Frame Requirements, 122
    Seek, 120
    Server Considerations, 124
    TFTP Error Frame Requirements, 123
    TFTP RRQ Frame Requirements, 122
    vs. RARP, 124
    Write, 119
BootStatus, 53
Buses
    I/O, 25
Byte Ordering
    Little Endian, 23

C

Cache
    Organization of, 44
Cache Architecture, 31
CD-ROM, 43
Character Set, 103
Checksum, 54
Close, 82, 108
Compliant Systems
    Elements of, 28
    Optional Devices, 37
Configuration Query Functions, 79
    GetConfigurationData, 81
ConsoleIn, 55
ConsoleOut, 55
Control Sequence Introducer, 103
Control Sequences, 105
CSI *See* Control Sequence Introducer

D

Data Exchange
    Media Compatibility, 14
Data Interchange Standards, 49
Data Ordering, 30
DebugBlock, 52
Directory Operations
    GetDirectoryEntry, 95
Directory Tree, 74
Diskette, 42

E

EnterInteractiveMode, 79
Environment Variable Functions, 91
    GetEnvironmentVariable, 92
    SetEnvironmentVariable, 91
Environment Variables
    Software Loading, 56
    Standard, 55
    TimeZone, 57
Ethernet, 40
    Connectors Supported, 40
Exception Block, 51
ExecAddr, 75
Executable Image
    Requirements of, 50
Execute, 77
Execution Environment
    Initial, 51

F

File Header Flags, 50
File Header Magic, 50
Firmware
    Standard Device I/O Functions, 46
    System, 26
Firmware Functions, 47
FirmwareVector, 52, 96
FirmwareVectorLength, 52
Fixed Disk, 43
FPU
    Requirements of, 31

G

GetConfigurationData, 81
GetDirectoryEntry, 95
GetEnvironmentVariable, 92
GetMemoryDescriptor, 93
GetReadStatus, 85, 120
GetSystemId, 92
GetTime, 94
Global Pointers, 50

H

HAL *See* Hardware Abstraction Layer
Halt, 78
Hard Disk *See* Fixed Disk
Hardware Abstraction Layer, 23
    Functionality Abstracted, 24

I

I/O Buses, 25
    Addenda for, 46
    EISA, 25
    Futurebus+, 25
    Requirements of, 46
    Type Identifier, 46
    VMEbus, 25
I/O Functions *See* Input/Output Functions
IEEE 802.5, 41
Input/Output Functions, 82
    Close, 84
    GetReadStatus, 85
    Mount, 88
    Open, 83
    Read, 85
Interfaces
    Standard, 26
Invoke, 76
ISO/IEC 9945-1, 57

K

Keyboard, 32
    Layout of, 33
    Requirements of, 33

L

Libraries
    Shared, 50
Little Endian, 23, 30
Loadable Image, 50
Loadable Services, 24

M

Media Formats, 41
    CD-ROM, 43
    Diskette, 42
    Fixed Disk, 43
    System Load Requirements, 42
Memory
    Minimum Requirements, 44
    System Load Requirements, 44
Memory Query Functions
    GetMemoryDescriptor, 93
MIPS
    COFF, 75, 77
MIPS I, 23, 30
MIPS II, 23, 30
Mount, 88

N

Network Interface
- Ethernet, 40
- Token Ring, 41

Network Protocols
- BOOTP, 116, 117
- BOOTP Request Frame Requirements, 122
- Clarifications, 121
- DLC RIPL, 116, 124
- Ethernet MAC Requirements, 128
- LLC Requirements, 121
- References, 117

Network Protocols for System Load, 44

O

Object Formats, 50, 74

Open, 83, 118, 125

Operating Environments, 14

Operational States, 20

Optional Header Magic, 50

OSLoader, 56

OSLoadFilename, 56

OSLoadOptions, 56

OSLoadPartition, 56

P

Parallel Interface
- Characteristics of, 38

Parallel Port
- Requirements of, 38

Peripheral Attachment Subsystems, 46

Pointing Device
- Characteristics of, 33
- Requirements of, 33

Processing Subsystems, 22, 25, 30, 44

ProcessorId, 53

ProductId, 92

Program Termination Functions, 78
- EnterInteractiveMode, 79
- Halt, 78
- PowerDown, 78
- Reboot, 79
- Restart, 78

Protocol References, 117
- RIPL, 125

R

Real Time Clock, 32

Reboot, 79

Restart, 78

Restart Block, 52

RIPL
- Close, 126
- Ethernet MAC Requirements, 128
- FIND Frame Requirements, 128
- FOUND Frame Requirements, 130
- GetReadStatus, 126
- Mount, 127
- Open, 125
- Protocol Clarifications, 127
- Read, 126
- Seek, 127
- Server Considerations, 133
- System Interface Mapping, 125
- Token-Ring MAC Requirements, 128
- Write, 126

RTC, 32

S

SavedStateArea, 53

SCSI Interface
- Characteristics of, 39
- Requirements of, 39

Seek, 82

Serial Interface
- Characteristics of, 38

Serial Ports
- Requirements of, 37

SetEnvironmentVariable, 91

Shared Libraries, 50

Software Loading Functions, 75

StackAddr, 76

Standard Data Structures
- Exception Block, 51
- System Parameter Block, 51

Standard Interfaces, 26
- Firmware, 26
- Hardware Abstraction Layer, 23

Standard Subsystems, 25
- I/O, 46

Standard System Elements, 28, 29
- Audio Port, 35
- Deviation from, 29
- Hardware, 28
- Keyboard, 32, 33
- Network Interface, 40
- Parallel Interface, 38
- Pointing Device, 33
- SCSI Interface, 39
- Serial Interface, 38
- Video Subsystem, 34

Status Codes, 48

Subsystems
- I/O, 25
- Processing, 25
- Standard, 25

System Console, 102
- Control Sequences, 103
- Functionality of, 102
- Operational Characteristics, 108
- Output Functions, 104

System Firmware, 26
- Operational Phases of, 21
- Temporary, 101

System Functions, 47
- Parameter Passing, 48
- Status Codes, 48

System ID Query Function
- GetSystemId, 92
- ProductId, 92
- VendorId, 92

System Interface Mapping, 118

System Load
- Network Protocols, 43
- Sources of, 14

System Parameter Block, 51
- Address of, 51
- Structure of, 51

System Partition, 56, 73
- Directory Tree, 74
- Media Formats, 42
- Networked, 117
- Pathnames in, 74

System Timer, 32

T

Time Functions
- GetTime, 94
- TimeZone, 94

TimeZone, 57, 94

Token Ring, 41

TopAddr, 75

TTY Devices, 108

V

VendorId, 92

Video Subsystem
- Characteristics, 34
- Implementation of, 35
- Requirements of, 34