DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers

User Manual

Order Number: EK–VME54–UM. A01

This manual provides an introduction to the Alpha VME 5/352 and 5/480 single-board computers (SBCs), explains how to use the console firmware, and discusses diagnostics and troubleshooting.

Revision/Update Information: This is a new manual.

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, October 1997

FCC Notice:
This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his own expense.

Warning!
This is a Class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

Achtung!
Dieses ist ein Gerät der Funkstörgrenzwertklasse A. In Wohnbereichen können bei Betrieb dieses Gerätes Rundfunkstörungenauftreten, in welchen Fällen der Benutzer für entsprechende Gegenmaßnahmen verantwortlich ist.

Attention!
Ceci est un produit de Classe A. Dans un environment domestique, ce produit risque de créer des interférences radioélectriques, il appartiendra alors à l'utilisateur de prendre les mesures spécifiques appropriées.

Canadian EMC Notice:
"This Class [A] Digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations."

"Cet appareil numerique de la class [A] respecte toutes les exigences du Reglement sur le materiel broilleur du Canada."

The following are trademarks of Digital Equipment Corporation: DECchip, DECnet, DECpc, DIGITAL, OpenVMS, ThinWire, VAX, and the DIGITAL logo.

The following are third-party trademarks:

DALLAS is a registered trademark of Dallas Systems Corporation.
DIGITAL UNIX and UNIX are registered trademarks licensed exclusively by X/Open Company Ltd.
IBM is a registered trademark of International Business Machines Corporation.
Intel is a trademark of Intel Corporation.
NCR is a registered trademark of NCR Corporation.
VIC64 is a trademark of Cypress Semiconductor Corporation.
VxWorks is a registered trademark of Wind River Systems, Inc.

All other trademarks and registered trademarks are the property of their respective holders.

Contents

Part II The Console

4 Console Basics

5 Using the Console

6 Console Command Reference

Part III Diagnostics

7 Diagnostics and System Initialization

8 Console Mode Diagnostics

Part IV Appendixes

A Console Command Summary

B Troubleshooting

C Module Connector Pin Assignments

Figures

Tables

Preface

Purpose of this Manual

This manual introduces you to the DIGITAL Alpha VME 5/352 and 5/480 single-board computers (SBCs) by discussing physical, power, and environmental requirements and describing the module and functional components. This manual also explains how to use the console firmware and discusses diagnostics and troubleshooting.

Intended Audience

This manual is for OEM system integrators who are designing and building a DIGITAL Alpha VME 5/352 or 5/480 SBC into specific application systems. These systems may range in scope from a single Alpha VME 5/352 or 5/480 SBC to highly complex multiprocessor systems that include a variety of hardware. Hardware and mechanical engineers refer to the physical and environmental specifications. Field and manufacturing technicians and support specialists use information in this manual to configure systems and diagnose problems.

This manual assumes that readers have prerequisite knowledge and experience with the following:

- System design
- VMEbus design and specifications

Structure of this Manual

This manual consists of four parts and an index organized as follows:

Part I, Introduction

- Chapter 1, Specifications and Requirements, provides product specifications; physical, power, and environmental requirements; and FCC regulations.
- Chapter 2, Module Components, introduces the physical components of the SBC product.
- Chapter 3, Functional Components, describes the SBC's functional components.

Part II, The Console

- Chapter 4, Console Basics, gets you started with using the console.
- Chapter 5, Using the Console to Operate the SBC, explains how to perform various tasks, using the console.
- Chapter 6, Console Command Reference, describes available console commands.

Part III, Diagnostics

- Chapter 7, Diagnostics and System Initialization, introduces types of diagnostics tests, discusses system initialization, and describes the power-on self-test (POST) diagnostics for nonvolatile RAM and memory.
- Chapter 8, Console Mode Diagnostics, describes diagnostics that you can initiate from the console.

Part IV, Appendixes

- Appendix A, Console Command Summary, serves as a quick reference to available console commands.
- Appendix B, Troubleshooting, provides some guidance with troubleshooting a Alpha VME 5/352 or 5/480 SBC system.
- Appendix C, Module Connector Pin Assignments, describes the pin assignments for the various module connectors.

Conventions

This section defines terminology, abbreviations, and other conventions used in this manual.

Abbreviations

- Register access

  The following list describes the register bit and field abbreviations:

Bit/Field Abbreviation	Description
MBZ (must be zero)	Bits and fields specified as MBZ must be zero.
RO (read only)	Bits and fields specified as RO can be read but not written.
RW (read/write)	Bits and fields specified as RW can be read and written.
SBZ (should be zero)	Bits and fields specified as SBZ should be zero.
WO (write only)	Bits and fields specified as WO can be written but not read

- Binary multiples

The abbreviations K, M, and G (kilo, mega, and giga) represent binary multiples and have the following values:

Abbreviation	Binary Multiple
K	2^{10} (1024)
M	2^{20} (1,048,576)
G	2^{30} (1,073,741,824)

For example:

2 KB	= 2 kilobytes	= 2 x 2^{10} bytes
4 MB	= 4 megabytes	= 4 x 2^{20} bytes
8 GB	= 8 gigabytes	= 8 x 2^{30} bytes

Addresses

Unless otherwise noted, addresses and offsets are hexadecimal values.

Bit Notation

Multiple-bit fields can include contiguous and noncontiguous bits contained in angle brackets (< >). Multiple contiguous bits are indicated by a pair of numbers separated by a colon (:). For example, <9:7,5,2:0> specifies bits 9, 8, 7, 5, 2, 1, and 0. Similarly, single bits are frequently indicated with angle brackets. For example, <27> specifies bit 27.

Caution

Cautions indicate potential damage to equipment or loss of data.

Data Field Size

The term INT*nn*, where *nn* is one of 2, 4, 8, 16, 32, or 64, refers to a data field of *nn* contiguous NATURALLY ALIGNED bytes. For example, INT4 refers to a NATURALLY ALIGNED longword.

Data Units

The following data unit terminology is used throughout this manual.

Term	Words	Bytes	Bits	Other
Byte	1/2	1	8	–
Word	1	2	16	–
Longword	2	4	32	Longword
Quadword	4	8	64	2 Longwords
Octaword	8	16	128	2 Quadwords
Hexword	16	32	256	2 Octawords

Keyboard Keys

The following keyboard key conventions are used throughout this manual.

Convention	Example
Control key sequences are represented as Ctrl/*X*. Press Ctrl while you simultaneously press the *X* key	Ctrl/C
In plain text, key names match the name on the actual key.	Return key
In tables, key names match the name of the actual key and appear in square brackets ([ ]).	[Return]

Examples

Prompts, input, and output in examples are shown in a monospaced font. Interactive input is differentiated from prompts and system output with bold type. For example:

```
>>> echo This is a test.[Return]
This is a test.
```

Ellipsis points indicate that a portion of an example is omitted.

Names and Symbols

The following table lists typographical conventions used for names of various items throughout this manual.

Items	Example
Bits	**sysBus<32:2>**
Commands	**boot** command
Command arguments	*address* argument
Command options	**-sb** option
Environment variables	AUTO_ACTION
Environment variable values	HALT
Files and pathnames	`/usr/foo/bar`
Pins	LIRQ pin
Register symbols	VIP_ICR register
Signals	**iogrant** signal
Variables	*n*, *x*, *mydev*

Note

Notes emphasize particularly important information.

Numbering

Numbers are decimal or hexadecimal unless otherwise indicated. The prefix 0x indicates a hexadecimal number. For example, 19 is decimal, but 0x19 and 0x19A are hexadecimal (see also Addresses). Otherwise, the base is indicated by a subscript; for example, 100_2 is a binary number.

Ranges and Extents

Ranges are specified by a pair of numbers separated by two periods (..) and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets (< >) separated by a colon (:) and are inclusive.

Bit fields are often specified as extents. For example, bits <7:3> specifies bits 7, 6, 5, 4, and 3.

Register and Memory Figures

Register figures have bit and field position numbering starting at the right (low-order) and increasing to the left (high-order).

Memory figures have addresses starting at the top and increasing toward the bottom.

Syntax

The following syntax elements are used throughout this manual. Do not type the syntax elements when entering information.

Element	Example	Description
[ ]	[**-file** *filename*]	The enclosed items are optional.
|	- | + | =	Choose one of two or more items. Select one of the items unless the items are optional.
{ }	{- | + | =}	You must specify one (and only one) of the enclosed items.
( )	(a,b,c)	You must specify the enclosed items together.
...	arg...	You can repeat the preceding item one or more times.

UNPREDICTABLE and UNDEFINED

This manual uses the terms UNPREDICTABLE and UNDEFINED. Their meanings are different and must be carefully distinguished.

UNPREDICTABLE results or occurrences do not disrupt the basic operation of the processor. The processor continues to execute instructions in its normal manner. In contrast, UNDEFINED operations can halt the processor or cause it to lose information.

For More Information

For more information, refer to the following:

- Your supplier
- A DIGITAL Field Applications Engineer
- The DIGITAL OEM web site at `http://www.digital.com/oem`.
- The following DIGITAL Alpha VME 5/352 and 5/480 SBC documentation, which is available on the DIGITAL OEM web site:

Document	Order Number	Description
DIGITAL Alpha VME 5/352 and 5/480 Board Computer Family Data Sheet		Describes the DIGITAL Alpha 5/352 and 5/480 SBCs, highlighting product features and specifications.
DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers Cover Letter	EK–VME54–CL	Highlights important product information and explains how to acquire the *DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers User Manual* and *DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers Technical Reference*.
DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers Warranty and Parts Information	EK–VME54–WI	Explains the warranty of your DIGITAL Alpha VME 5/352 or 5/480 SBC and provides parts information for ordering.
DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers Installation Guide	EK–VME54–UM	Explains how to install your DIGITAL Alpha VME 5/352 or 5/480 SBC. Use this guide if you need to adjust jumper settings or remove and reinstall field replaceable units (FRUs).
DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers User Manual	EK–VME54–UM	Introduces the product by discussing product specifications and requirements and describing the module and functional components. This manual also explains how to use the console firmware and discusses diagnostics and troubleshooting.
DIGITAL Alpha VME 5/352 and 5/480 Single Board Computers Techincal Reference	EK–VME54–TM	This manual discusses system address mapping, the VME interface, system registers, and system interrupts.

- The following DIGITAL documentation:

Document	Order Number
Alpha AXP Architecture Reference Manual	EY–T132E–DP
Alpha Architecture Handbook	EC–QD2KB–TE
Alpha Microprocessors SROM Mini-Debugger User's Guide	EC–QHUXB–TE
Answers to Common Questions about PALcode for Alpha AXP Systems	EC–N0647–72
Digital Semiconductor Alpha 21164 Microprocessor Product Brief	EC–QP97C–TE

Document	Order Number
Digital Semiconductor 21052 PCI–PCI Bridge Data Sheet	EC–QHURB–TE
Digital Semiconductor 21164 Alpha Microprocessor Data Sheet	EC–QP98B–TE
Digital Semiconductor 21172 Core Logic Chipset Product Brief	EC–QUQHA–TE
Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual	EC–QP99B–TE
Digital Semiconductor 21172 Core Logic Chipset Technical Reference Manual	EC–QUQJA–TE
DIGITAL UNIX Guide to Real-time Programming	AA–PS33D–TE
DIGITAL UNIX: Writing PCI Bus Device Drivers	AA–Q7RQC–TE
DIGITAL UNIX: Writing VMEbus Device Drivers	AA–Q057G–TE
Manpages on the VxWorks Real-Time Tools for Alpha CD-ROM	Not applicable
PALcode for Alpha Microprocessors System Design Guide	EC–QFGLC–TE

- The following specifications, which are available through the indicated vendor or organization:

Document	Vendor or Organization
CY7C9640 Specification	Cypress Semiconductor Corp.
Intel 82378ZB PCI-ISA Bridge Chip Specification	Intel Corp.
PCI Local Bus Specification Rev 2.1	PCI Special Interest Group
Super I/O FDC37C6656T Specification	Standard Microsystems Corp.
Symbios 53C810 SCSI Controller Specification	Symbios
TOY clock DS1386 Specification	Dallas Semiconductor
VIC64 Specification	Cypress Semiconductor Corp.

Part I
Introduction

Part I introduces the DIGITAL Alpha VME 5/352 and 5/480 single-board computers (SBCs). This part consists of the following chapters:

- Chapter 1, Specifications and Requirements
- Chapter 2, Module Components
- Chapter 3, Functional Components

1
Specifications and Requirements

This chapter discusses specifications and requirements for the DIGITAL Alpha VME 5/352 and 5/480 single-board computers (SBCs). Specifically, Sections 1.1 through 1.4 discuss:

- Product specifications, Section 1.1
- Physical requirements, Section 1.2
- Power requirements, Section 1.3
- Environmental specifications and requirements, Section 1.4

Section 1.5 discusses the product's regulatory compliance.

1.1 Product Specifications

Based on the 21164 Alpha microprocessor, the DIGITAL Alpha VME 5/352 and 5/480 SBCs run at 352 MHz and 480 MHz, respectively. Unofficially, the 5/480 model achieves SPECint95 at 13.8 and SPECfp95 at 15.5 (peak geometric means), while model 5/352 achieves SPECint95 at 10.7 and SPECfp95 at 13.7 (peak geometric means).

Other distinguishing features include improved cache and memory configurations. The 2 MB of on-board ECC protected Level 3 backup cache (Bcache) operates at 700 MB/s. You can populate four memory connectors with 16 to 512 MB of ECC protected dynamic random access memory (DRAM). The memory is autoconfigured for a 128- or 256-bit data bus. A 256-bit wide bus operates at 355 MB/s and a 128-bit wide bus operates at 210 MB/s.

Table 1–1 lists the Alpha VME 5/352 and 5/480 SBC specifications:

Table 1–1 Alpha VME5/352 and 5/480 SBC Specifications

Alpha processor	
Alpha microprocessor	21164A
CPU speed	5/352 – 352 MHz 5/480 – 480 MHz
Chip cache	Level 1 8/8 KB, Level 2 96 KB unified, I/D
Performance (unofficial)	5/352 – SPECint95: 10.7, SPECfp95: 13.7 5/480 – SPECint95: 13.8, SPECfp95: 15.5
Memory	
Cache	2 MB of on-board Level 3 cache

Table 1–1 Alpha VME5/352 and 5/480 SBC Specifications (Continued)

DRAM	16 to 512 MB ECC protected Autoconfiguration on 128- or 256-bit data bus Single bit error correction Double bit error detection Must be configured in pairs of EDO DIMM memory modules
Flash EPROM	4 MB (3.5 MB available to the user application)
Nonvolatile RAM (NVRAM)	32 KB
Networking	
Features	Alpha 21040 PCI Ethernet controller DMA (bus master) 256-byte send and receive FIFO queues Double bandwidth with full duplex Ethernet
Interconnect	10BASE–T Ethernet (twisted pair)
Interfaces	
SCSI interface	Symbios 53C810 single-ended, 8-bit SCSI–2 with DMA Up to 10 MB/s transfer rate SCSI connection through VMEbus P2 connector
Serial interface	82C42PE and FDC37C665GT Super I/O chip Two asynchronous DEC423 ports 75 to 19200 baud through two MMJ front panel connectors Keyboard, mouse, and parallel ports
PCI I/O companion card	Accepts two PCI mezzanine cards IEEE P1386.1 compliant
Clocks and timers	
Real-time clock	DS1386 RTC with Lithium (<0.5 grams) battery backup
Timers	Three 16-bit timers Two timers are driven at 10 MHz One timer is clocked by external input (through the P2 connector) for event counting or synchronization
Watchdog timer	Programmable timeout Output reset is available on the P2 connector
VME specifications	
VMEbus interface	VIC64 interface chip Conforms to ANSI/IEEE standard 1014–1987 Supports extensions for 64-bit data transfers IEC 821 and 297
VMEbus transactions	Master: A32/24/18, D64/32/16/8 Slave: A32/24/16, D64/32/16/8 UAT, BLT, MBLT
VMEbus arbitration	System controller with configurable arbitration PRI, RRS, SGL, FAIR

Table 1–1 Alpha VME5/352 and 5/480 SBC Specifications (Continued)

VMEbus interrupts	Handles all seven levels 8-bit software programmable status Requester for all seven levels Software-programmable vector
VMEbus connector	DIN 41612 style C 96 contacts P1/P2 connector
Other VMEbus features	SYSCLK and SYSRESET
Physical characteristics	
Single-board computer	Dual-height Eurocard format (6U8HP) 233 x 160 x 40.3 mm (9.17 x 6.3 x 1.59 in.)
Weight	1.014 kg (2.21 lbs.), including four DIMMs
Number of slots	2 (3 with the optional PMC I/O companion card)
PCI mezzanine card	150 x 75 mm (5.9 x 2.95 in.)
Breakout module	Dual-slot version
Power specifications	
Configuration	CPU with 512 MB and no PMC option
5 Vdc	352 MHz – 9 A idle, 13 A peak 480 MHz – 11 A idle, 15 A peak
12 Vdc	0.2 A
-12 Vdc	<01. A
Dissipation (typical)	5/352 – 50 W 5/480 – 60 W
Environmental specifications	
Operating temperature	0° C to 50° C with forced air cooling
Storage temperature	-40° C to 66° C
Temperature change	20° C/hour
Relative humidity	10% to 95% (noncondensing)
Operating systems	
DIGITAL UNIX	Version 4.0A or higher
VxWorks for Alpha	Version 5.2C or higher

1.2 Physical Requirements

DIGITAL Alpha VME 5/352 and 5/480 SBCs have the industry-standard 6U VME form factor and requires two adjacent backplane slots in your VME chassis. A third slot is required if you use the optional PMC I/O companion card.

Once you identify the slots, you must make sure sufficient space exists on the back of the selected slots to install a primary breakout module. This module requires a minimum of 38 mm (1.5 in). For a description of the primary breakout module, see Section 2.6.

If you choose to use the secondary breakout module, you need an incremental clearance of at least 56 mm (2.25 inches) to install the module. For a description of the secondary breakout module, see Section 2.7.

1.3 Power Requirements

The Alpha VME 5/352 and 5/480 SBCs require power voltages of +5 V and ±12 V. The VME backplane provides the power to the logic of the SBCs through the P1 and P2 VMEbus connectors.The primary power for the SBCs is 5 V, which is provided by the P1 and P2 VMEbus connectors on the CPU module and the P2 VMEbus connector on the I/O module. A required primary breakout module augments the current capacity of the backplane's etch and connectors by shunting power from the I/O module connectors to the CPU module connectors.

The two DC-to-DC converters — 5 V to 2.5 V and 5 V to 3.3 V — provide power for CPU module and I/O module operation. The 5 V to 2.5 V converter provides power for the Alpha 21164 core logic. The 5 V to 3.3 V converter provides power for the 21172 core logic chipset, SRAM, DRAMs, SCSI chip, and Ethernet controller. Both converters operate in an 85% to 95% conversion efficiency range, requiring no heat sink.

The required primary breakout module, which is installed on the rear of the VME backplane directly behind the slots occupied by the CPU and I/O module assembly, provides additional current to the CPU module from the I/O module.

An optional +5 V STANDBY is available to provide power for the time-of-year (TOY) clock and NVRAM chip.

Table 1–2 provides the power ratings for the various voltage supplies supported by the Alpha VME 5/352 and 5/480 SBCs.

Table 1–2 Input Power Requirements

Voltage Supply	Tolerance	Maximum Ripple	5/352 Idle Current	5/352 Peak Current	5/480 Idle Current	5/480 Peak Current
+5 V	+0.25 V –0.125 V	50 mV	9 A	13 A	11 A	15 A
+12 V	+0.60 V –0.36 V	50 mV	150 mA	250 mA	150 mA	250 mA
+5 V STDBY	+0.25 V –0.125V	50 mV	25 mA	50 mA	25 mA	50 mA
-12 V	+ 0.36 V –0.60V	50 mV	150 mA	250 mA	150 mA	250 mA

The peak current shown in Table 1–2 assumes an Alpha VME 5/480 SBC is populated with 512 MB of DRAM.

Warning

The +5 V tolerance and ripple specifications shown in Table 1–2 must be met when supplying the peak current specified. If they are not met, undefined operation will result.

1.4 Environmental Specifications and Requirements

DIGITAL Alpha VME 5/352 and 5/480 SBCs require a VME chassis with sufficient cooling. Section 1.4.1 lists the environmental specifications for the SBCs. Section 1.4.2 explains cooling requirements.

1.4.1 Environmental Specifications

Table 1–3 shows the environmental specifications for the Alpha VME 5/352 and 5/480 SBCs.

Table 1–3 Environmental Specifications

Condition	Range or Value
Operating	
Temperature range	0° C (32° F) to 50° C (122° F)
Relative humidity	5% to 90% (noncondensing)
Altitude	6,000 feet (maximum)
Maximum wet bulb	28° C (82° F)
Minimum dew point	2° C (36° F)
Vibration	5 to 500 Hz, 0.1 g, 3 axis
Shock	11 ms, 10 g, 3 axis
Meantime between failures[1] – 5/480	250,000 hours at 25° C
Meantime between failures[1] – 5/352	300,000 hours at 25° C
Nonoperating	
Temperature range	-40° C (-40° F) to 65° C (149° F)
Storage (shipping)	40,000 feet
Relative humidity	5% to 95% (noncondensing)
Packaging weight	0.89 kg (1.96 lb)
Maximum wet bulb	32° C (90° F)
Vibration	1.5 g, 3 axis

[1]MTBF (MIL–HDBK–217F)

Notes

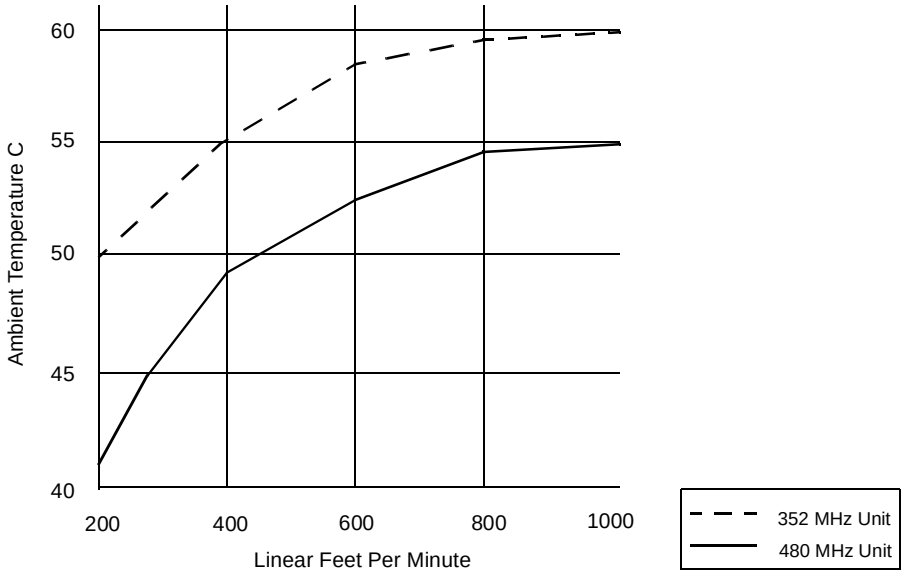
Real failures for MBTF figures are defined as random component failures that are not caused by customer errors, workmanship related failures, third-party component issues, or design related problems where corrective action has been implemented.

The operating temperature range is 0° C to 50° C. This is dependent on processor speed and enclosure air flow (see Figure 1–1).

1.4.2 Cooling Requirements

The Alpha VME 5/352 and 5/480 SBCs provide a heat sink for CPU thermal control. The amount of cooling required is defined by the operating environment to which the SBC assembly is subjected. The curve shown in Figure 1–1 defines the amount of ambient air the SBC assembly requires in linear feet per minute at various ambient temperatures. Actual cooling depends on the turbulence in the air stream as it enters the assembly volume.

Figure 1–1 Required Air Flow Relative to Ambient Temperature



Note

The maximum temperature, when measured between the heat sink studs on the base of the heat sink, must be less than 68° C.

1.5 Regulatory Compliance

The DIGITAL Alpha VME 5/352 and 5/480 SBCs have been tested and shown to operate within a suitable enclosure with the following regulatory compliances:

- EMC, CE, and VCCI limits for a Class A device
- UL, CSA, and TUV safety limits

These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used as instructed in the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Installation Guide*, may cause harmful interference to radio communications. Operation of an Alpha VME 5/352 or 5/480 SBC in a residential area is likely to cause harmful interference, in which case the interference is required to be corrected at the user's own risk.

When used in an appropriate enclosure, an Alpha VME 5/352 or 5/480 SBC can operate at the level of a Class B device. If used as a Class B device, your application may require shielded cables for all I/O interfaces.

Note

It is incumbent upon Original Equipment Manufacturers (OEMs) to obtain regulatory FCC approval for a consolidated system.

2
Module Components

The DIGITAL Alpha VME 5/352 and 5/480 SBCs consist of a single CPU module and support modules that provide I/O, memory, and power. This chapter describes the module components. The chapter begins with an overview (Section 2.1) and then describes the following:

- CPU module, Section 2.2
- I/O module, Section 2.3
- CPU and I/O assembly controls and indicators, Section 2.4
- Memory modules, Section 2.5
- Primary breakout module, Section
- Secondary breakout module, Section 2.7
- PMC I/O companion card, Section 2.8

2.1 Module Component Overview

Alpha VME 5/352 and 5/480 SBCs can consist of two or three 6U modules depending on whether you use an optional PMC I/O companion card. The base SBC assembly includes a CPU module and an I/O module. The CPU module features either a 352 MHz or 480 MHz 21164 Alpha microprocessor and a supporting 21172 core logic chip set. Four DIMM sockets for DRAM and 2 MB of Level 3 SRAM Bcache also reside on the CPU module. Two DC-to-DC power converters — 5 V to 2.5 V and 5 V to3.3 V — provide power for the CPU module's operation. The CPU module is shipped preassembled with a required I/O module. The I/O module connects to the CPU module through a PCI–32 interface.

The I/O module provides support for your application's I/O devices. Key components of this module include:

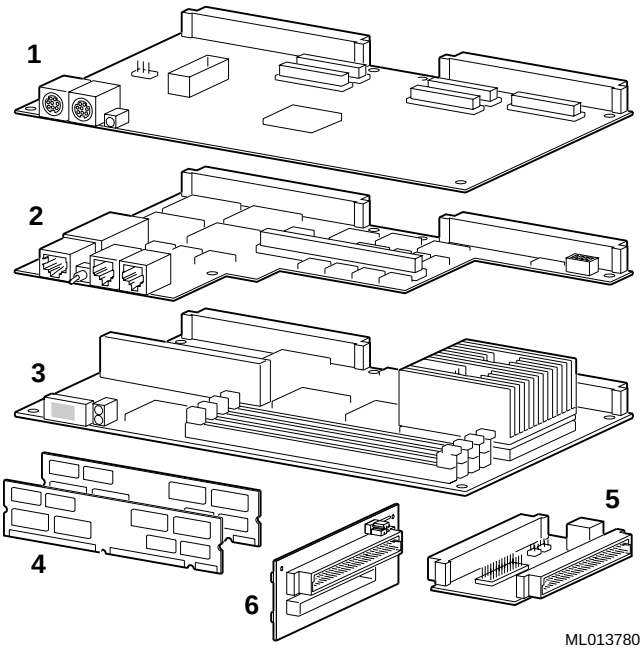
- PCI-to-VME64 bridge (DC7407 VIP and VIC64)
- PCI-to-SCSI–2 controller (53C810)
- PCI-to-Ethernet controller (21040)
- PCI-to Nbus bridge (82378ZB)
- PCI–32 interface to an optional PMC I/O companion card

The Nbus supports a diskette drive, two serial-line ports, a parallel port, a keyboard and mouse, the Flash ROM, the TOY clock, and NVRAM.

The optional PMC I/O companion card provides a PCI-to-PCI bridge, two PMC option slots, and keyboard, mouse, and diskette drive connectors.

Figure 2–1 identifies the module components of an Alpha VME 5/352 or 5/480 SBC and optional PMC I/O companion card.

Figure 2–1 Alpha VME 5/352 and 5/480 Module Components



The numeric callouts in the figure identify the following key components:

1 PMC I/O companion card option

2 I/O module

3 CPU module

4 Memory modules

5 Secondary breakout module

6 Primary breakout module

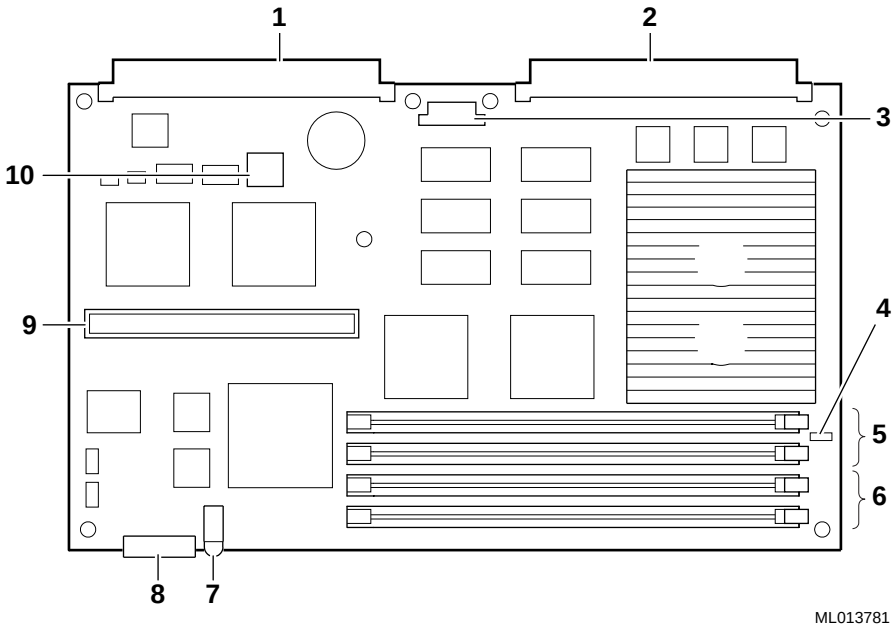
Note

The I/O module (2) and CPU module (3) are attached and share a common front panel. These modules should be detached only to replace the SROM. They appear separately in Figure 2–1 only to provide a view of primary SBC module components.

2.2 CPU Module

The CPU module is the compute engine of Alpha VME 5/352 and 5/480 SBCs. Figure 2–2 shows the layout and primary components.

Figure 2–2 CPU Module Layout



The numeric callouts in the figure identify the following key components:

1 P1 VMEbus connector

2 P2 VMEbus connector

3 64-bit PCI connector (not used)

4 J11 bus grant pass-through jumper

5 Connectors for memory DIMMs 2 and 3

6 Connectors for memory DIMMs 0 and 1

7 Power and VME slave activity/watchdog timeout LED

8 Status display

9 I/O module connector

10 SROM

2.3 IO Module

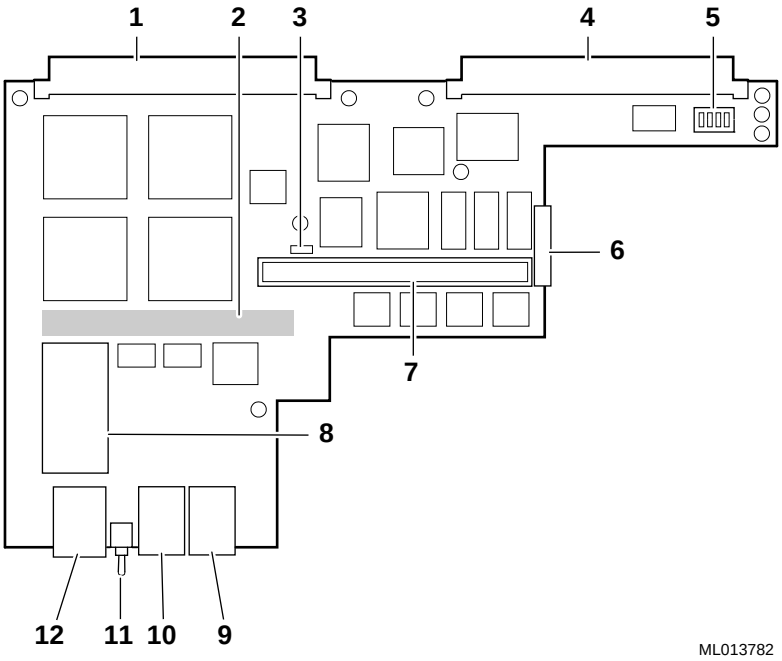
The I/O module is a required second tier module that handles all I/O activity for the Alpha VME 5/352 and 5/480 SBCs. This module plugs into the I/O module connector on the CPU module.

Note

The I/O module is attached to the CPU module when you receive it. Disassemble the CPU and I/O assembly only if you need to replace the SROM.

Figure 2–3 shows the layout and primary components.

Figure 2–3 I/O Module Layout



The numeric callouts in the figure identify the following key components:

1 P1 VMEbus connector
2 Connector to CPU module (on the back of the I/O module)
3 Debug jumper (for use with Serial ROM Mini-Console only)
4 P2 VMEbus connector
5 Configuration switchpack
6 Caterpillar insulation strip
7 PMC I/O companion card connector
8 Nonvolatile RAM/time-of-year (TOY) clock
9 Auxiliary serial port
10 Console serial port
11 Reset/Halt switch
12 Twisted-pair Ethernet connector

2.4 CPU and I/O Assembly Controls and Indicators

The CPU and I/O modules are delivered as a single assembly. The modules are attached and share a single front panel. Figure 2–4 shows the controls and indicators on that front panel and Table 2–1 describes their functions.

Figure 2–4 Controls and Indicators

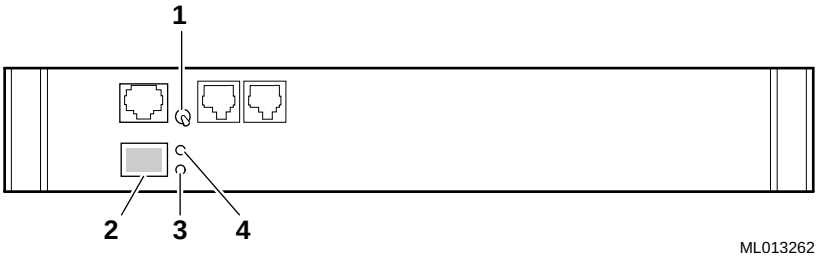


Table 2–1 Controls and Indicators

Callout	Control or Indicator	Description
1	Reset/Halt switch	A switch that resets the SBC when pressed in the Reset (up) direction and halts the operating system when pressed in the Halt (down) direction. A reset operation starts SROM execution the same way as when you power on the system. When you use the Halt switch, the SBC enters console mode. **Caution:** Keep in mind that reset and halt operations can cause loss of data.[1]
2	Status display	A display that shows which test is running during power-on self-test (POST) diagnostics. When the POST diagnostics are complete, the display is under control of the operating system or an application program.
3	VME Slave Activity/Watchdog Timeout LED	An amber LED with two functions. The LED flashes when the SBC is accessed as a slave by another device on the VMEbus. The LED lights continuously when the watchdog timer has timed out. **Note:** The LED can appear to light continuously when the module is receiving slave accesses. Since the LED glows for 1/3 of a second each time it flashes, three slave accesses per second could make the LED light continuously.
4	Power LED	A green LED that is lit when the power is on.

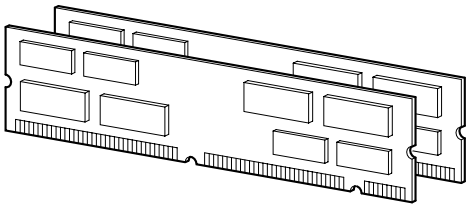
[1]See your operating system documentation for information on how to recover from reset and halt operations.

2.5 Memory Modules

The Alpha VME 5/352 and 5/480 SBCs support memory configurations that range from 16 to 512 MB of dynamic random access memory (DRAM). This memory is accessible from the CPU, PCI bus, and VMEbus.

You can plug either two or four dual integrated memory modules (DIMMs), ranging from 8 MB to 128 MB, into the memory connectors on the CPU module. Figure 2–5 shows a typical memory module.

Figure 2–5 Memory Module



The number of DIMMs you use determines the memory bus bandwidth, and consequently the overall speed of data write and read operations to and from memory. DIGITAL recommends that you use four DIMMs to achieve maximum performance. No jumper changes are required. The system automatically configures memory based on the DIMMs you install. The following table shows the width of the memory bus and its performance when you use two and four DIMMs:

Number of DIMMs	Bus Width	Memory Bandwidth
2	128 bits	210 MB/s
4	256 bits	355 MB/s

Error correction code (ECC) is provided for single-bit errors and error detection is provided for double-bit errors. For details on how the operating system reports and handles ECC errors, see your operating system documentation.

In addition to the requirement of using either two or four DIMMs, all DIMMs you use must be identical in size (number of MB), speed, and architecture (EDO).

Note

DIGITAL memory DIMMs are supplied in pairs. DIGITAL may source the pairs of DIMMs from different memory vendors. To ensure proper operation of your SBC, you must install the DIMMs as supplied pairs in memory connectors 0 and 1 or 2 and 3. If you choose to use only two DIMMs, you must populate memory connectors 0 and 1.

Table 2–2 shows valid DIMM combinations.

Table 2–2 Valid DIMM Combinations

Memory Size (MB)	DIMM 0 (MB)	DIMM 1 (MB)	DIMM 2 (MB)	DIMM 3 (MB)
16	8	8		
32	8	8	8	8
32	16	16		
64	16	16	16	16
64	32	32		
128	32	32	32	32

Table 2–2 Valid DIMM Combinations (Continued)

Memory Size (MB)	DIMM 0 (MB)	DIMM 1 (MB)	DIMM 2 (MB)	DIMM 3 (MB)
128	64	64		
256	64	64	64	64
256	128	128		
512	128	128	128	128

For information on memory installation, see the *DIGITAL Alpha VME 5/352 and VME 5/480 Single-Board Computers Installation Guide*.

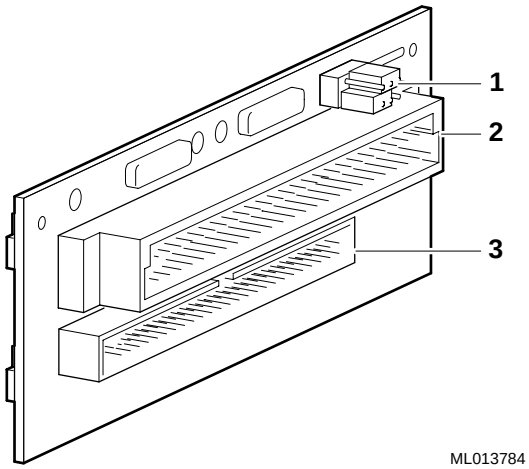
2.6 Primary Breakout Module

The primary breakout module is a required module that plugs into your VMEbus backplane behind the slots occupied by your Alpha VME 5/352 or 5/480 SBC CPU and I/O modules. This breakout module supplies additional power to the CPU module by way of the VMEbus P2 connector and provides:

- A connector for attaching a SCSI bus
- Additional P2 options, such as the secondary breakout module
- SCSI termination control
- A connection for and control of a watchdog timeout signal
- A connector to Alpha VME external timing signals

Figure 2–6 shows the primary breakout module.

Figure 2–6 Primary Breakout Module



The numeric callouts in the figure identify the following key components:

1 SCSI termination and watchdog reset signal jumpers

2 Connector for the secondary breakout module or an external monitoring device

3 SCSI cable connector

A reset input signal on pin C10 of the primary breakout module's VMEbus P2 connector is available for resetting the SBC. This signal is low during normal operation and high during a watchdog timer reset in parallel with the Reset switch on the SBC's front panel. Because pin C10 is a nonbuffered input pin, you should use shielded wiring to apply the reset input signal.

Caution

You must use the primary breakout module included in your Alpha VME 5/352 or 5/480 SBC hardware kit. Applying power to a DIGITAL Alpha VME 5/352 or 5/480 SBC **WITHOUT** that primary breakout module in place, or **WITH** the breakout module included with the AXPvme 160, 166, or 230 (part number 54-22605-01) in place, may damage your backplane, the Alpha VME 5/352 or 5/480 SBC, or both.

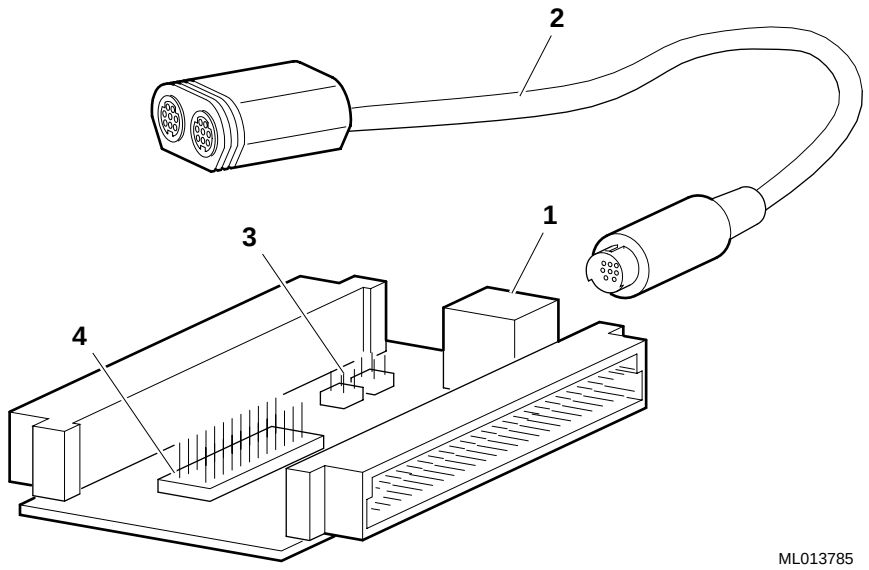
For information on primary breakout module jumper settings, see the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Installation Guide*.

2.7 Secondary Breakout Module

The secondary breakout module is an optional module that connects to the primary breakout module. Connectors on the secondary breakout module include a PS/2 keyboard and mouse Y-cable connector and a parallel port connector. The primary use of this module is to add a serial-line (keyboard and mouse) connector and parallel port to the rear of the VME chassis.

Figure 2–7 shows the secondary breakout module.

Figure 2–7 Secondary Breakout Module



The numeric callouts in the figure identify the following key components:

1 PS/2 keyboard and mouse connector

2 PS/2 keyboard and mouse Y-cable (supplied in PMC I/O companion card kits, EBV1P)

3 Keyboard and mouse jumper

4 Parallel port

Note

The Alpha VME 5/352 and 5/480 SBCs support a PS/2-type 101-compatible keyboard and mouse.

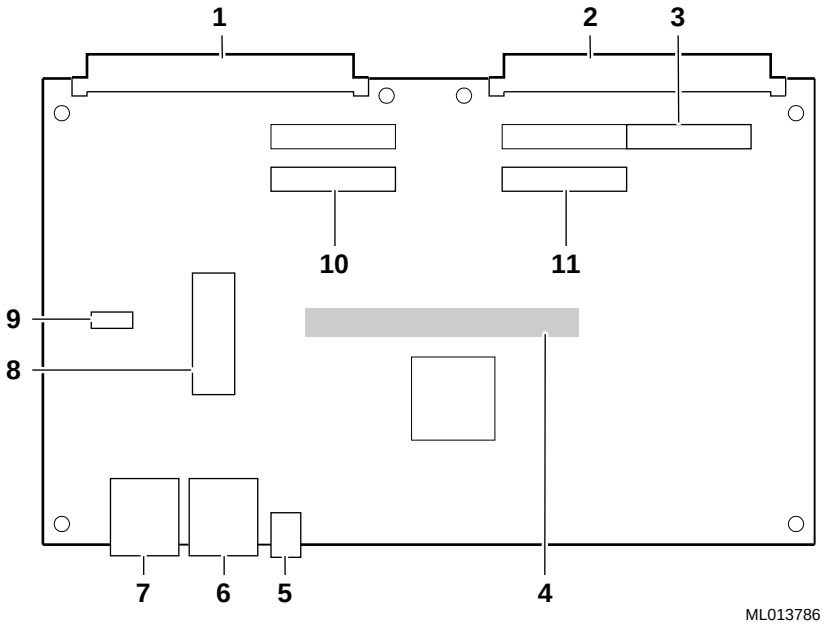
2.8 PMC I/O Companion Card

The PMC I/O companion card is an optional third tier module that plugs into a connector on the I/O module. Using the PMC I/O companion card, you can expand your SBC's I/O capabilities by adding interfaces, such as a second Ethernet interface or a graphics card. Primary components on the companion card include connectors for two PMC options, a PCI-to-PCI bridge chip, keyboard and mouse connectors, two VMEbus connectors, and a VMEbus P2 signal connector. The VMEbus P2 signal connector provides a way of sending I/O signals from a PMC option to a device attached to the VMEbus P2 connector instead of to the front panel of the PMC option card.

To use the PMC I/O companion card, you must have three adjacent slots available in your VME chassis.

Figure 2–8 shows the layout of the card.

Figure 2–8 PMC I/O Companion Card Layout



The numeric callouts in the figure identify the following key components:

1 P1 VMEbus connector

2 P2 VMEbus connector

3 VMEbus P2 signal connector for PMC option 1

4 I/O module connector (on the back of the PMC I/O companion card)

5 Power LED

6 Keyboard connector

7 Mouse connector

8 Diskette drive connector

9 Signaling level jumper (jumper MUST be set to 5.0 V)

10 PMC option 2 connector

11 PMC option 1 connector

Note

The Alpha VME 5/352 and 5/480 SBCs support a PS/2-type 101-compatible keyboard and mouse.

The 34-pin diskette drive connector (see item 8 in Figure 2–8) provides a way of attaching a diskette drive (for example, an RX23 or RX26). To use this connector, you must make or buy a cable that is best suited for your application. DIGITAL supplies only the pin assignments for the connector.

For a description of the connector pin assignments, see Appendix C.

3
Functional Components

This chapter describes the functional components associated with the DIGITAL Alpha VME 5/352 and 5/480 SBCs. The chapter begins with an overview (Section 3.1) and then describes the following:

- 21164 Alpha microprocessor chip, Section 3.2
- 21172 core logic chipset, Section 3.3
- Bcache subsystem, Section 3.4
- Memory subsystem, Section 3.5
- SROM, Section 3.6
- Clock interface, Section 3.7
- PCI interface, Section 3.8
- Nbus interface, Section 3.9
- VME interface, Section 3.10

For information on the address mapping, registers, and system interrupts associated with these components, see the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Technical Reference*.

3.1 Functional Component Overview

Figure 3–1 identifies the functional components of the Alpha VME 5/352 and 5/480 SBCs. The Alpha VME 5/352 and 5/480 CPU modules are based on the 21164 Alpha microprocessor, and run at 352 MHz and 480 MHz, respectively. The 21172 core logic chip set consists of the 21172–CA control, I/O interface, and address (CIA) chip and four 21172–BA data switch (DSW) chips. Nine SRAMs provide 2 MB of Bcache and two or four main memory DIMMs provide from 16 to 512 MB of EDO memory. The system clock uses a phase lock loop (PLL)/buffer circuit to provide SYSCLK signals to 10 system components at 32 MHz.

The CPU module interfaces with the I/O module through a 32-bit PCI bus. As Figure 3–1 shows, the I/O module provides a:

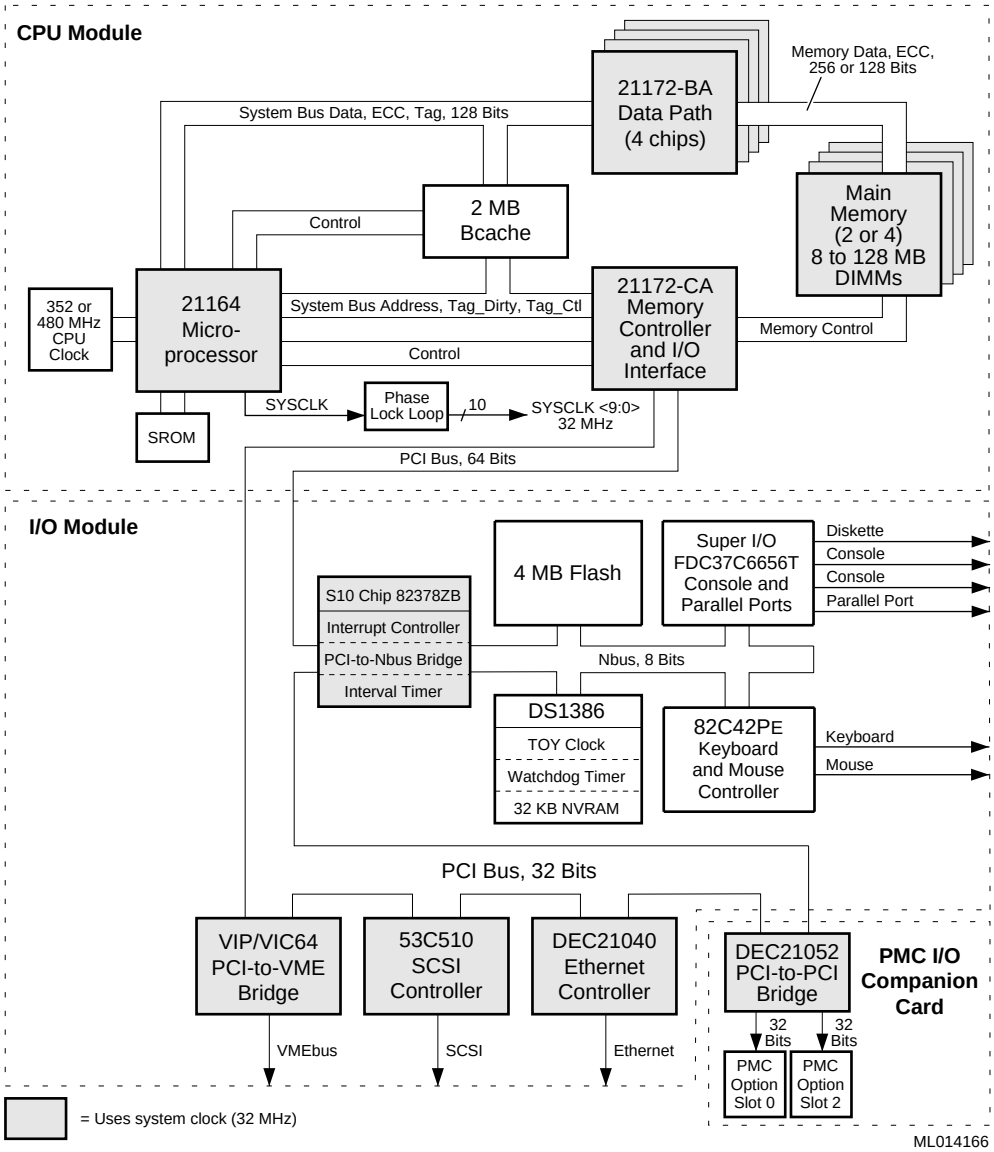
- PCI-to-VME64 bridge (DC7407 VIP and VIC64), which provides an interface to the VMEbus
- PCI-to-SCSI controller (53C810), which provides an interface to SCSI devices
- PCI-to-Ethernet controller (21040), which provides a network interface
- PCI-to-Nbus bridge (82378ZB), which provides access to the system's 8-bit Nbus and includes interrupt controller and interval timer support
- PCI–32 interface to an optional PMC I/O companion card

The I/O module's Nbus is a resource bus that is based on the ISA bus. The Nbus handles the read and write cycles for the following:

- 4M of flash ROM
- Super I/O chip (FDC37C6656T) resources, which include console and parallel ports and a diskette drive
- TOY clock, watchdog timer, and NVRAM chip (DS1386) resources
- Keyboard and mouse controller (82C42PE)

The I/O module interfaces to an optional PMC I/O companion card through the 32-bit PCI bus. The PMC I/O companion card uses a DEC 21052 PCI-to-PCI bridge to provide access to two PMC option slots. This optional card also provides keyboard, mouse, and diskette drive connectors.

Figure 3–1 Alpha VME 5/352 and 5/480 Functional Components



3.2 21164 Alpha Microprocessor

The Alpha VME 5/352 and 5/480 SBCs are based on the 21164 Alpha microprocessor, which is a superscalar pipelined processor manufactured using 0.35 µm CMOS technology. It is packaged in a 499-pin IPGA carrier.

The 21164 microprocessor can issue four Alpha instructions in a single cycle, thereby minimizing the average cycles per instruction (CPI). A number of low-latency and/or high-throughput features in the instruction issue unit and the onchip components of the memory subsystem further reduce the average CPI.

The 21164 microprocessor and associated PALcode implements IEEE single-precision and double-precision, VAX F_floating and G_floating data types, and supports longword (32-bit) and quadword (64-bit) integers. Byte (8-bit) and word (16-bit) support is provided by byte-manipulation instructions. Limited hardware

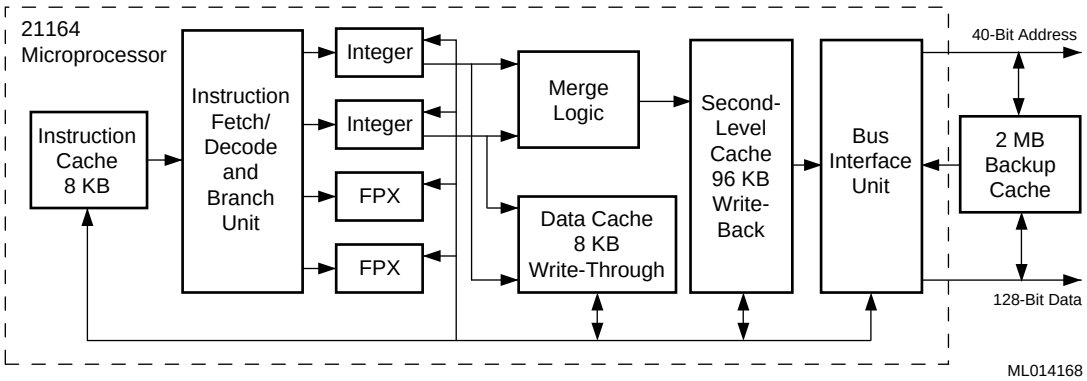
support is provided for the VAX D_floating data type. Partial hardware implementation is provided for the architecturally optional FETCH and FETCH_M instructions.

Other features of the microprocessor include:

- An onchip, demand-paged memory-management unit with a translation buffer
- Two onchip, high-throughput pipelined floating-point units, capable of executing both DIGITAL and IEEE floating-point data types
- An onchip, 8 KB virtual instruction cache (Icache) with 7-bit ASNs (MAX_ASN=127
- An onchip, dual-read-ported, 8 KB data cache (Dcache)
- An onchip, write buffer with six 32-byte entries
- An onchip, 96 KB, 3-way, set-associative, write-back, second level (level 2) mixed instruction and data cache
- A 128-bit data bus with onchip parity and error correction code (ECC) support
- An external third level (level 3) synchronous 2 MB backup cache (Bcache)
- An internal clock generator providing a high-speed clock used by the 21164 microprocessor, and a pair of programmable system clocks for use by the CPU module
- Onchip performance counters to measure and analyze CPU and system performance
- Chip and module level test support, including an Icache test interface to support chip and module level testing
- A 3.3 V external interface and 2.5 V core power for reduced power consumption

Figure 3–2 shows the microprocessor's functional units and caches in a functional block diagram.

Figure 3–2 21164 Alpha Microprocessor Functional Block Diagram



For more detailed information on the microprocessor, see the *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*.

3.3 21172 Core Logic Chipset

The DIGITAL 21172 core logic chipset supports the 21164 Alpha microprocessor in high-performance uniprocessor systems. The chipset includes an interface to the 64-bit peripheral component interconnect (PCI) bus, and associated control and data paths for the 21164 microprocessor chip, memory, and level 3 Bcache.

Sections 3.3.1 and 3.3.2 discuss the chipset components and features. For more detailed information on the 21172 core logic chipset, see the *Digital Semiconductor 21172 Core Logic Chipset Technical Reference Manual*.

3.3.1 Chipset Components

The chipset consists of:

- A control, I/O interface, and address (CIA) chip – 21172-CA chip

  The CIA chip is a 388-pin plastic ball grid array (PBGA) package that provides control functions for main memory, a bridge to the 64-bit PCI bus, and control functions for the DSW chips and part of the I/O data path.

- Four data switch (DSW) chips – 21172-BA chips

  The DSW chips are 208-pin plastic quad flat pack (PQFP) packages that provide bidirectional data paths between the 21164 microprocessor, main memory, Bcache, the CIA chip, and part of the I/O data path. The majority of the DSW logic consists of data buffers and multiplexers. Using two encoded control fields, the CIA chip directs data flow to and from the DSW chips.

3.3.2 Chipset Features

The chipset includes the majority of functions required to develop high performance systems that require minimum discrete logic on the module. Features include:

- Support for the 21164 Alpha microprocessor chip
- A 64-bit, ECC-protected data path (IOD bus) between the CIA and DSW chips
- A 128-bit ECC-protected data path (system bus) between the 21164 and DSW chips
- A 256-bit ECC-protected memory data path (memory bus) between the DSW chips and memory
- A 32 MHz system bus interface
- Support for 2 MB of write-back, ECC-protected, level 3 Bcache using the flush cache coherency protocol
- Support for 16 to 512 MB of EDO memory
- PCI bus support that includes 64-bit multiplexed address and data paths, 64-bit PCI address handling, and scatter-gather mapping

- 32 MHz PCI clock frequency
- DSW chips that provide a victim buffer for read miss/victim transitions

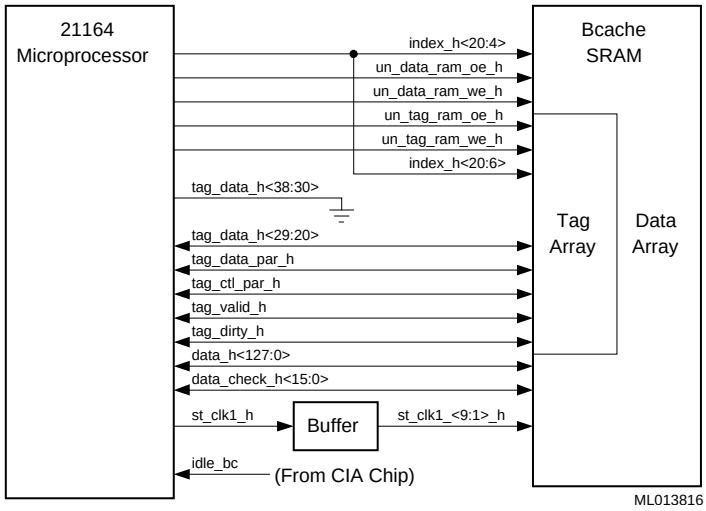
3.4 Bcache Subsystem

The DIGITAL Alpha VME 5/352 and 5/480 SBCs provides 2 MB of direct mapped Bcache. The Bcache is populated with nine 9 nanosecond, 64K-bit X 36-bit synchronous static random access memories (SRAMs). Bcache features include:

- A block size of 64 bytes
- System bus Bcache private read/write transfer rate of 700 MB/s
- ECC protection
- Use of the flush cache coherency protocol as described in the *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*

The 21164 Alpha microprocessor controls the level 3 Bcache array as shown in Figure 3–3.

Figure 3–3 Level 3 Bcache Array



3.5 Memory Subsystem

The Alpha VME 5/352 and 5/480 SBCs support two or four dynamic random access memory (DRAM) DIMMs for up to a total of 512 MB of 60 nanosecond, EDO main memory. The memory resides in a single bank. Table 2–2 lists valid DIMM combinations.

Quadword error checking and correction (ECC) is supported on the memory and system buses. The 21172 core logic chipset controls and routes all CPU-to-memory caching and PCI direct memory access (DMA) operations. The DSW and CIA components of the chipset provide a high-speed memory data path that has a width of either 128 or 256 bits, depending on the mode in which the SBC is oper-

ating. When you use two DIMMs, the SBC operates in 128-bit mode; when you use four DIMMs the SBC operates in 256-bit mode. The memory bus bandwidth in 128-bit mode is 210 MB/s, while the bandwidth in 256-bit mode is 355 MB/s.

The memory subsystem optimizes its cache read miss with victim write cycle to improve memory and system bus bandwidth. The optimizations are achieved by partitioning the the memory row and column addressing such that the read miss row and victim row addresses match.

The cache read miss cycle begins when the 21164 Alpha microprocessor recognizes a cache read miss with victim. When a read miss with victim is identified, the microprocessor instructs the CIA chip to take the victim and then get the read miss data. The CIA chip places the victim data in a DSW buffer while initiating a memory read cycle (RAS–CAS–RAS). The CIA and DSW chips then supply the read data to the microprocessor and cache then write the victim data to memory (CAS–CAS). The resulting memory cycle — CAS – RAS (read 32 bytes) – RAS (read 32 bytes) – RAS (write 32 bytes) – RAS (write 32 bytes) — completes in 360 ns or 355 MB/s.

3.6 SROM

The SROM for the Alpha VME 5/352 and 5/480 SBCs contains 8 KB of code that is loaded into the Alpha 21164 microprocessor's Icache serially when the system powers up or during a reset. Execution is passed to this code in PAL mode. SROM initialization is explained in detail in Chapter 7.

The SROM is socketed to allow future firmware upgrades.

3.7 Clock Interface

The CPU clock circuit used by the Alpha VME 5/352 and 5/480 SBCs multiplies a 16 MHz clock frequency by 22 or 30 and buffers the results, supplying the Alpha 21164 microprocessor with a 352 MHz or 480 MHz clock speed. The microprocessor divides the input value 352 or 480 by 11 or 15, respectively, to generate the system clock.

The 21164 system clock signal (SYSCLK) drives a phase lock loop (PLL)/buffer circuit. That circuit, in turn, generates 10 copies of the 32 MHz SYSCLK signal for the 21172 core logic chip set components and all PCI devices.

The 21172 core logic chipset generates its own 1x and 2x clock signals on each DSW and CIA chip.

3.8 PCI Interface

The PCI interface consists of a PCI bus that serves as the base of the I/O subsystem, connecting all of the system's PCI devices. The I/O subsystem consists of the 21172 core logic CIA and DSW chips and the following PCI devices:

- Ethernet controller
- SCSI controller
- PMC I/O companion card

- Nbus interface
- VME interface

Sections 3.8.1 to 3.8.3 briefly discuss Ethernet, SCSI, and PCI Expansion Card support. For introductions to the Nbus and VME interfaces, see Sections 3.9 and 3.10.

3.8.1 Ethernet Controller

The Ethernet controller for the Alpha VME 5/352 and 5/480 SBCs is based on the DECchip 21040-AA. This chip keeps processor intervention in local area network (LAN) control to a minimum. The chip behaves:

- As a bus slave when communicating with the PCI bus to gain access to configuration and control/status registers
- As a bus master when communicating with memory

The Ethernet controller handles the following types of cycle termination:

- Target-initiated retry
- Abort
- DEVSEL abort

Target-aborted terminations cause an interrupt.

The physical connection to the network is through the Ethernet 10BASE–T twisted-pair connector located on the front panel of the CPU and I/O subassembly.

The Ethernet ID address for the Alpha VME 5/352 or 5/480 SBC assembly is stored in a 20-pin socketed PLCC.

For more information on programming and using the DECchip 21040-AA, see the *DECchip 21040-AA Specification*.

3.8.2 SCSI Controller

The SCSI controller for the Alpha VME 5/352 and 5/480 SBCs is based on the Symbios 53C810 chip. This controller allows you to attach up to seven SCSI devices to your SBC.

The primary breakout module provides an interface to a standard SCSI cable. This module brings the SCSI bus to a standard 50-pin SCSI connector pinning for direct connection to an unshielded SCSI A-cable. A 6-pin jumper block on the module controls SCSI termination as follows:

- Enables SCSI termination when the jumper is set across pins 1 and 3
- Disables SCSI termination when the jumper is set across pins 3 and 5

The controller can affect high-level SCSI operations with very little intervention from the processor. The controller accomplishes this through its low-level register interface or by applying Symbios SCSI scripts.

Once the controller is configured in PCI address space, programming of the Symbios 53C810 chip is compatible with the Symbios 53C720 chip.

For more information on programming the Symbios 53C720 chip, see the chip's programming guide.

3.8.3 PMC I/O Companion Card

The optional PMC I/O companion card provides a 21052 PCI-to-PCI bridge chip and two sets of PMC connectors for adding one double-width or two single-width PMC option modules. One of the PMC connector sets includes a third connector that allows I/O access through the P2 connector.

PCI bus arbitration supports two PMC devices with up to four interrupt request lines. The PCI clock is driven from the CPU and I/O subassembly at a frequency of 32 MHz. The card connectors provide 3V and 5V supply voltages. Although you can have mixed supply voltages between cards, the PCI bus signaling voltage must be configured to 5 V when the card is installed.

3.9 Nbus Interface

The Nbus interface is a simple nonmultiplexed resource bus that is based on the ISA bus and supports 8-bit data transfers and 16-bit addressing. This bus provides an interface to the PCI bus through an Intel System I/O chip (82378ZB). The interface translates PCI I/O references to the Nbus into simple read and write cycles for resources attached to the Nbus lines. Such resources include the system's:

- Interrupt controllers
- Flash ROM
- TOY clock
- Watchdog timer
- NVRAM
- Interval timer
- Keyboard and mouse controller
- Super I/O chip

3.9.1 Interrupt Controllers

Most interrupts on Alpha VME 5/352 and 5/480 SBCs are routed through the following interrupt controllers:

- Xilinx interrupt controller
- VIC64 chip system interrupt controller
- SIO chip (82378ZB) programmable interrupt controller

The Xilinx interrupt controller handles CPU interrupts. This controller consists of four interrupt mask registers that generate CPU interrupt request signals.

The VIC64 chip interrupt controller handles VMEbus interrupts. It controls two external/system interrupt sources: DC7407 status and DC7407 errors. Each of these sources has an associated interrupt control register (ICR) that allows the interrupt to be programmed with an interrupt priority level (IPL) or disabled.

Use of the VIC64 chip in Alpha VME 5/352 and 5/480 SBCs as an interrupt controller is modified slightly by the operation of the DC7407, the SIO chip, and the interrupt/mask registers.

The SIO chip interrupt controller delivers interrupts from the mouse, keyboard, and Super I/O chip (37C665) to the interrupt/mask register.

For more information about the interrupt controllers and the handling of system interrupts, see the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Technical Reference* .

3.9.2 Flash ROM

The Alpha VME 5/352 and 5/480 SBCs have a total of 4 MB of electrically erasable and writable flash ROM. The flash ROM is segmented into 1 MB windows, using bits <1:0> of a module control register. The system console firmware is prewritten into the first 512 KB, providing you with 3.5 MB of additional space to use for your application.

To protect the contents of the flash ROM from unauthorized or accidental updates, you must close DIP Switch 2 on the I/O module before enabling write operations. That switch must always be open unless you are updating the flash ROM. (The state of the switch is stored in Flash Switch bit <3> of the module control register.) Independent of the state of the switch, you can overwrite the setting in the software to enable automatic updates.

3.9.3 TOY Clock

The Dallas Semiconductor DS1386 chip provides the SBC's time-of-year (TOY) clock functionality. This chip also supports the watchdog and SRAM functionality as nonvolatile random access memory (NVRAM).

Note

The Alpha VME 5/352 and 5/480 SBCs do not support the DS1386 chip's alarm features.

The TOY clock maintains the system's time: year, month, date, day, hour, minute, second, 110th of a second, and 1/100th of a second. The clock corrects the date for months with fewer than 31 days and for leap years. In addition, the clock can maintain the time in 24-hour or 12-hour AM/PM format.

The square wave output of the chip generates a fixed 1024 Hz interval and time-keeping accuracy is better than +/- minute/month at 25°C.

The clock maintains time in the absence of Vcc by using an internal lithium (less than 0.5 grams) energy cell that has an active life of at least 10 years. In addition, internally the clock protects against spurious accesses during power transitions.

Some applications may require the TOY clock and NVRAM to operate from an external uninterruptable power supply (UPS). The Alpha VME 5/352 and 5/480 SBCs have an onboard switch (J3 switch 1) to allow a connection to the 5 V standby connection (5VSTDBY) on the VMEbus. When Switch 1 is closed, the VME 5VSTDBY is connected to the TOY supply through isolation diodes.

The chip is socketed to allow:

- Replacement when the internal power source is no longer functional
- Physical removal of the NVRAM

The TOY clock registers are updated every 0.01 seconds. You gain access to the clock to examine or set the current time by using the console **date** command (see Section 5.5).

3.9.4 Watchdog Timer

The watchdog timer resides on the Dallas Semiconductor DS1386 chip. The watchdog timer allows hardware to bring the system back to a known state when a software failure occurs.

An application can initialize the watchdog timer with a value in the range 0.01 to 99.9 seconds. If left unaccessed, the timer decrements towards 0. If the timer reaches 0, the watchdog timer halts the system (jump to Halt entry in firmware) and then forces the module hardware to be reset (some 300 ms later). The application can maintain the module by periodically accessing the watchdog timer registers. When you access these registers, the watchdog timer resets back to the initialization value. Therefore, as long as the worst-case time between watchdog timer access is less than the programmed timeout value, the module functions normally.

The Alpha VME 5/352 and 5/480 SBCs indicate the status of the on-board watchdog timer with the signal WD_STATUS_OC on pin C6. This signal is driven low when an on-board watchdog timer expires. The device that drives the signal is a 74LS05 open-collector inverter. This device is capable of sinking the signal a maximum of 8 mA (IOL). You can pull up the WD_STATUS_OC signal to the 5 V rail by using a 2 KΩ resistor and setting the primary breakout module jumper across pins 4 and 6 (default). To disconnect the resistor from the 5 V rail, set the jumper across pins 2 and 4.

In addition to the hardware support for watchdog timer operation, you can configure the firmware to dispatch to user code or continue with its default reset action on watchdog timeout. The firmware can detect the expiration of the watchdog timer during a reset operation by examining the hardware reset reason register. The jump to the Halt code just before the reset enables the firmware to record a snapshot of the processor's state before the hardware reset is complete.

3.9.5 NVRAM

Within the TOY clock, the Alpha VME 5/352 and 5/480 SBCs offer just under 32 KB of on-board SRAM that is backed up by battery. The RAM is provided by the Dallas Semiconductor DS1386 chip and is held nonvolatile by a built-in lithium battery source.

The nonvolatile RAM (NVRAM) is accessible for read and write operations in Nbus space. The DS1386 chip contains 32 KB read/write byte elements. The lowest 14 of these bytes have special register functions for operation of the TOY clock and watchdog timer. You can use the remaining bytes, 32754 bytes, as general-purpose bytewide read/write RAM.

3.9.6 Interval Timer

The interval timer for the Alpha VME 5/352 and 5/480 SBCs is based on the 82C54 chip. On power up, the 82C54 chip is in an undefined state and must be initialized before being used. For information on how to initialize the chip, see the DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Technical Reference.

3.9.6.1 Timers

This chip is made up of three independent 16-bit counter/timers that are functionally identical:

Table 3–1 Timers

Timer	Description
Timer 0	Must be clocked externally by P2 pin C13. Optionally, this timer's gate input can be driven by P2 pin C14. When this timer makes a low-to-high transition, its output causes the assertion of an input request (IRQ). To dismiss the IRQ, you need to access the timer interrupt status register.
Timer 1	Operates as a rate generator with its output being driven off-module by P2 pin C12. This timer is clocked by a fixed 10 MHz. The output is also routed directly to the VIC local IRQ input <3>.
Timer 2	Operates as a rate generator with its output connected to P2 pin C11. This timer is clocked with the same fixed 10 MHz. You can also use the output on the module to generate an IRQ. If enabled, Timer #0's output during a transition from low-to-high causes the assertion of an IRQ. To dismiss the IRQ, you need to access the timer interrupt status register.

The timers are implemented by register/interrupt logic. The programming interface is byte wide in the Nbus region of PCI I/O space.

3.9.6.2 Timer Modes

In addition to supporting the three timers discussed in Section 3.9.6.1, the Alpha VME 5/352 and 5/480 SBCs implement two timer modes (modes 1 and 3) provided by the 82C54 chip for timers 1 and 2. The hardware connections for the timer output are available on the P2 VMEbus connector. The timers are driven from an internally generated 10 MHz asynchronous clock.

Table 3–2 Timer Modes

Mode	Description
1	Allows the application to write a value *n* to the timer. An external hardware trigger causes the timer to count down from *n* to zero. If a new value *n* is written to an associated mode 1 register before the countdown reaches zero, the timer begins counting from the new value at clock *n*+1.
3	Allows the application to write a value *n* to the timer. The timer uses the value to generate a square wave with a period equal to *n* times the 10 MHz clock period.

Applications can uses these timers for a variety of off-module functions. For more information about how to use the timers and timer modes, see the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Technical Reference*.

3.9.7 Keyboard and Mouse Controller

The keyboard and mouse controller is provided by an Intel 82C42PE single-chip microcomputer. The controller is programmed to be IBM PC/AT compatible and can drive the keyboard and PS/2 type mouse supported by DECpc systems. The keyboard and mouse ports are female 6-pin mini-DIN, PS/2 type connectors. The controller is programmed to allow either device to operate in either port.

3.9.8 Super I/O Chip

The FDC37C665GT Super I/O chip (not to be confused with the standard I/O, or SIO, chip) supports serial-line port channels A and B (16550 UARTS) and a parallel port. It provides first-in-first-out (FIFO) data access for the serial ports and EPP/ECP modes for the parallel port.

The Alpha VME 5/352 and 5/480 SBCs use channel A for the console. The firmware configures this channel as an asynchronous line, using baud rate, parity, data bit, and stop bit configuration data that you define and is stored in NVRAM. If NVRAM does not contain valid data on power-up, the SBC configures channel A with defaults of 9600 baud, no parity, eight bits, and one stop bit.

The system firmware does not commit or initialize channel B.

3.10 VME Interface

The PCI-to-VME interface for the Alpha VME 5/352 and 5/480 SBCs conforms to the IEC 821, IEEE1014–1987, and D64 sections of IEEE1014 Rev.D (draft) standards. The interface is implemented using the following components:

- VIP ASIC (DC7047B) chip

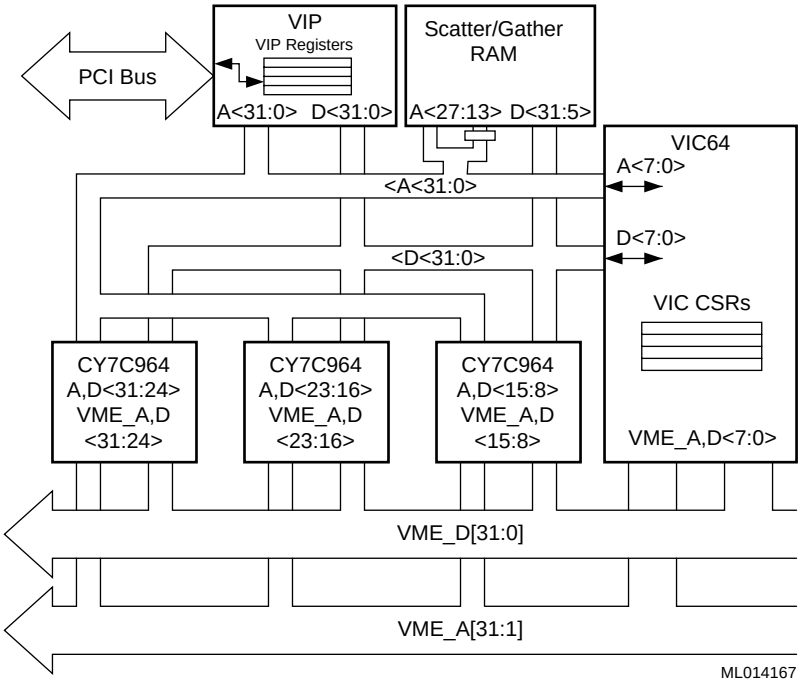
- The Cypress Semiconductor VIC64 VMEbus interface chip set
- Three CY7C964 bus interface chips
- Static scatter-gather RAM for address mapping
- Support logic implemented with programmable logic devices (PLDs)

The VIP/VIC64 chip combination accepts and generates VMEbus D08, D16, D32, and D64 data transfers and protocols. The chip combination supports addressing modes A16, A24, and A32 as a master or slave on the VMEbus.

The VIP chip uses information stored in the scatter-gather RAM to perform big-to-little endian data translation (byte swapping) and address mapping when data moves to and from the VMEbus.

Figure 3–4 shows the interface components and the address and data paths between them.

Figure 3–4 PCI-to-VME Interface Components



3.10.1 VIP Chip

The VIP chip controls the 32-bit wide PCI bus. Its PCI configuration registers allow it to function as the PCI bus target and initiator. The VIP chip:

- Functions as a PCI slave to all processor I/O read and write operations that target the VIP registers, the CY7C964 chip registers, the scatter/gather RAM, or VME memory space
- Responds to PCI interrupt acknowledge cycles when set up as the PCI interrupt responder
- Functions as a PCI master in response to the VIC64 chip requesting data from or sending data to PCI memory

- Performs address translation between the PCI bus and the VMEbus for transfers to and from the VMEbus

3.10.2 VIC64 and CY7C964 Chips

The VIC64 and CY7C964 chips control the VMEbus. The VIC64 chip functions as a VMEbus slave in response to VME addresses that match those set up by the address base and address base mask registers. This chip functions as VMEbus master:

- In response to the processor reading from and writing to VME memory (programmed I/O)
- To execute DMA transactions (master block transfers) set up by the processor in the VIP/VIC64 interface

For more information on the VIC64 and CY7C964 chips, see the Cypress Semiconductor *VIC068 User's Guide*, VIC64 design notes, and *CY7C964 User's Guide*.

3.10.3 Address Mapping and the Scatter-Gather Map

The VIP chip translates addresses by using a mapping table in scatter-gather RAM called the scatter-gather map. The scatter-gather map translates addresses for outbound and inbound VMEbus transactions.

For outbound transactions, the VIP chip maps a 512 MB region of PCI memory space to the VMEbus. The outbound scatter-gather map translates a maximum of 2K naturally aligned 256 KB pages within that 512 MB region to 256 KB of naturally aligned pages on the VMEbus (A32, A24, or A16). A PCI address is used as an index into the scatter-gather map to give the corresponding VME address.

For inbound transactions, the VIP chip maps naturally aligned 8 KB regions of VMEbus A32 and A24 address spaces to naturally aligned 8 KB regions of PCI address space (memory or I/O). The inbound scatter-gather map consists of two parts. One part translates up to 2K pages (8 KB) of VMEbus A24 address space to 8 KB pages of PCI address space. The other part maps up to 16K pages (8 KB) of VMEbus A32 address space to 8 KB pages of PCI address space. An incoming VME address is used as the index to select the PCI address.

The scatter-gather map may be accessed from the PCI bus (written to or read from) under VIP control. Scatter-gather entries also contain information to control inbound accesses and byte swapping.

The VIP chip contains a single entry scatter-gather cache and a set of registers. The cache stores the last accessed outbound scatter-gather entry and its corresponding scatter-gather address index. The registers provide mapping for inbound and outbound transactions (one mapping in each direction).

For more information about VME interface address mapping, see the *DIGITAL Alpha VME 5/352 and VME 5/480 Single-Board Computers Technical Reference*.

Part II
The Console

Part II discusses the console interface for the DIGITAL Alpha VME 5/352 and 5/480 single-board computers (SBCs). This part consists of the following chapters:

- Chapter 4, Console Basics
- Chapter 5, Using the Console
- Chapter 6, Console Command Reference

4
Console Basics

The Alpha VME 5/352 and 5/480 SBC console provides an interface to the SBC firmware. From a video terminal or video terminal emulator, you can use console firmware commands to perform operations such as configuring your system, debugging your application, embedding script code in the NVRAM, or updating the firmware.

This chapter introduces you to console basics by:

- Explaining required serial-line settings, Section 4.1
- Identifying console features, Section 4.2
- Explaining how to enter console mode, Section 4.3
- Explaining how to exit console mode, Section 4.4
- Discussing online help, Section 4.5
- Providing an overview of console commands, Chapter 4.6
- Describing special keys, Section 4.7
- Discussing command line characteristics, Section 4.8
- Describing console command operators, Section 4.9
- Explaining how to control the radix of command input, Section 4.10
- Explaining how to use flow control, Section 4.11
- Explaining how to filter output, Section 4.12
- Explaining how to redirect I/O, Section 4.13
- Explaining how to run commands in background mode, Section 4.14
- Discussing the use of scripts, Section 4.15
- Explaining how to copy scripts over the network, Section 4.16

4.1 Setting Up the Console for Use

To use the console firmware, you need to connect your SBC to a console device. The console device can be a video terminal connected with a serial line or a PC or workstation connected to the system through the network running a terminal emulator.

For information on installing serial-line or network cables, see the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers Installation Guide*.

Once you have connected the SBC to a console device, set up the device to use the following parameters:

- Send/receive 9600 baud
- DEC VT100 (ANSI)
- Eight (8) bit data word
- No parity
- One (1) stop bit
- Xon/Xoff

4.2 Console Features

The Alpha VME 5/352 and 5/480 SBC console environment is extremely powerful and features:

- An operator interface
- An operating system bootstrap mechanism
- Operating system restarts
- Self-test and extended functional diagnostics

You can use UNIX command methods, such as piping, I/O redirection, and command-level scripting. Because the console is built around a multitasking kernel, it can support more complex functions, such as system exercisers, the Maintenance Operations Protocol (MOP) listener, and remote console operations.

4.3 Entering Console Mode

Console mode provides the user interface to the SBC's firmware. You enter this mode automatically when the power-on self-test (POST) completes. Upon entering console mode, the system displays the following prompt:

```
>>>
```

The system also enters console mode when:

- You press the Halt/Reset switch on the front panel.
- The SBC receives a VMEbus reset signal and configuration switch 3 on the I/O module is enabled.
- You use the operating system command for entering console mode.
- The operating system executes a HALT instruction.
- The watchdog timer is enabled, and the system software allows the timer to time out.
- You initiate an external hardware reset by using pin C10 on the P2 VMEbus connector.

Note

Depending on the operating system and applications running at the time, pressing the Halt/Reset switch or receiving a VMEbus reset signal with configuration switch 3 enabled could damage application files.

4.4 Exiting Console Mode

To exit console mode, use the console command **boot**.

4.5 Online Help

The Alpha VME 5/352 and 5/480 SBC console provides online help for each console command. Sections 4.5.1 through 4.5.3 discuss:

- How to display online help
- How to display help for multiple commands
- How to control the display of online help

4.5.1 Displaying Online Help

To display online help, specify the **help** or the **man** command with the name of the command for which you are seeking help. If you do not specify a command name, the console displays a complete listing of console commands.

For help on the **help** or **man** command, including help on the symbols used to represent syntax, specify **help** or **man** with the **help** command as shown in the following example:

```
>>> help help
```

4.5.2 Displaying Online Help for Multiple Commands

You can request help on multiple commands in a single command line by separating the command names with a space or by using wildcards. The following example shows how to display help on the **examine** and **deposit** commands:

```
>>> help examine deposit
NAME
       examine
FUNCTION
       Display data at a specified address.
SYNOPSIS
       examine [-{b,w,l,q,o,h,d}] [-{physical,virtual,gpr,fpr,ipr}]
              [-n <count>] [-s <step>]
              [<)device>:]<address>

NAME
       deposit
FUNCTION
       Write data to a specified address.
SYNOPSIS
       deposit [-{b,w,l,q,o,h}] [-{physical,virtual,gpr,fpr,ipr}]
              [-n <count>] [-s <step>]
              [<device>:]<address> <data>
```

To display help on all commands that begin with “st”, such as **start** and **stop**, specify an asterisk (*) as follows:

```
>>> help st*
```

4.5.3 Controlling the Display of Online Help

If full help is available, the commands **help *** and **man *** display all information on all commands. To control the amount of help text that the console displays at a time, combine the **help** or **man** command with the **more** command. The following example combines the **help** and **more** commands:

```
>>> help * | more
```

This command combination displays a screen of text at a time. Press the spacebar to continue the display or press Ctrl/C to terminate the display.

4.6 Console Command Overview

The Alpha VME 5/352 and 5/480 SBC console interface consists of a set of commands for operating a system, running diagnostics, and verifying application design. Some of the commands are similar in function to UNIX commands.

Chapter 6 describes the console commands in detail. Table 4–1 shows a sampling of the most commonly used commands.

Table 4–1 Commonly Used Console Commands

Command	Description
boot	Bootstraps the system
cat	Copies the contents of files to standard output
deposit	Writes data to a specified address
echo	Sends specified text to the current output device
eval	Evaluates a specified expression
examine	Displays the contents of a specified address
exer	Exercises system devices with read, write, and comparison operations
grep	Searches for expressions and writes the search results to standard output
hd	Dumps the contents of a file
help	Displays the definition and syntax for specified commands
ls	Displays a listing of files in the system
man	Displays the definition and syntax for specified commands
memexer	Executes memory test processes in the background
memtest	Executes memory tests
ps	Displays the status and statistics associated with system processes
sa	Specifies the processors on which a specified process can run

Table 4–1 Commonly Used Console Commands (Continued)

Command	Description
set	Sets the value of an environment variable
show	Displays information about the system
sleep	Suspends execution of a console process

Most console commands require that you specify arguments. In most cases, you can also specify options that give you a finer level of control over the command's execution. Options have a hyphen (-) prefix, as in -b. When specifying options, you must separate the option from the command and arguments with spaces. For example, you must specify `e -b 0`. If you enter `e-b 0`, the console issues an error message.

4.7 Special Keys

Table 4–2 lists special key and key combinations that perform specific console operations.

Table 4–2 Special Keys for Console Operation

Keys	Operation
Ctrl/U	Ignores the current command line.
Backspace/Delete	Deletes a character within the command line.
Ctrl/S	Suspends command output to the console terminal.
Ctrl/Q	Resumes command output to the console.
Ctrl/C	Aborts the current command, if possible. The console program has no control of an abort once it passes control to another program, such as an operating system or loadable diagnostic.
Ctrl/R	Retypes the current command line.
Ctrl/O	Causes the console to throw away output characters rather than send them to the terminal. Entering another Ctrl/O resumes sending output characters to the terminal.
Up and down arrows	Recall command lines.

4.8 Command Line Characteristics

The character sequence used for the console prompt (>>>) is:

```
0Dh 0Ah 0Dh 3Eh 3Eh 3Eh 20h
```

This translates to:

```
<CR>  <LF> <CR>   > > >  <SP>
```

Host system software executing a binary load operation on the console terminal port can look for this character string to determine when to respond.

Commands are limited to 80 characters. Characters that you enter beyond the 80-character limit replace the last character in the buffer. Depending on your terminal, the lost characters may be displayed, but they are not included in the actual command line.

The command interpreter is not case-sensitive. Lowercase ASCII characters a through z are treated as uppercase characters.

The parser rejects characters with codes greater than 0x7F. However, such characters are acceptable in comments.

The console does not provide type-ahead buffer support. The console checks characters received before the console prompt appears for special characters (Ctrl/S, Ctrl/Q, Ctrl/C), but otherwise discards the characters.

4.9 Console Command Operators

Table 4–3 lists operators that extend the console command interface.

Table 4–3 Console Command Operators

Operator	Name	Description
>	Output creation	Writes output to a specified destination, such as a file. **Form:** > *destination*
>>	Output append	Adds output to a specified destination, such as a file. **Form:** >> *destination*
<	Input redirection	Reads input from a specified source. **Form:** < *source*
<<	Here document	Reads input from standard input until a specified string is found at the beginning of a line. **Form**: << *string*
|	Pipe	Uses the output of the first command as the input for the second command. **Form**: *cmd1* | *cmd2*
;	Sequence	Runs the first command to completion before running the second command. **Form**: *cmd1*; *cmd2*
\	Line continuation	Continues the command on the next line. The prompt changes to _> until the command is completed. **Form**: *cmd1* \ _> *cmd2*

Table 4–3 Console Command Operators (Continued)

Operator	Name	Description
#	Line comment	Ignores the text that follows the operator. Used for embedding comments in command scripts or logs. **Form**: # *text*
&	Background	Runs the command in a background process. The command line remains available for a new command. **Form**: *cmd1* &
&a	Affinity	Runs the process on the CPU that is allowed by the processor affinity mask, *m*. You can specify multiple processors by using a list or range. **Form**: &a *m*
( ), { }, ‘ ‘	Grouping	Shows which commands are grouped together in complex command lines. These operators override the precedence of pipe, sequence, and background operators. **Form**: {*cmd1*; *cmd2*} | *cmd3*
*, ?, [...]	Pattern specifiers	Specifies a character or group of characters to match in character strings. * matches any characters or none ? matches any single character [...] matches any of the enclosed characters **Form:** str*, arg?, [1 2 3]

4.10 Controlling the Radix of Command Input

By default, the console interprets numbers that you enter in a console command line as hexadecimal. To change the radix of command input to decimal, precede the input value with %d. To explicitly specify a hexadecimal radix, precede the input value with %x.

4.11 Using Flow Control

The console provides reserved words that you can use in flow control structures. The reserved words include:

case	**elif**	**fi**	**in**	**while**
do	**else**	**for**	**then**	
done	**esac**	**if**	**until**	

The syntax for valid control structures follows:

- **while** *command_sequence* **done**

- **while** *command_sequence* **do** *command_sequence* **done**
- **until** *command_sequence* **done**
- **until** *command_sequence* **do** *command_sequence* **done**
- **for** *name* **do** *command_sequence* **done**
- **for** *name* **in** *list* **do** *command_sequence* **done**
- **case** *word* **in** *case_part_list*
  *pattern***)** *command_sequence* **;;**
  [*pattern* **)** *command_sequence* **;;**]
  esac
- **if** *command_sequence*
  then *command_sequence*
  [**elif** *command_sequence* **then** *command_sequence*)]
  [**else** *command_sequence*]
  fi

The console determines conditional branching in **if**, **while**, and **until** loops by checking the exit status of the command sequence that follows the control structure. In general, an exit status of zero indicates success and results in the execution of the true path.

The following example uses the **eval** command to extract an exit status from variable *junk*. The console command **set** initializes the variable.

```
>>> set junk 0
>>> show junk
junk                    0
>>> eval junk
0
>>> if (eval junk) then (echo true) else (echo false) fi
0
true
>>> set junk 1
>>> if (eval junk) then (echo true) else (echo false) fi
1
false
>>> set junk 2
>>> if (eval junk)
_> then (echo true)
_> else (echo false) fi
2
false
>>>
```

4.12 Filtering Output

You can search for specific values in a device by using a pipe with the **grep** command. A *pipe* (|) enables the output of one command to be the input for the next command without creating an intermediate file. The **grep** command filters its input according to the command argument. Because the **grep** command requires input, a pipe is used to channel the output of the **examine** command into the **grep** command.

The following example uses **grep** to search for a pattern in memory. In this case, **grep** parses all the output lines from the **examine** command, but only permits lines that contain *ABCDEF12* to reach the display. You can also use the **grep** command to search for patterns that do not match the model provided; that is, it searches for every line that does **not** contain the input pattern. The following example sets up memory and then uses **grep** to filter the output.

```
>>> d pmem:3fff000 0 -n 8  # Clear some memory.

>>> d 3fff020 ABCDEF12     # Drop in a target.

>>> e 3fff000 -n 8         # Display memory.

pmem:          3FFF000 0000000000000000
pmem:          3FFF008 0000000000000000
pmem:          3FFF010 0000000000000000
pmem:          3FFF018 0000000000000000
pmem:          3FFF020 00000000ABCDEF12
pmem:          3FFF028 0000000000000000
pmem:          3FFF030 0000000000000000
pmem:          3FFF038 0000000000000000
pmem:          3FFF040 0000000000000000

>>> e 3fff000 -n 8 | grep ABCDEF12# Display only lines with ABCDEF12.

pmem:          3FFF020 00000000ABCDEF12
```

4.13 Redirecting I/O

By default, console commands display on the console terminal. You can redirect output to other devices or files by using the redirection operator (>). In the following example, the output of the **examine** command is redirected to file `foo`, which is created dynamically from the console's memory heap. The console command **cat**, displays the contents of the new file. The **rm** command deletes the `foo` file.

```
>>> ls foo           # Check to see if foo exists.

foo no such file

>>> e 3fff000 -n 1 > foo# Redirect examine output to file foo.

>>> ls foo           # Check to see if foo exists.

foo

>>> cat foo          # Display foo.

pmem:          3FFF000 0000000000000000
pmem:          3FFF008 0000000000000000

>>> rm foo           # Delete (remove) file foo.

>>> ls foo           # Check to see if foo exists.

foo no such file
```

4.14 Running Commands in Background Mode

You have the option of executing console commands in background mode. When a command executes in background mode, the console creates a process for executing the command and leaves the main process available for you to enter a new command. You can execute any console command in the background by placing the background operator **&** at the end of the command.

The following example starts three processes in the background. The **exer** command invokes the first process, which reads data from block 0 of a disk. Then, the **memtest** command creates two processes that perform console memory tests. In all three cases, the console immediately returns with the console prompt and waits for you to enter another command.

```
>>> show device              # See what devices are available.

dka0.2.0.1.0               dka0              dka0
eza0.0.0.0.0               EZA0              08-00-2B-1D-02-91
ezb0.0.0.1.0               EZB0              08-00-2B-1D-02-92
pka0.7.0.2.0               PKA0              SCSI Bus ID 7

>>> exer dka0 -sb 0 -p 0 & # Read block 0 forever.

>>> memtest -p 0 &         # Start up the memory test forever.

>>> memtest -p 0 &         # Start up another memory test task.

>>>
```

4.15 Creating Scripts

A script is a file that contains a sequence of console commands. The console firmware contains many scripts, such as the power-up script, that you can run by typing the name of the script file.

If you have a complex command or a series of commands that you have to use frequently, you can write a script for your convenience. Use the **echo** command and the output creation operator (**>**) to write characters to a file. The file is the script. The following example creates the script `foo`, which invokes the **examine** command.

```
>>> echo e pmem:3fff000 > foo# Write "e 0" to file foo.

>>> cat foo                 # List foo.

e pmem:3fff000

>>> foo                     # Execute script foo.

pmem:          3FFF000 0000000000000000
```

To add another command to the script, use the append operator (>>). If the command you are appending contains characters that could be interpreted by the **echo** command, specify the characters with a grouping operator. The following example uses the single quote (') grouping character to prevent the command-separator operator (**;**) in the appended command from terminating the **echo** command.

```
>>> echo 'd 3fff000 5 ; e 3fff000' >> foo # Append "d 0 5 ; e 0" to
foo.

>>> cat foo                          # List foo.

e pmem:3fff000
d 3fff000 5 ; e 3fff000

>>> foo                              # Execute foo.

pmem:          3FFF000 0000000000000000
pmem:          3FFF000 0000000000000005
```

You can also use a grouping operator to create a script that contains many commands. You have to rearrange the **echo** command so that the appended characters are at the end. Then, use the open grouping operator to open the character string and take as many lines as needed to create the script before specifying the close grouping operator. The following example shows how to create a long script using grouping operators:

```
>>> echo > foo 'ex 3fff000

_> d 3fff000 7
_> e 3fff000
_> d 3fff000 5
_> e 3fff000'

>>> cat foo

ex 3fff000
d 3fff000 7
e 3fff000
d 3fff000 5
e 3fff000

>>> foo

pmem:          3FFF000 0000000000000000
pmem:          3FFF000 0000000000000007
pmem:          3FFF000 0000000000000005
```

4.16 Copying Scripts Over the Network

The console provides a mechanism for transferring command scripts over the network. You can create scripts on an OpenVMS system and then fetch them from the console of an Alpha VME 5/352 or 5/480 SBC by using the following procedure:

1. Create a file of console commands in the OpenVMS environment, using your favorite editor. The following example shows the OpenVMS **create** command being used to create a script file called `sample`.

   ```
   $ create sample.
   show version
   ls –l sample
   (Control-Z exit)
   $
   ```

2. Make the script file compatible with the MOP load protocol. To accomplish this, run the **add_header.exe** program to append a one-block header to the file, making it compatible with the MOP load server. This executable program is on the Firmware Update CD at **[ALPHAVME]ADD_HEADER.EXE**. If you prefer, copy the file to the SYS$LOGIN area and define it as a foreign command, for example, **addhead**. To run the program, invoke **addhead** and supply the file name as input and a name for the resulting output file.

   Note

   The current MOP load protocol only supports 15-character file names. To make use of all 15 characters in the name, do not specify a file extension. The MOP server defaults to a file extension of.**sys**.

3. Place the output file in the MOP server's load file directory, MOP$LOAD. Whenever MOP gets a request for the script, it searches in its service area.

At this point, the script file is available on the Ethernet segment of the MOP server. If the Alpha VME 5/352 or 5/480 SBC is on the same Ethernet segment as the MOP server, the following example copies the script file over the network. The string, **mopdl:sample.sys/eza0**, specifies that the file, **sample.sys**, can be accessed over the Ethernet device, **eza0**, using the MOP download protocol driver, **mopdl:**.

```
>>> cat mopdl:sample.sys/eza0# Be patient! The MOP protocol is slow.

show version
ls –l sample
>>>
```

You can then use the redirection operator (>) to redirect the output of the **cat** command to a local file. The following **cat** command redirects output to **sample**.

```
>>> cat mopdl:sample/eza0 > sample# Remember be patient!
```

When the console prompt returns, the copy operation is complete. You can then display and execute the resident script file, **sample**, by using the following sequence of console commands:

```
>>> cat sample

show version
ls –l sample

>>> sample

version             V1.1-0 Jul 1 1996 10:16:59

rwx-   rd                512/2048           0   sample
```

5
Using the Console

This chapter explains how to use the console command interface to:

- Manage environment variables, Section 5.2
- Boot the system, Section 5.3
- Use TFTP to read files across the network, Section 5.4
- Manage the TOY clock, Section 5.5
- Get system information, Section 5.6
- Update firmware, Section 5.7
- Examine and deposit data, Section 5.8
- Manage the console, devices, and processor, Section 5.9
- Manage memory, Section 5.10
- Perform network operations, Section 5.11
- Set reboot to the SROM Mini-Console, Section 5.12
- Control the LED, Section 5.13
- Run the power-up diagnostics script, Section 5.14
- Manage the error log in NVRAM, Section 5.15
- Evaluate expressions, Section 5.16
- Manage console processes, Section 5.17
- Manage files and file content, Section 5.19

5.1 Summary of Console Operations

The DIGITAL Alpha VME 5/352 and 5/480 SBC console interface consists of commands for managing the operation of your SBC, running diagnostics, and verifying the integrity of your system design. Table 5–1 lists the types of operations you can perform by using the console commands.

Table 5–1 Summary of Console Operations

Operation	Command
Managing Environment Variables	
Set the value of an environment variable	**set**
Set all environment variables to their default values	**init_ev**
Delete an environment variable from the system's name space	**clear**

Table 5–1 Summary of Console Operations (Continued)

Operation	Command
Booting the System	
Boot the system	**boot**
Managing the TOY Clock	
Set or display the date and time stored in the TOY clock	**date**
Disable the TOY clock's internal oscillator	**set toy sleep**
Getting System Information	
Display the value of a specified environment variable	**show**
Display the system configuration	**show config**
Display the devices and controllers in the system	**show device**
Display the address of the Alpha hardware restart parameter block (HWRPB)	**show hwrpb**
Display the character illuminated on the LED	**show led**
Display a map of the system's virtual memory	**show map**
Updating Firmware	
Update firmware in the system's flash ROMs	**update**
Examining and Depositing Data	
Write data to a specified memory location, register, device, or file	**deposit**
Display the contents of a memory location, register, device, or file	**examine**
Managing the Console, Devices, and the CPU	
Initialize the a device or the CPU	**init**
Stop the CPU or system devices	**stop**
Start system devices	**start**
Exercise system devices with read, write, and comparison operations	**exer**
Managing Memory	
Allocate a block of memory from the system's heap	**alloc**
Free a block of memory that has been allocated from the system's heap	**free**
Change the ownership of a block of memory	**chown**
Display the state of dynamic memory	**dynamic**
Display the system's virtual memory map	**show map**
Test memory	**memtest**
Start a specified number of memory test processes that are to run in the background	**memexer**
Performing Network Operations	

Table 5–1 Summary of Console Operations (Continued)

Operation	Command
Perform maintenance operations protocol (MOP) operations, such as loopbacks, ID requests, and remote file loads	**net**
Setting Reboot to the SROM Mini-Console	
Enter Serial ROM Mini-Console after the next reboot	**set reboot srom**
Controlling the LED	
Specify a character to be displayed on the front panel LED	**set led**
Show the character currently being displayed on the front panel LED	**show led**
Running the Power-Up Diagnostics Script	
Run the power-up script	**pwrup**
Managing the Error Log in NVRAM	
Clear and initialize the area of NVRAM used for console error logging	**clear_log**
Display error log information stored in NVRAM	**show_log**
Evaluating Expressions	
Evaluate expressions	**eval**
Managing Console Processes	
Create a new shell process	**sh**
Exit the current shell process	**exit**
Start the execution of a program or driver at a specified address	**start**
Display console process status and statistics	**ps**
Delete specified console processes	**kill**
Break from a for, while, or until control loop	**break**
Return a failure status	**false**
Specify the processors on which a console process can run	**sa**
Display the semaphores known to the system	**semaphore**
Set the priority of a console process	**sp**
Suspend the execution of a console process	**sleep**
Managing Files and File Content	
Copy specified files to standard output	**cat**
Change the attributes of a specified file	**chmod**
Dump the contents of a file	**hd**
List the files and inodes that are in the system	**ls**
Remove specified files from the system	**rm**

Table 5–1 Summary of Console Operations (Continued)

Operation	Command
Sort the content of a file	**sort**
Writes specified text to standard output	**echo**
Search for expressions in specified files	**grep**
Copy a line from the input channel of a file to the standard output channel for that file	**line**

5.2 Managing Environment Variables

Environment variables define the following types of configuration information for a system's firmware and operating system:

- Boot parameters
- Console terminal characteristics
- Options associated with diagnostic tests
- Network protocols and associated characteristics and data
- Values for storage bus adapters
- Versions of PALcode and console firmware
- PCI bus settings
- Use of TGA video cards
- VMEbus settings
- VxWorks boot file

The data defined by the environment variables is stored in memory. Some of the data is stored in volatile memory and some is stored in nonvolatile memory.

You can use console commands to set and display the values of environment variables and delete environment variables from the system's name space.

5.2.1 Environment Variable Summary

Table 5–2 lists the environment variables with possible values and brief descriptions.

Table 5–2 Environment Variables

Variable	Parameter Values	Description
AUTO_ACTION	BOOT, HALT, or RESET	Defines the action of the console following an error, halt, or power-up. Default is HALT.
BOOT_DEV	–	Specifies the device list to be used by the last, or currently in progress, bootstrap attempt. The console modifies BOOT_DEV at console initialization and when a bootstrap is initiated by a **boot** command. The value of BOOT_DEV is set from the device list specified by the **boot** command or, if no deivce list is specified, BOOTDEF_DEV. The console uses BOOT_DEV without change on all bootstrap attempts that are not initiated by a **boot** command.
BOOT_FILE	*file-name*	Specifies the file name to be used when a bootstrap requires a file name, when the bootstrap is not the result of a **boot** command, or when no file name is specified with the **boot** command. The console passes the value between the console presentation layer and system software without interpretation.
BOOT_OSFLAGS	For use with UNIX: a (automatic boot) s (stop in single-user mode) i (interactive boot) D (full dump and s)	Specifies arguments to be passed to system software when the bootstrap is not the result of a **boot** command or when no arguments are specified with the **boot** command. The console passes the value between the console presentation layer and system software without interpretation. The default is NULL.
BOOTDEF_DEV	*device-list*	Specifies the device list from which bootstrapping is to be attempted when no path is specified with the **boot** command.
BOOTED_DEV	A device in the BOOT_DEV list	Specifies devices to be used by the last or currently in progress bootstrap attempt.
BOOTED_FILE	Derived from BOOT_FILE or the current **boot** command	Specifies the file name to be used by the last or currently in progress bootstrap attempt. The console passes the value between the console presentation layer and system software without interpretation.
BOOTED_OSFLAGS	Derived from BOOT_OSFLAGS or the current **boot** command	Specifies arguments to be passed to system software during the last or currently in progress bootstrap attempt. The console passes the value between the console presentation layer and system software without interpretation.
CHAR_SET	0 (ISO-LATIN_1)	Specifies current console terminal character-set encoding.
CONSOLE	–	Specifies whether console input and output are to use the console serial line or a graphics console, if present.

Table 5–2 Environment Variables (Continued)

Variable	Parameter Values	Description
D_BELL	ON or OFF	Specifies whether the bell is to sound on error. The default is OFF.
D_CLEANUP	ON or OFF	Specifies whether cleanup code is to be executed at the end of diagnostics. The default is ON.
D_COMPLETE	ON or OFF	Specifies whether a diagnostic completion message is to be displayed. The default is OFF.
D_EOP	ON or OFF	Specifies whether end-of-pass messages are to be displayed. The default is OFF.
D_GROUP	FIELD, MFG, or other (up to 32 characters)	Specifies the diagnostic group to be executed. The default is FIELD.
D_HARDERR	CONTINUE, HALT, or LOOP	Defines the action that is to be taken following a hard error detection. The default is HALT.
D_OPER	ON or OFF	Specifies whether an operator is present. The default is OFF.
D_PASSES	0 (run indefinitely), 1 (pass), or a user-defined value	Specifies the diagnostic pass count. The default is 1.
D_REPORT	SUMMARY, FULL, or OFF	Specifies the level of information to be provided by diagnostic error reports. The default value is FULL.
D_SOFTERR	CONTINUE, HALT, or LOOP	Defines the action that is to be taken following soft error detection. The default is CONTINUE.
D_STARTUP	ON or OFF	Specifies whether a diagnostic startup message is to be displayed. The default is OFF.
D_TRACE	ON or OFF	Specifies whether trace messages are to be displayed. The default is OFF.
DUMP_DEV	*device*	Specifies that a device is to write operating system crash dumps.
ENABLE_AUDIT	ON or OFF	Specifies whether audit trail messages are to be generated during bootstrap. The default is ON.
EWA0_ARP_TRIES	*n*	Specifies the number of transmissions to be attempted before the Internet Address Resolution Protocol (ARP) fails. Values less than 1 cause the protocol to fail immediately. The default is 3, which translates to an average of 12 seconds before failing. Interfaces on busy networks may need higher values.
EWA0_BOOTP_FILE	*file-name*	Specifies a generic file name to be included in an Internet Boot Protocol (BOOTP) request. The BOOTP server returns a fully qualified file name for booting. There is no default.
EWA0_BOOTP_SERVER	*server-name*	Specifies a server name to be included in a BOOTP request. This can be set to the name of the server from which the machine is to be booted, or left empty.

Table 5–2 Environment Variables (Continued)

Variable	Parameter Values	Description
EWA0_BOOTP_TRIES	*n*	Specifies the number of transmissions that are to be attempted before BOOTP fails. Values less than 1 cause the protocol to fail immediately. The default is 3, which translates to an average of 12 seconds before failing. Interfaces on busy networks may need higher values.
EWA0_DEF_GINETADDR	–	Specifies the initial value for EWA0_GINETADDR when the interface's internal Internet database is initialized from BOOTP (EWA0_INET_INIT is set to BOOTP).
EWA0_DEF_INETADDR	–	Specifies the initial value for EWA0_INETADDR when the interface's internal Internet database is initialized from BOOTP (EWA0_INET_INIT is set to BOOTP).
EWA0_DEF_INETFILE	–	Specifies the initial value for EWA0_INETFILE when the interface's internal Internet database is initialized from BOOTP (EWA0_INET_INIT is set to BOOTP).
EWA0_DEF_SINETADDR	–	Specifies the initial value for EWA0_SINETADDR when the interface's internal Internet database is initialized from BOOTP (EWA0_INET_INIT is set to BOOTP).
EWA0_INET_INIT	NVRAM and default BOOTP	Specifies whether the interface's internal Internet database is to be initialized from non-volatile RAM (NVRAM) or from a network server (by way of BOOTP).
EWA0_LOOP_COUNT	*x*	Specifies the number of times each message is looped. The default is 0x3e8.
EWA0_LOOP_INC	*x*	Specifies the amount the message size is to be increased from message to message. The default is 0xa.
EWA0_LOOP_PATT	0xffffffff = all patterns 0 = all zeros 1= all ones 2 = all fives 3 = all as 4 = incrementing 5 = decrementing	Specifies the type of data pattern that is to be used for loopback.
EWA0_LOOP_SIZE	*x*	Specifies the size of the loop data to be used. The default is 0x2e.
EWA0_LP_MSG_NODE	*n*	Specifies the number of messages to be sent to each node originally. The default is 7.
EWA0_MODE	TWISTED-PAIR or FULL (full-duplex twisted-pair)	Specifies the operating mode of the embedded Ethernet controller.

Table 5–2 Environment Variables (Continued)

Variable	Parameter Values	Description
EWA0_PROTOCOLS	BOOTP, MOP, or BOOTP,MOP	Specifies the network protocol to be enabled for booting and other functions. The default is MOP. A null value is equivalent to BOOTP,MOP.
EWA0_TFTP_TRIES	*n*	Specifies the number of transmissions that are to be attempted before the Trivial File Transfer Protocol (TFTP) fails. Values less than 1 cause the protocol to fail immediately. The default value is 3, which translates to an average of 12 seconds before failing. Interfaces on busy networks may need higher values.
LANGUAGE	00 none (cryptic) 30 Dansk 32 Deutsch 34 Deutsch (Schweiz) 36 English (American) 38 English (British/Irish) 3A Espanol 3C Francais 3E Francais (Canadian) 40 Francais (Suisse Romande) 42 Italiano 44 Nederlands 46 Norsk 48 Portugues 4A Suomi 4C Svenska 4E Vlaams Other reserved	Specifies the current console terminal language (integer ID).
LANGUAGE_NAME	*language-name*	Specifies the ASCII string of the current console terminal language code as defined by LANGUAGE.
LICENSE	MU – multi-user system SU – single-user system	Specifies whether a software license is in effect.
MODE	FASTBOOT or NOFASTBOOT	Specifies whether diagnostics are to be run when the firmware is initialized.
PAL	*n*	Specifies versions of VMS and OSF PALcode in the firmware.
TGA_SYNC_GREEN	*x*	Specifies a hexadecimal byte indicating whether video synchronization should be driven on the green channel for up to eight TGA video cards. Video card 0 corresponds to bit 0, card 1 to bit 1, and so on. Use with the CONSOLE environment variable.
TTY_DEV	*n*	Specifies the current console terminal unit. Indicates which entry of the CTP Table corresponds to the actual console terminal. The default is 0 (30 hex).
VERSION	*version*	Specifies the version of the console code firmware.
VME_A32_BASE	*address*	Specifies the base address of VMEbus A32 space.
VME_A32_SIZE	*n*	Specifies the size of VMEbus A32 space.

Table 5–2 Environment Variables (Continued)

Variable	Parameter Values	Description
VME_A24_BASE	*address*	Specifies the base address of VMEbus A24 space.
VME_A24_SIZE	n	Specifies the size of VMEbus A24 space.
VME_A16_BASE	*address*	Specifies the base address of VMEbus A16 space.
VME_CONFIG	*mode*	Specifies the VME setup mode. This variable is used by the operating systems for storing VME configuration information for the initialization of the VME corner. See your operating system documentation for more information.
VX_BOOTLINE	*file-name*	Specifies the name of the file to be used for the VxWorks bootstrap.

5.2.2 Setting Environment Variables

To set the values of environment variables, use the **set** command. This command requires that you specify the name of an environment variable and either a numeric or ASCII string value. Section 5.2.1 provides a complete listing of available environment variables.

If at any time you need to restore a variable to its default value, you can do so by using the **set** command's **-default** option. Or, if you want to set all environment variables to their default values at the same time, use the **init_ev** command.

For any environment variable changes that you make with the **set** or **init_ev** command to take effect, you must reset the system or issue the **init** command.

Note

Before you change the value of an environment variable, you should understand the implications of the change.

5.2.3 Displaying the Values of Environment Variables

You can display the values of environment variables by using the **show** command. As indicated in the following table, the extent of this command's output depends on the argument that you specify.

To display...	Specify...
The value of a specific variable	The name of that variable
The values of a group of related variables	A name that includes a wildcard (*); for example, BOOT*
The values of all variables	No argument

To see the changes to variables that you reset, you must reset the system or issue the **init** command before using **show**.

5.2.4 Removing Environment Variables from System Name Space

If a subset of the environment variables do not apply to your system configuration, you may want to consider removing them from the system name space. To remove a variable from the name space, specify the variable as an argument to the **clear** command. If you specify a variable name that includes a wildcard, such as EWA0_*, the command removes a group of related variables from the name space. In the case of the EWA0_* example, the command removes all environment variables that begin with EWA0_.

Note

Some environment variables are permanent and are not affected by this **clear** command.

5.3 Booting the System

You boot your SBC to initialize the processor, load a program image, and transfer control to that image. To initiate a boot operation, use the **boot** command. In the command line, you have the option of specifying:

- One or more devices from which the system is to be booted
- A program image to be booted
- Boot flags for passing additional information along to the operating system
- The protocol to be used for booting over the network
- That the console gain control immediately after the boot image is loaded

5.3.1 Specifying Boot Devices

You can specify the boot device for an SBC by setting the value of the environment variable BOOTDEF_DEV or by specifying one or more devices with the **boot** command. BOOTDEF_DEV defines a default boot device list.

To override the default boot device list, specify one or more devices with the **boot** command. If you specify multiple devices, separate device names with a comma (without spaces). The console firmware attempts to boot the system from each device in order. When a device boots successfully, the firmware passes control to the boot image on that device.

Note

If you include network devices in the boot device list, place them at the end of the list. This is necessary because network boots terminate only if a fatal error occurs or an image loads successfully.

5.3.2 Specifying a Boot Image

When an Alpha VME 5/352 or 5/480 SBC boots successfully, the console firmware passes control to a boot image. You can specify the boot image that is to be used by setting the value of the environment variable BOOT_FILE or by specifying the file name of a boot image with the **boot** command. BOOT_FILE defines the default boot image. To override the default, specify a boot image file name with the **-file** option in the **boot** command line.

5.3.3 Passing Additional Boot Information to the Operating System

You have the option of passing boot information, in addition to the boot image, to the operating system. You can specify the additional information as longword data in the definition for the BOOT_OSFLAGS (or BOOTED_OSFLAGS) environment variable or with the **boot** command. The information can consist of one or more longword values. If you specify multiple values, separate the values with a comma (without spaces). The environment variables define the default boot information. To override the default, specify boot information with the **boot** command's **-flags** option.

5.3.4 Booting Over the Network

If you choose to boot an Alpha VME 5/352 or 5/480 SBC over the network, you need to define the Ethernet protocol that is to be used. Depending on your system configuration, you can use the DECnet maintenance operation protocol (MOP), the Internet boot protocol (BOOTP), or both.

You can specify the protocols to be used by setting the value of the environment variable EWA*n*_PROTOCOLS (*n* identifies the network interface) or the **boot** command's **-protocols** option to MOP or BOOTP. If you specify both protocols, the console firmware tries to use each protocol in the order listed to solicit a boot server. If you do not define EWA*n*_PROTOCOLS, both protocols are enabled.

The following example causes the console firmware to try to use BOOTP and then MOP to complete a network boot using interface ewa0:

```
>>> set EWA0_PROTOCOLS BOOTP,MOP
```

5.3.4.1 Internet Protocols

For the Internet environment, the console uses the protocols BOOTP and TFTP to support network booting and file transfers. An Internet network boot occurs as follows:

1. BOOTP broadcasts a boot request.

   BOOTP copies the values of the environment variables EWA*n*_BOOTP_SERVER and EWA*n*_BOOTP_FILE to the fields *sname* and *file* in the request packet. The *sname* field specifies the host from which the SBC wants to boot. If it does not matter which server responds to the request, you can leave EWA*n*_BOOTP_SERVER undefined.

The *file* field identifies the boot file the server is to include in its response. For example, if the file is specified generically as “unix” or “lat”, the boot server would respond with a fully qualified file path to be used with TFTP. If a machine will always be booting the same file, you can leave EWA*n*_BOOTP_FILE undefined.

BOOTP establishes a connection with a boot server, which in turn provides the SBC with the information it needs to obtain the boot image from the server. The BOOTP server delivers the information in a message packet. Using the same format, the SBC stores the information in a 300-byte Internet database. When the SBC receives the BOOTP packet, the database is marked as initialized.

2. The SBC uses TFTP to acquire the boot image.

   TFTP uses the remote host address and the file name of the boot image to get the boot image file from the boot server (host system). TFTP gets this information from the BOOTP packet, the **boot** command’s *file-name* argument, or the BOOT_FILE environment variable.

   If the value of BOOT_FILE is not specified in the correct format, TFTP fails. A common practice used to avoid this failure is to leave BOOT_FILE undefined. This causes TFTP to default to using the values of EWA*n*_DEF_SINETADDR and EWA*n*_DEF_INETFILE.

Both BOOTP and TFTP use the Internet user datagram protocol (UDP) as their primary transport mechanism. UDP is an unreliable, connectionless datagram delivery service.

For complete descriptions of the Internet protocols, see Douglas Comer’s *Internetworking with TCP/IP, Vol I, Principles, Protocols and Architecture*, Second edition, Prentice Hall.

5.3.4.2 Defining Fields of the Internet Database

BOOTP and TFTP rely on Internet configuration information that you define for the system by setting network environment variables. You must define a set of variables for each network interface in the system. The system stores the configuration information for each interface in a separate 300-byte Internet database. These databases have the same format as BOOTP packets; the BOOTP driver reads from and writes to the databases in binary form directly.

The following table lists the environment variables that define the most important work data. Unlike other environment variables, these variables are nonvolatile.

Environment Variable	Description
EWA*n*_DEF_INETADDR	The Internet address of a network interface on the SBC. The Internet protocols TFTP and address resolution protocol (ARP) require the correct Internet address to operate properly. Enter the address in dotted decimal notation (*n.n.n.n*).
EWA*n*_DEF_SINETADDR	The Internet address of the remote host system to be contacted by TFTP. The remote host system might not be on the local area network (LAN). Enter the address in dotted decimal notation (*n.n.n.n*).
EWA*n*_DEF_GINETADDR	The Internet address of a remote Internet gateway on the LAN. TFTP cannot communicate beyond the LAN if this address is incorrect. Enter the address in dotted decimal notation (*n.n.n.n*).
EWA*n*_DEF_SUBNETMASK	The Internet subnet mask to be used. Enter the mask in dotted decimal notation (*n.n.n.n*).
EWA*n*_DEF_INETFILE	The file to be booted. The value that you specify must be a valid file name or path name for the TFTP server on the remote host system.

Each network interface must have its own set of variable definitions. The variable *n* in the names of the preceding environment variables, identifies a specific interface. For example, all variables associated with network interface 0 have the prefix EWA0.

Note

If you misconfigure the Internet network parameters, the Internet protocols are robust enough to work intermittently, making it difficult to debug failures.

5.3.4.3 Internet Database Initialization

The Internet database on an Alpha VME 5/352 or 5/480 SBC is initialized each time the system is booted as a result of a TFTP or BOOTP invocation.

TFTP Initialization

A TFTP invocation is the more common form of Internet database initialization. If TFTP is invoked and the Internet database has not yet been marked as initialized, initialization occurs automatically, based on the definition of the environment variable EWA*n*_INET_INIT. If this variable is set to BOOTP (the default), the BOOTP protocol driver broadcasts a BOOTP request and stores the response in the database, initializing it.

If EWZ*n*_INET_INIT is set to NVRAM, the values of the following nonvolatile Internet environment variables are copied to corresponding fields in the Internet database:

EWA*n*_DEF_INETADDR

EWA*n*_DEF_SINETADDR

EWA*n*_DEF_GINETADDR

EWA*n*_DEF_SUBNETMASK

EWA*n*_DEF_INETFILE

TFTP assumes that you have set the values of these variables in advance of its invocation. For example:

```
>>> SET EWA0_DEF_INETADDR 16.123.16.53
>>> SET EWA0_DEF_SINETADDR 16.123.16.242
>>> SET EWA0_DEF_GINETADDR 16.123.16.242
>>> SET EWA0_DEF_SUBNETMASK 255.255.255.0
>>> SET EWA0_DEF_INETFILE bootfiles/alphavme5
>>> SET EWA0_INET_INIT NVRAM
```

BOOTP Initialization

Alternatively, the Internet database might be initialized by BOOTP. This may result from an explicit invocation of BOOTP or as a consequence of invoking TFTP. Generally, BOOTP copies the reply packet it receives into the Internet database, initializing it. However, if BOOTP is invoked with the NOBROADCAST parameter, as shown below, no request is broadcast, no reply is received, and no data is placed in the database:

```
bootp:nobroadcast/ewa0
```

5.3.4.4 Using Retransmission to Improve Robustness

The Internet protocols ARP, BOOTP, and TFTP retransmit failed message packets to improve robustness. If the initial transmission of a packet is not answered appropriately, the protocol software retransmits the packet. By default, the protocols attempt three transmissions.

If your Alpha VME 5/352 or 5/480 SBC is on a busy network or is associated with servers that handle heavy network loads, you may need to increase the retransmission count. You can adjust the number of retransmissions associated with a given protocol by setting the following environment variables:

EWA*n*_ARP_TRIES

EWA*n*_BOOTP_TRIES

EWA*n*_TFTP_TRIES

If you set one of these variables to a value that is less than one, the protocol fails immediately.

Three retries translates to an average of 12 seconds before failing. The retransmission algorithms use a randomized exponential backoff delay. If the first try fails, a second try occurs about 4 seconds later. A third try occurs after another 8 seconds, a fourth after 16 seconds, and so on, up to 64 seconds. These times are averages since random jitter of about +/- 50% is added to each delay. For example, if EWA0_ARP_TRIES is 3, ARP fails if it does not get a response within 12 seconds on the average; the actual timeout is between 6 and 18 seconds.

EWAn_TFTP_TRIES, EWAn_BOOTP_TRIES, or EWAn_ARP_TRIES.

5.3.4.5 Different Ways of Booting Over the Internet

The following list shows the priority of the different ways of booting an initialized system over the Internet:

1. Specify the file name of the image to be booted and a network boot device in the **boot** command line. For example:

   ```
   >>> boot -file filename ewa0
   ```

   If the pathname for the file includes slashes (/), specify each slash as a double slash (//). For example:

   ```
   >>> boot -file //var//adm//ris//ris0.alpha//alphavme5 ewa0
   ```

2. Assign the file name of the image to be booted to the environment variable BOOT_FILE and then specify the network device in the **boot** command line. If the pathname for the file includes slashes (/), specify each slash as a double slash (//). For example:

   ```
   >>> set BOOT_FILE //var//adm//ris//ris0.alpha//alphavme5
   >>> boot ewa0
   ```

3. Assign the file name of the image to be booted to the environment variable EWA0_INETFILE and then specify the network device in the **boot** command line. For example:

   ```
   >>> set EWA0_INETFILE/var/adm/ris/ris0.alpha/alphavme5.exe
   >>> boot ewa0
   ```

   This method uses only the TFTP protocol. All other fields in the BOOTP packet must already be initialized with valid information from a previous Internet boot.

4. Assign the file name of the image to be booted to the environment variable EWA0_BOOTP_FILE and then specify the network device in the **boot** command line. For example:

   ```
   >>> set EWA0_BOOTP_FILE /var/adm/ris/ris0.alpha/alphavme5.exe
   >>> boot ewa0
   ```

   The file name defined by EWA0_BOOTP_FILE becomes the file name in the outgoing BOOTP request packet.

5. Do not define or specify an image to be booted. Just execute the **boot** command as follows:

   ```
   >>> boot ewa0
   ```

   With this method, because none of the environment variables are defined, the boot process runs through both the BOOTP and TFTP stages of an Internet network boot (see Section 5.3.4.1). Any server that receives the boot request replies.

Note

In the client-server paradigm, the way the firmware acts is affected by the software running on the server. Thus, the format of the file specification used with TFTP depends on the server. For example, if you are booting from a UNIX server, you must specify a complete pathname. See your operating system documentation for details about your server software.

5.3.5 Invoking the Console as Soon as the Boot Image is Loaded

Normally, when you boot an image, that image takes control of the system as soon as the image is loaded and the associated page tables and other data structures are set up. If you have a need to interact with the system after booting (for example, to debug the system or change environment variable settings), use the **boot** command's **-halt** option. This option forces the boot code to invoke the console program once the boot image is loaded and all associated page tables and data structures are set up.

Note

The **-halt** option does not shut down console device drivers.

5.4 Using TFTP to Read Files Across the Network

In addition to serving as a boot protocol, the TFTP driver provides a mechanism for reading files across the network. For example, you can use a TFTP specification when issuing the **cat** command to copy the contents of a remote file to standard output.

The syntax for a TFTP specification follows:

tftp:*n.n.n.n*:*pathname/network-interface*

The *n.n.n.n* represents an Internet address in dotted decimal notation. This must be the Internet address of the remote system from which you want to read the file. The colon (:) separates the Internet address from the pathname for the file to be read. If the pathname includes slash (/) characters, you must replace them with double slashes (//) in the specification. Specify *network-interface* as ewa*n*, where *n* identifies the interface. The following example displays the file /usr/foo/bar, which is on a remote system with address 16.123.16.242, using network interface ewa0:

```
>>> cat tftp:16.123.16.242://usr//foo//bar/ewa0
```

For convenience, you can save the Internet address in an environment variable. For example:

```
>>> set ktrose 16.123.16.242
>>> cat tftp:$ktrose://usr//foo//bar/ewa0
```

If you omit the address and file specification, TFTP uses the server address and file names defined by EWA*n*_DEF_SINETADDR and EWA*n*_DEF_INETFILE.

5.5 Managing the TOY Clock

The time-of-year (TOY) clock maintains the SBC's time, including the year, month, date, day, hour, minute, second, 1/10th of a second, and 1/100th of a second. Using console commands, you can:

- Display the clock's time and date
- Set the clock's time and date

- Disable the clock's internal oscillator

5.5.1 Displaying the TOY Clock's Time and Date

To display the TOY clock's time and date, use the **date** command without any arguments. For example:

```
>>> date
10:29:04  August   3, 1997
```

5.5.2 Setting the TOY Clock's Time and Date

If the internal oscillator for the TOY clock becomes disabled due to use of the **set toy sleep** command or another cause, you may need to reset the clock's time and date the next time you power up the system.

To set the time and date, issue the **date** command with a time specification of the form *yyymmddhhmm.ss*., which specifies:

Time Component...	As...	With a Value in the Range...
Year	*yyyy*	0000 to 9999
Month	*mm*	01 to 12
Day	*dd*	01 to 31
Hour	*hh*	00 to 23
Minute	*mm*	00 to 59
Second	*ss*	00 to 59

When you reset the time and date, you must specify at least four digits, which are interpreted as hours and minutes. If you specify six digits, the digits specify the day, hours, and minutes.

5.5.3 Disabling the TOY Clock's Internal Oscillator

If you are testing an Alpha VME 5/352 or 5/480 SBC TOY clock or if you are planning to put one of these SBCs in storage, you may want to use the **set toy sleep** command to disable the TOY clock's internal oscillator. Disabling the oscillator before storing the SBC can extend the shelf life of the oscillator's lithium battery. The oscillator is reenabled and the clock starts counting time again the next time you power up the SBC. Once the system is powered up again, you must reset the SBC's time and date.

Note

Alpha VME 5/352 and 5/480 SBCs are shipped with sleep mode enabled to conserve battery life.

5.6 Getting System Information

You can acquire information about your system by using the **show** command. You specify this command with an environment variable or a predefined keyword argument. When you specify an environment variable, the command displays the value of that variable. For example, the following command displays the default system power-up action as defined by the environment variable AUTO_ACTION:

```
>>> show auto_action
boot
>>>
```

For a complete listing of environment variables, see Section 5.2.1.

By specifying the **show** command with a keyword argument, you can display the following information on your console terminal:

Information	Keyword
The system configuration	**config**
Devices and controllers on the system	**device**
The Alpha HWRPB	**hwrpb**
The character illuminated on the system's LED	**LED**
A map of the system's virtual memory	**map**

The following example displays information about the devices that are known to the system:

```
>>> show device
dkb0.0.0.1.0   DKB0          RZ57
mke0.0.0.4.0   MKE0          TZ85
eza0.0.0.6.0   EZA0          08-00-2B-19-60-31
ezb0.0.0.7.0   EZB0          08-00-2B-1A-2C-06
p_a0.7.0.0.0                 Bus ID 7
p_c0.7.0.2.0                 Bus ID 7
pkb0.7.0.1.0   PKB0          SCSI Bus ID 7
pke0.7.0.4.0   PKE0          SCSI Bus ID 7
```

5.7 Updating Firmware

During the life of your SBC, you may receive one or more update kits for loading new firmware into the flash ROMs (FEPROMs). The documentation provided in the firmware update kit will guide you through the update procedure. A summary of the procedure follows:

1. Close DIP switch #2 on the I/O module to allow the update image to write to the FEPROM.
2. Issue the **boot** command.
3. Issue the **update** command.

   The **update** command loads the FEPROM update image from a specified device into system memory. Once the image is loaded, the console prompts for confirmation for the update to continue.

4. Respond to the confirmation prompt.

   If you respond with No, the update process terminates. If you respond with Yes, the update image erases, programs, and verifies the target FEPROMs.

   Note

   Once you commit to the update at this point, you must not interrupt program execution. Doing so may result in the SBC being placed in an inoperable state.

   The update image verifies each byte of the FEPROM. Each step provides for a certain number of retries to perform the operation successfully on a particular byte of the EPROM. If a failure occurs during any of the steps, the console displays an error message.

5. Reset or power the system off and on to run the new image in the FEPROMs.
6. Open DIP switch #2 on the I/O module to disable write operations to the FEPROM.

Using **update** command options, you can specify the name of the FEPROM update image, whether MOP or TFTP is to be used as the source transport protocol, the device from which the image is to be loaded (ewa0), and whether the console or user flash is to be upgraded.

For more information about firmware updates, see the documentation provided in your firmware update kit.

5.8 Examining and Depositing Data

If you need to manipulate data within an Alpha VME 5/352 or 5/480 SBC, you can do so by using the **examine** and **deposit** commands. These commands manipulate byte streams (extents of memory, sets of registers, physical devices, or files) and address spaces, which this discussion collectively refers to as devices.

5.8.1 The Default Device

Unless otherwise specified, the default device is physical memory. If you specify another device, that device becomes the default. A default device is *sticky*, in that all subsequent commands affect that device until you explicitly specify another device.

5.8.2 Console Device Drivers

The console uses drivers as the mechanism for referring to various devices and provides drivers for the following Alpha devices:

Device Name	Description
pmem	Physical memory
vmem	Virtual memory
gpr	General-purpose registers
fpr	Floating-point registers
ipr	Internal processor registers
pt	PAL temporary register set
pcicfg	PCI configuration space
pcidmem	PCI dense memory space
pcismem	PCI sparse memory space
pciio	PCI I/O space
eerom	Environment variable and error log NVRAM
ferom	Intel 28F020 firmware FEPROM
toy	DS1386 registers, clock chip, and NVRAM

You can direct the **examine** or **deposit** command towards a specific device by specifying the corresponding device name in the command line.

5.8.3 Device Byte Offsets

One of the arguments that you must specify with the **deposit** and **examine** commands is the address of the data to be examined or the address at which data is to be deposited. Because the Alpha VME 5/352 and 5/480 SBCs treat an address space as a device, the *address* argument that you specify becomes a byte offset.

For example, **pmem:0** refers to the location in physical memory at offset zero, that is, physical address 0. If you do not supply a device name, the offset applies to the last device referenced (physical memory by default). However, in the remaining discussions, the terms *address* and *offset* are used synonymously.

The **examine** and **deposit** commands act on a physical address. You can specify the actual address or use a symbol in Table 5–3 to point to the address.

Table 5–3 Symbols Used by Examine and Deposit Commands

Symbol	Description
+	Next address
*	Current address
–	Previous address

These symbols work because the console keeps track of the last referenced address. If you issue an **examine** or a **deposit** command without an address, the console firmware uses the next address. The console computes the next address as the last referenced address plus the current data size.

5.8.4 Specifying a Data Size

You have the option of explicitly specifying the size of the data to be examined or deposited by including one of the following options in the command line:

Option	Data Size
-b	Byte
-w	Word
-l	Longword
-q	Quadword
-o	Octaword
-h	Hexaword

5.8.5 Depositing and Examining Data in Memory

The steps for gaining access to and manipulating data in memory are as follows:

1. Find an unused block of memory.

   To find a block of memory, use the **alloc** command (see Section 5.10.3 for more information).

   Note

   Because the console itself and other critical data structures reside in memory, be careful not to alter them.

   The **alloc** command in the following example finds an unused 1000-byte block of memory:

   ```
   >>> alloc 1000
   03FFF000
   ```

   The address of the allocated block is, in this case, 0x03FFF000.

2. Add a value to physical memory.

   Use the **deposit** command to add a value to physical memory. The following command adds a value of 1:

   ```
   >>> deposit pmem:3fff000 1
   ```

3. Check the contents of the address.

   Use the **examine** command to check the contents of the address. For example:

   ```
   >>> examine pmem:3fff000
   pmem:          3FFF000 00000001
   ```

You can abbreviate commands and you do not need to specify the device if you are referring to the default device. The following example shows the **deposit** and **examine** commands in an abbreviated form. The current device is still physical memory.

```
>>> d 3fff000 abcdef12      # Deposit new data there.

>>> e 3fff000               # Check it out.

pmem:          3FFF000 ABCDEF12
```

You can also specify command options. The following example shows how to use the **–n** option to specify a repeat count. The command is executed over *n*+1 successive addresses.

```
>>> d 3fff000 aaaa5555 –n 3 # Write to 4 locations, yes 4!

>>> e 3fff000 –n 3          # Notice that –n 3 yields n+1 or 4!

pmem:          3FFF000 AAAA5555
pmem:          3FFF004 AAAA5555
pmem:          3FFF008 AAAA5555
pmem:          3FFF00C AAAA5555
```

An alternate method for examining memory (or other devices or files) is to use the hex dump command, **hd**. The **–l** option for that command specifies the number of bytes to display.

```
>>> hd pmem:3fff000 –l 10  # Dump the allocated memory.

00000000  55 55 aa aa 55 55 aa aa 55 55 aa aa 55 55 aa aa
UUªªUUªªUUªªUUªª

>>> hd –l 20 show_status   # Dump part of SHOW_STATUS script.

00000000  65 63 68 6f 20 27 64 2f 53 27 20 3e 24 24 73 73 echo 'd/S' >$$ss
00000010  0a 65 63 68 6f 20 27 2d 2d 2d 27 20 3e 3e 24 24 .echo '---' >>$$
```

Note

Both **–l** and **–n** give the same result, but **–l** works only with **hd** and **–n** works only with **examine**.

5.8.6 Depositing and Examining Data in Registers

You can use the **deposit** and **examine** commands to manipulate data in registers. To operate on a register, include the address of the register in the command line in one of the following ways:

- Symbolically, for example, **r0** or **ksp**
- Explicitly, as offsets within device address space, for example, **gpr:0** or **ipr:0**

You can also use the symbolic addresses **+**, *****, **–**, and the implied address increment (no address specified). The following examples show the different ways to include an address:

```
>>> e r0              # Examine R0 symbolically,...
```

```
gpr:                0 (    R0) 0000000000000002
>>> e gpr:0           #...explicitly as device offset,...
gpr:                0 (    R0) 0000000000000002
>>> e 0               #         ...or implicitly as device offset.
gpr:                0 (    R0) 0000000000000002
>>> e 8               # Examine R1...
gpr:                8 (    R1) 000000000000C408
>>> e                 #           ...and the next R2.
gpr:               10 (    R2) 0000000000000000
>>> e ipr:0           # Examine an IPR...
ipr:                0 (   ASN) 0000000000000000
>>> e                 #               ...and the next...
ipr:                1 ( ASTEN) 0000000000000000
>>> e +               #               ...and the next...
ipr:                2 ( ASTSR) 0000000000000000
>>> e *               #               ...and the current...
ipr:                2 ( ASTSR) 0000000000000000
>>> e –               #               ...and the previous one.
ipr:                1 ( ASTEN) 0000000000000000
>>> e ksp             # Examine an IPR by name...
ipr:               12 (   KSP) 0000000000000F30
>>> e                 # ...and the next one.
ipr:               13 (   ESP) 0000000000000000
```

The **examine** and **deposit** commands support symbolic representation of the following processor registers:

Register	Meaning
pc	Program counter
sp	Stack pointer
ps	Processor status longword
–	Previous address

```
>>> e pc              # Program Counter
PC psr:                0 (    PC) 0000000000000D30
>>> e ps              # Process Status
ipr:               17 (    PS) 0000000000001F00
>>> e sp              # Stack Pointer
gpr:               F0 (   R30) 0000000000000F30
```

5.9 Managing the Console, Devices, and CPU

Console commands are available for managing the console, devices, and CPU of an Alpha VME 5/352 or 5/480 SBC. Using console commands you can:

- Initialize the console, a device, or the CPU
- Stop the CPU or a specified device
- Exercise devices with read, write, and comparison operations

5.9.1 Initializing SBC Components

Use the **init** command to initialize your SBC's devices or CPU. To initialize a specific device, specify the command with the **-d** option and the name of the device to be initialized. For example to initialize the network interface ewa0, enter:

```
>>> init -d ewa0
```

If you need to initialize the processor, specify the **init** command without any options as follows:

```
>>> init
```

5.9.2 Stopping and Starting the CPU or Devices

If you need to stop and start an Alpha VME 5/352 or 5/480 SBC CPU or the system devices, you can use the **stop** and **start** commands. To stop the CPU, enter just the command name **stop** on the command line. You can then restart the CPU by specifying the **start** command with the address at which execution is to begin.

To stop and start one or more devices, specify the **stop** and **start** commands with the **-drivers** option and one of the following:

Option Parameter	Stops or Starts
Specific device name	The specified device
Device prefix (for example, ewa)	All devices of the specified class
None	All system devices

5.9.3 Exercising Devices

You can exercise your SBC's devices with various read, write, and comparison operations by using the **exer** command. This command also can report performance statistics.

5.9.3.1 Exercise Buffers

The **exer** command uses two buffers in the "memzone" heap of main memory to perform the exercise operations. The command:

- Reads from a device to a buffer
- Writes from a buffer to a device
- Compares the contents of the two buffers

Prior to initiating any I/O operations, the command initializes the buffers with data patterns. By default, the data pattern for each buffer consists of 0x5A in every byte. Alternatively, you can specify your own data patterns with the **-d1** and **-d2** options. These options take a postfix string argument. For each byte in a given buffer, starting with the first byte:

1. **exer** passes the postfix string to the **eval** command
2. **eval** evaluates the string and returns a value
3. **exer** writes the value to the buffer

Note

The **exer** command never reinitializes the buffers, even after completing one or more exercise passes.

5.9.3.2 Exercise Operations

The types of I/O operations that the exerciser performs include the following:

- Read to a specific buffer
- Write from a specific buffer
- Write from a specific buffer without a lock
- Compare the contents of the two buffers
- Seek to the file offset prior to the last read or write
- Seek to varying device locations, using the console firmware's random number generator, before performing read and write operations
- Sleep for a specified number of milliseconds

5.9.3.3 Tailoring the Exercises

You can tailor the behavior of the **exer** command by using options to specify the following:

- The address range to exercise
- The packet size (number of bytes) to be used in each I/O operation
- The number of passes to run
- The number of seconds to run
- The sequence of I/O operations to be performed

5.9.3.4 Seeking to Random Device Locations

You can instruct the exerciser to seek to random device locations prior to performing I/O operations. You specify this action by including the action string ? with the **exer** command's **-a** option. The exerciser achieves randomization by using the console firmware's random number generator, which uses a linear congruential generator to generate the random numbers. The LCG algorithm is not truly random, but it comes closest to meeting the needs of the **exer** command. Each time

the exerciser calls the random number generator, it returns a number from a specified range. If the range of numbers is a power of two, then each subsequent call to the random number generator is guaranteed to return a different number from the range until all possible numbers within the range have been chosen. If the range of numbers is not a power of two, the exerciser uses the console firmware's random number generator with an upper bound that is greater than the actual range size but is a power of two. Then the exerciser uses the range size to perform a modulus operation on the number that the random number generator returns, thereby ensuring that a random number is generated within the random range size.

5.9.3.5 Returning Error Codes On I/O Failures

If you want the **exer** command to return an error code immediately after a read, write, or comparison error, set the environment variable D_HARDERR to HALT. If an error occurs and D_HARDERR is set to CONTINUE or LOOP, subsequent operations specified by the action string option can occur except for comparisons. For example, if a read error occurs, a subsequent comparison is skipped since a read failure preceding a comparison guarantees that the comparison fails. If subsequent block I/O operations succeed, comparisons of those blocks occur.

5.10 Managing Memory

The console interface includes commands you can use to:

- Display the state of dynamic memory
- Display a map of the system's virtual memory
- Allocate and free blocks of memory
- Change the ownership of a block of memory
- Test memory

5.10.1 Displaying the State of Dynamic Memory

Display the state of your SBC's dynamic memory by using the **dynamic** command. By default, the command displays state information for two heaps: a private console heap and remaining memory heap. The state information listed for each heap (or zone) includes:

- Starting address
- Size
- Used blocks
- Used bytes
- Free blocks
- Free bytes
- Utilization
- High water mark

To display information about a specific heap of memory, specify the address of that heap with the **-z** option.

A number of other options are available for controlling the information that the command displays and the operations it performs. Depending on the options you specify, the command may:

- Perform consistency checking on the heap
- Repair corrupted heap by flooding free blocks
- Include block headers in the display output
- Display state information on a per process basis
- Perform a validation test on the heap
- Set the size of the total memory for the system
- Extend the size of the default memory zone by a specified number of bytes

5.10.2 Displaying the System's Virtual Memory Map

To display your SBC's virtual memory map, use the **show map** command. The virtual memory map is empty after console initialization. If the command generates an empty map, you can fill the page tables by issuing the command **boot -halt**.

5.10.3 Allocating and Freeing Blocks of Memory

To allocate and free blocks of memory, use the **alloc** and **free** commands. The arguments that you specify with these commands must be hexadecimal values. When you allocate a block of memory, you must specify at least the number of bytes to allocate. Other arguments allow you to specify the modulus and remainder to be used for computing the beginning address of the requested block of memory.

A **-flood** option lets you flood the block of allocated memory with zeros. If you want to allocate memory starting at a specific heap address (for example, an address displayed by the **dynamic** command), you can specify that address with the **-z** option.

The **free** command returns the memory identified by specified addresses to the appropriate heap.

5.10.4 Changing the Ownership of a Block of Memory

Your SBC identifies the owner of a block of memory by associating that block with a process identifier (PID). To change the ownership of blocks of memory, specify the **chown** command with the PID of the new owner process and the starting addresses of blocks of memory that process is to own.

To display a listing of PIDs, issue the **ps** command.

5.10.5 Testing Memory

The following tests are available for exercising memory:

- Graycode memory test
- March memory test

- Random memory test
- Victim block test

To run the tests, issue the **memtest** command. By default, this command runs all four tests, starting at the address of the first free space in the memory zone. You can run a subset of the available tests by specifying the tests of interest with the **-t** option.

Note

If you use **memtest** to test large sections of memory, it might take a while for testing to complete.

5.10.5.1 Specifying the Range of Addresses to be Tested

Using various options, you can specify the range of memory addresses that are to be tested. You identify an address range by specifying a starting address with the **-sa** option and either an ending address, length, or block size (for the random memory test only) with the **-ea**, **-l**, or **-bs** option. Block size equals the specified length for all tests except the random memory test. The default block size is 1892 bytes.

Specify the length or block size in bytes. If you specify the length of the address range, the ending address equals the starting address plus the length.

When you specify a starting address, **memtest** calls the `malloc` function to allocate the specified amount of memory plus 32 bytes, beginning at that starting address. The extra 32 bytes are reserved for `malloc` header information. Therefore, if you specify starting address 0xa00000 and a length of 0x100000, **memtest** allocates from address 0x9fffe0 through 0xb00000. Generally, this is transparent. However, it could be confusing if you begin two **memtest** processes simultaneously with one beginning at address 0xa00000 for length 0x100000 and the other at 0xb00000 for length 0x100000. This will result in the second **memtest** process displaying the following message:

```
"Unable to allocate memory of length 100000 at starting address b00000."
```

The second process should use the starting address 0xb00020.

5.10.6 Graycode Memory Test

The graycode memory test uses the following algorithm to test a specified section of memory:

data = (*x*>>1)^*x*

The variable *x* is an incremented value.

The test makes three passes over the memory being tested:

For Pass	The Test
1	Writes a data pattern that alternates graycode and inversed graycode to each longword. This causes all but one data bit to toggle between each longword write. For example, graycode(0)=0x00000000 while the inverse of graycode(1)=0xFFFFFFFE.
2	Reads the data at each location, verifies the data, and writes the inverse of the data. The test performs these operations one longword at a time to ensure that: • All data bits are written as a one and zero. • All but one data bit toggle between longword writes. • Address shorts are identified.
3	Reads and verifies each location.

You can instruct the graycode memory test to:

- Perform pass 1 only by specifying the fast mode option **-f**. When you use this option, the test detects ECC/EDC errors only.
- Increment through the memory being tested by a specified number of quadwords. For example, an increment of 1 tests every other quadword. Specify the increment with the **-i** option. This option is useful for testing the same physical address range on multiple CPUs.
- Serialize access to memory by setting up a memory barrier after each memory access. The memory barrier option, **-mb**, is available only if you are running the test in fast mode.

5.10.6.1 March Memory Test

The march memory test uses a marching 1s and 0s algorithm to test a specified section of memory. The default data patterns that the test uses are 0x55555555 and its inverse 0xAAAAAAAA. You can specify an alternative data pattern with the **memtest** command's **-d** option.

The march memory test makes three passes over the memory being tested:

For Pass	The Test
1	Writes the default or a specified data pattern to the specified memory location, starting at the specified starting address and repeating through the specified length.
2	Reads the data pattern that has been written to memory, starting at the specified starting address, and writes back the inverse. The test operates on the data pattern a longword at a time until it reaches the specified length.
3	Reads back the inverse of the data pattern, starting at the end of the memory region being tested, and writes back 0s. The test operates on the data pattern a longword at a time until it reaches the specified starting address.

5.10.6.2 Random Memory Test

The random memory test writes random data to random addresses using random data sizes, lengths, and alignments. The test gains access to every memory location in the specified range of addresses to be tested so long as the length does not exceed 8 MB. When the length exceeds 8 MB, the test applies a modulo function to the seed, which can result in some addresses being tested multiple times and others not being tested at all.

The random memory test proceeds as follows:

1. Gets an address index into the random number generator's LCG structure based on the length of the address range being tested.
2. Gets a data index based on a random data seed that you specify with the **memtest** command's **-rs** option and the size of the address range.
3. Calls the random number generator with the acquired address index and an initial address seed of 0 to get a random address.
4. Calls the random number generator with the acquired data index and the specified data seed to get the longword of data to be used during testing. The lower bit of the random data determines whether the test performs longword or quadword transactions. (Use of the lower bit speeds up the test by eliminating the need for another call to the random number generator.)
5. Stores the random data at the random address.
6. Flushes the data out to the Bcache.
7. Reads the data back into memory.
8. Compares the data that was written and read. In the case of quadword write and read operations, the test forms the quadword by shifting the longword of random data to the left by 32 and ORing it with the original data's complement.

Note

The run time of the random memory test can be noticeably longer than that of the other memory tests because the test requires two calls to the console firmware's random number generator every time the test writes data.

5.10.6.3 Victim Eject Memory Test

The victim eject memory test exercises memory using a specified block of data. By default, the test uses a block containing four longwords of 0xFs, four longwords of 0s, four longwords of 0xFs, and 4 longwords of 0s, in that order. You have the option of instructing the test to use a block of data that you set up prior to running the test. You specify the address of the block of data with the **memtest** command's **-ba** option.

The victim eject memory test proceeds as follows:

1. Writes the specified block of data to the specified starting address.

2. Adds 4 MB to the starting address.
3. Writes arbitrary data to the new resulting address. This causes the original data to be victimized to memory.
4. Reads data starting at the original starting address.
5. Verifies that the data is correct.
6. Increments the starting address by a block.
7. Repeats steps 1 through 6 for the remainder of the specified address range.

5.10.6.4 Specifying Other Test Options

Other memory test options are available for:

- Requesting that all specified memory be allocated and tested randomly
- Timing the memory tests
- Requesting that the tests use the specified memory without an allocation
- Allocating memory to be tested from the firmware heap
- Using a memory barrier after each memory access to serialize access to the memory
- Specifying a group name
- Specifying a soft error threshold

5.10.6.5 Running Multiple Memory Tests

You can start multiple memory tests running in the background by using the **memexer** command. Issue this command with an integer value indicating the number of test processes you want to start.

5.11 Performing Network Operations

The console interface's **net** command provides a way of initiating basic maintenance operations protocol (MOP) operations for a specified network port. The default port is ewa0. By using various command options, you can:

- Display the status of the network port, including the values of MOP counters
- Display the network port's Ethernet station address
- Reinitialize port drivers
- Initialize MOP counters
- Send a MOP request ID to a specified node
- Send an Ethernet loopback to a specified node
- Request a MOP loopback and specify the number of seconds to wait for the loopback messages
- Send a reboot request to a remote boot node
- Display the values of Ethernet port CSRs
- Enable and disable the extended design verification test (DVT) loop service

- Change the mode of the port device
- Specify a remote node address to be used for Ethernet loopbacks, MOP requests, and remote boot requests
- Broadcast a MOP load request for a specified file
- Set the version of MOP to be used

5.12 Setting Reboot to the SROM Mini-Console

Generally, when you power on or reboot your Alpha VME 5/352 or 5/480 SBC, the SBC enters console mode after the POST diagnostics complete. Under certain conditions it may be necessary for you to enter SROM Mini-Console mode instead. For example, you may want to do this to debug the PCI bus. While in SROM Mini-Console mode there is less activity on the bus and you do not have to be concerned with interrupts generated by other system devices.

To enter this mode, use the **set reboot srom** command. After issuing the command, the SBC enters SROM Mini-Console mode the next time you power on or reboot the system.

Note

If the I/O module's debug jumper is installed, the system displays the SROM Mini-Debugger prompt every time you power on the system. While in the SROM Mini-Debugger, you can start the SROM console by entering the **st** command and then entering address 0x8000 at the address prompt as follows:

```
SROM> st
a> 8000
```

5.13 Controlling the LED

The console commands **set led** and **show led** are available for you to display characters on the system's front panel LED and to check the current value being illuminated on the LED. When using **set led**, you specify the character you want displayed. You can also indicate that the character be displayed in bright mode by specifying the **-b** option. By default, characters are displayed in dim mode.

5.14 Running the Power-On Diagnostics Script

You can start the system's power-on self-test (POST) diagnostics from the console by entering the **pwrup** command. This command initializes the network environment variables, runs memory tests, and executes the contents of the NVRAM script.

For more information about the POST diagnostics, see Chapter 8.

5.15 Managing the Console Error Log

The console firmware logs console errors in an area of NVRAM. Using console commands, you can display the contents of and initialize the log.

5.15.1 Displaying the Contents of the Console Error Log

To display the contents of the console error log, use the **show_log** command. Options allow you to control whether the command displays information about a specified number of most recent errors (**-n**), all errors (**-all**), or new (**-new**) errors.

The command displays the following types of information associated with each error:

- Date and time of the error
- The diagnostic that was running at the time of the error
- The pass count
- The test number
- The failing point
- Error message text

5.15.2 Initializing the Console Error Log

At any time, you can clear and initialize the console error log by issuing the **clear_log** command. This command sets the entire log area to zero and resets all error logging pointers, counters, and initialization flags accordingly.

Prior to initializing the error log area, the command displays the following confirmation message:

```
Error Log data in NVRAM will be destroyed!!
Continue (y/n)?
```

If you prefer not to be prompted for confirmation, specify the command's **-nc** option.

5.16 Evaluating Expressions

The console firmware evaluates postfix expressions that you specify with the **eval** command. The expression must consist of two numeric operands and an operator, in that order. Valid operators include:

Operator	Meaning
+	Add the operands.
-	Subtract the second operand from the first.
*	Multiply the operands.
/	Divide the first operand by the second.

The default radix for operands and command output is decimal. Command options **-ib**, **-io**, **-id**, and **-ix** allow you to specify the radix as binary, octal, decimal, or hexadecimal, respectively. Similarly, the options **-b**, **-o**, **-d**, and **-x** specify the radix of the command's output.

5.17 Managing Console Processes

At any given time, you can have multiple console processes running on your Alpha VME 5/352 or 5/480 SBC. Each console process is a shell process that implements most of the functionality that is offered by the UNIX Bourne shell.

The console interface provides commands that help you manage your console processes. Commands are available for:

- Creating and exiting console processes
- Monitoring the status of system processes
- Setting the priority of a console process
- Specifying the CPU on which a console process can run
- Suspending the execution of a console process
- Stopping and deleting processes from the system
- Breaking from control loops
- Returning a failure status
- Displaying the semaphores known to the system

5.17.1 Creating and Exiting Console Processes

Create (spawn) new console processes by using the **sh** command. You can pass arguments to the new process and use options to control whether:

- Lines are to be displayed as they are read
- A command should be displayed just before being executed
- The contents of standard input (*stdin*) should be deleted when the process exits
- Lexical elements (tokens) should be displayed as they are recognized
- Rules should be displayed as they are executed
- The names of routines should be displayed as they are called

When you are ready to exit a console process, you can do so by using the **exit** command. You can specify a status value to be returned on exit. If you choose not to specify an exit status, the command returns the status of the last command executed.

5.17.2 Monitoring Processes

The console monitors all processes while they are executing. To see the status of all the processes, use the **ps** command. This command displays the following information for each console process in the system:

- Process identifier (PID)
- Address of the process control block (PCB)
- Process priority
- CPU time
- Processor affinity
- CPU
- Program running
- Process state

To see the status of a specific process, use the **grep** command with a pipe to filter the output, as shown at the end of the following example:

```
>>> ps               # Display complete process status.

ID       PCB     Pri CPU Time Affinity CPU  Program    State
-------- -------- - ----- -------- --- -------- -----------------
0000006c 001423a0 3     2 00000001 0         ps running
0000005c 00144b40 2 19253 00000001 0    memtest ready
0000005b 00147a60 2     9 00000001 0      sh_bg waiting on 00144B40
00000059 0014c060 2 21750 00000001 0    memtest ready
00000058 0014edc0 2     5 00000001 0      sh_bg waiting on 0014C060
00000056 00152860 2     3 00000001 0   exer_kid waiting on mscp_rsp
00000055 00153ae0 2     2 00000001 0       exer waiting on exer_tqe
00000054 00181580 2     6 00000001 0      sh_bg waiting on 00153AE0
0000004f 00154d60 5    38 ffffffff 0  pke0_poll waiting on tqe

       .
       .
       .
>>> ps | grep exer  # Check exer.
00000056 00152860 2         6 00000001 0     exer_kid waiting on
mscp_rsp
00000055 00153ae0 2         2 00000001 0         exer waiting on
exer_tqe
```

5.17.3 Setting the Priority of Processes

If the system is running multiple processes, you may find it necessary to set process priorities to ensure proper system operation. Set a process' priority by specifying the PID and a priority value with the **sp** command. Priority values range from 0 to 7 with 7 being the highest. To determine the PID of a process use the **ps** command.

5.17.4 Specifying the CPUs on Which a Process Can Run

If your application environment consists of multiple CPUs, you can specify an affinity mask that indicates on which CPUs a process can run. Bits 0 and 1 of the mask correspond to CPUs 0 and 1, respectively.

Suppose a process is in the ready state on CPU 0 and CPU 1 is idle. You might consider changing the CPU affinity so that the process can run on CPU 1. To do this, use the **sa** command. Specify the command with the PID of the process and a mask value. For example, to set the mask such that a process can execute on CPU 1, specify a mask value of 2.

5.17.5 Suspending Processes

You can suspend the execution of the current console process for a specified amount of time by using the **sleep** command. By default, the command suspends the process for one second. When the console process is suspended, another console process that is in the ready state can start executing.

If the default sleep time is insufficient, you can specify a different value and you can use the **-v** option to specify milliseconds.

5.17.6 Stopping Processes

To stop a process and delete it from the system, use the **kill** command. You must specify the process identifier (PID) for each process that is to be stopped. If you do not know the PID for a given process, acquire it by using the **ps** command.

The following example uses the **ps** command to acquire the PIDs for processes running **memtest**, stops and deletes the process that has PID 59, and then reissues the **ps** command to check whether the process associated with that PID was deleted.

```
>>> ps | grep memtest# Find a process to kill.

0000005c 00144b40 2     135733 00000001 0      memtest ready
00000059 0014c060 2     138258 00000001 0      memtest ready

>>> kill 59          # Kill one of the memtests.

>>> ps | grep memtest# Display our background tasks.

0000005c 00144b40 2     135733 00000001 0      memtest ready
```

5.17.7 Breaking from Control Loops

To break from a **for**, **while**, or **until** program loop, use the **break** command. This command exits the current console process and returns a status code. You can specify the status code that is to be returned. If you omit the status code, **break** returns the status of the last console command executed.

5.17.8 Returning a Failure Status

You can return a failure status from the console by using the **false** command.

5.18 Displaying Semaphores

To display information about all the semaphores known to the system, use the **semaphore** command. The command traverses the semaphore queue and for each known semaphore, displays the following:

- Name
- Value
- Address
- Address of the first waiting process

5.19 Managing Files and File Content

Several console commands are available for managing files and file content. You can:

- Display the contents of a file (standard output)
- Change the attributes of a file
- Dump the contents of a file
- List the files and inodes in the system
- Delete files from the system
- Sort the contents of a file
- Write text to a file (standard output)
- Search for expressions within files
- Copy a file from the standard input channel of the current process to the standard output channel of that process

For more information on performing these operations, see descriptions of the **cat**, **chmod**, **echo**, **grep**, **hd**, **line**, **ls**, **rm**, and **sort** commands in Chapter 6.

6

Console Command Reference

This chapter describes the DIGITAL Alpha VME 5/352 and 5/480 SBC console commands. The descriptions are ordered alphabetically by command name for quick reference. The command descriptions include the following information:

- Explanation of usage
- Syntax
- Arguments
- Options
- Examples
- Related commands

alloc – allocate a block of memory

Allocates a block of memory from heap. Once allocated, test routines can write to and read from the allocated memory. Only one routine can write to the memory at a time, but multiple routines can read from the memory simultaneously.

Syntax

alloc *size* [*modulus*] [*remainder*] [-flood]
[-z *heap_address*]

Arguments

size

Specifies the number of bytes of memory to be allocated. Specify the size as a hexadecimal value.

modulus

Specifies the modulus for the beginning address of the block of memory being allocated. Specify the modulus as a hexadecimal value.

remainder

Specifies the remainder to be used with the modulus for computing the beginning address of the block of memory being allocated. Specify the modulus as a hexadecimal value.

Options

–flood

Flood the allocated block of memory with zeros.

–z *heap_address*

Allocate memory from the memory zone that starts at the specified heap address. To view the starting addresses of the system's memory zones, use the **dynamic** command.

Example

```
>>> alloc 200

00FFFE00

>>> free fffe00
>>> set base ‘alloc 400‘
>>> show base

base                    00FFFC00

>>> memtest $base
>>> free $base
>>> clear base
```

See Also

dynamic, **free**

boot – boot the system

Initializes the processor, loads a program image from a boot device, and transfers control to that image.

Syntax

boot [-file *boot_file*] [-flags *longword ,...*]
[-protocols *enet_protocol*] [-halt] [*boot_device*]

Argument

boot_device

The path for a device or list of devices from which the console firmware is to boot the system. If you specify a list of devices, separate the device names with commas (,) and no spaces. For example:

```
>>> boot ewa0,dka0
```

The firmware tries to boot the system from each device in the list in order. When one of the devices boots successfully, control passes to the booted image.

Note

Place network devices at the end of a boot device list. This is necessary because network bootstraps only terminate if a fatal error occurs or an image is successfully loaded.

If you omit *boot_device*, the firmware uses a boot specification previously defined with an environment variable. For example, if you used the **set** command to associate the environment variable BOOTDEF_DEV with a boot device, the firmware will use that boot device as the default.

Note

When you specify a boot device with the **boot** command, that device specification overrides the current default boot device for the current boot request, but does not change the setting of the corresponding environment variable.

Options

–file boot_*file*

Load the specified boot file image into the system. If you do not specify this option, the console firmware loads a boot file previously associated with the environment variable BOOT_FILE.

–flags *longword, ...*

Use the specified longwords as additional boot information for the operating system. If you do not specify this option, the firmware loads flags previously associated with the environment variable BOOT_FLAGS or BOOTED_FLAGS.

Note

If you specify flags in the **boot** command line, the flags override the current default values for the current boot request, but do not change the setting of the corresponding environment variable.

–protocols *enet_protocol*

Use the specified Ethernet protocols for a network boot. Specify MOP for a DECnet MOP boot, BOOTP for a TCP/IP boot, or both. If you specify both, the firmware attempts to use each protocol to solicit a boot server. If you do not specify a protocol, the firmware uses the protocol previously associated with the environment variable, EWA0_PROTOCOLS.

–halt

Forces the boot operation to halt and invoke the console program once the boot image is loaded and page tables and other data structures are set up. Console device drivers are not shut down when you specify this option.

Examples

1. `>>> boot`

   The system tries to boot from a default boot device. If you have not set up a default boot device, the console program returns an error message.

2. `>>> boot ewa0`

   The system boots from the Ethernet port EWA0.

3. `>>> boot -file avme.sys ewa0`

   The system boots the file `avme.sys` from Ethernet port EWA0.

4. `>>> boot -fi //usr//local//bootfile//alphavme_v1_1-0 -protocol bootp ewa0`

   The system uses TCP/IP BOOTP to perform a network boot from Ethernet port EWA0.

5. `>>> boot -flags 0,1`

   The system boots from a previously defined default boot device with boot flag settings 0 and 1.

6. `>>> boot -halt dka0`

   The system boots from the SCSI disk, `dka0`, but remains in console mode.

See Also

set, **show**

break – break from a program loop

Breaks from a for, while, or until loop. The console firmware exits the current shell with a status or returns the status of the last command.

Syntax

break [*break_level*]

Argument

break_level

Specifies the status code to be returned by the shell.

Example

```
>>> for i in 1 2 3 4 5 ; do  echo $i ; break ; done
1
>>>
```

cat – copy files

Copies specified files to standard output. You also can use this command to copy or append one file to another by specifying I/O redirection.

Syntax

cat [-l *length*] [*file ...*]

Arguments

file ...

Specifies the names of one or more input files to be copied. If you do not specify a file on the command line, the command copies standard input to standard output.

Options

–l *length*

Copy the specified number of bytes of each input file. Specify the length as a decimal value.

Examples

1.
```
>>> echo > foo 'this is a test.'
>>> cat foo
this is a test.
>>>
```

Creates the file `foo` with the **echo** command, and then uses the **cat** command to send the contents of the file to the standard output.

2.
```
>>> cat -l 6 foo
this i
>>>
```

Sends the first six bytes of the file `foo` to the standard output.

See Also

echo, **ls**, **rm**

chmod – change file attributes

Changes the attributes of files or inodes. This command provides a subset of the capabilities of the equivalent UNIX command.

Syntax

chmod [[{- + =}{r w x b z}]...] *file*...

Arguments

file ...

Specifies the files or inodes for which the attributes are to be modified.

Options

-

Clear the specified attributes.

+

Set the specified attributes.

=

Set the specified attributes and clear all other attributes not included in the command line.

r

Set or clear the read attribute.

w

Set or clear the write attribute.

x

Set or clear the execute attribute.

b

Set or clear the binary attribute.

z

Set or clear the expand attribute.

Examples

1. `>>> chmod +x script`
   Sets the executable attribute for the file `script`.
2. `>>> chmod =r errlog`
   Sets the file `errlog` to read only and clears all other attributes.
3. `>>> chmod -w dk*`
   Makes all SCSI disks nonwriteable.

See Also

chown, **ls –l**

chown – change ownership of memory block

Changes the ownership of a memory block to a specified process.

Syntax

chown ***pid address ...***

Arguments

pid

Specifies the hexadecimal process identifier (PID) of the new owner process. To display the PIDs of the system's current processes, use the **ps** command.

address ...

Specifies the hexadecimal addresses of the memory blocks for which ownership is to be changed.

Example

```
>>> chown 'ps | grep idle | find 0' 'alloc 200'
```

Uses the **ps** command to display the system's current processes, pipes the output to the **grep** command to find an idle process, and then uses the **alloc** command to return the starting address of the first free block of 200 bytes.

See Also

alloc, **dynamic**, **ps**

clear – delete environment variable

Deletes an environment variable from the system's name space.

Note

Some environment variables, such as BOOTDEF_DEV, are permanent and cannot be deleted.

Syntax

clear ***envar***

Argument

envar

Specifies the name of the environment variable to be deleted.

Example

```
>>> clear foo
>>>
```

Deletes the environment variable `foo.`

See Also

set, **show**

clear_log – clear error log in NVRAM

Clears and initializes the area of NVRAM used for console error logging. The console firmware clears the entire area of NVRAM where fault information is stored and resets miscellaneous pointers, counters, and initialization flags used in the error logging process.

Notes

When you use **clear_log**, the current contents of the NVRAM error log area is destroyed and lost forever.

Console error logging is completely independent of the operating system's error logging.

Syntax

clear_log

Options

–nc

Do not prompt for confirmation before starting the clear operation. By default, the firmware prompts for confirmation before starting the clear operation. Specify the **–nc** option if you do not want the firmware to prompt you.

Example

```
>>> clear_log

Error Log data in NVRAM will be destroyed!!
Continue (y/n)?
y
Initializing NVRAM Error Log...
```

Prompts for confirmation to continue with the clear operation. When the user responds with y, the firmware clears and initializes the NVRAM error log.

See Also

show_log

date – display or change the date and time

Displays or changes the date and time stored in the system's time-of-year (TOY) clock.

Note

The date and time are not preserved if the TOY clock battery has been disabled with the **set toy sleep** command. The next time the system is powered on the firmware reenables the battery, but you may need to reinitialize the date and time.

The format of the date and time registers for the console is as described in the DS1386 specification, except that the year register contains the number of years 1858. This is done to retain compatibility with the openVMS and UNIX operating systems.

Syntax

date [[[[*yyyy*]*mm*]*dd*]*hhmm*[*.ss*]]

Arguments

yyyymmddhhmm.ss

Specifies the new date and time as follows:

Field	Meaning	Valid Range of Values
yyyy	Year	0000 to 9999
mm	Month	01 to 12
dd	Day	01 to 31
hh	Hour	00 to 23
mm	Minutes	00 to 59
ss	Seconds	00 to 59

If you omit this argument, the command displays the current date and time.

To modify the date or time, you must specify at least the hour and minute fields (four digits). If you include six digits, the command interprets the input as the day, hour, and minute fields. The command inherits values for fields that you omit from the specification.

Example

```
>>> date 199708031029.00
>>> date
10:29:04 August 3, 1997
>>>
```

deposit – write data to memory

Writes data to a memory location, register, device, or file.

After initialization, if you have not specified a data address or size, the default address space is physical memory, the default data size is a quadword, and the default address is zero.

You specify an address or device by concatenating the device name with the address, for example, `pmem:0`, and by specifying the size of the space to which to write.

If you do not specify an address, the data is written to the current address in the current data size (the last previously specified address and data size).

If you specify a conflicting device, address, or data size, the console ignores the command and issues an error.

Syntax

deposit [-b -w -l -q -o -h] [-physical -virtual -gpr -fpr -ipr] [-n *count*] [-s *step*] [device] *address data*

Arguments

device

Specifies the device name or address space to which the data is to be written. Specify one of the following:

Value	Description
pmem:	Physical memory.
vmem:	Virtual memory. The console firmware checks on accessibility and protection. If the access would not be allowed to a program running with the current program stack, the firmware issues an error message. If memory mapping is not enabled, virtual addresses are equal to physical addresses.
gpr:	General purpose register. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **r0**, **r1** through **r31**, **ai**, **ra**, **pv**, **fp**, **sp**, or **rz**.
fpr:	Floating-point register set. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **f0** through **f31**.
ipr:	Internal processor register set. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **ps**, **asn**, **asten**, **astsr**, **at**, **fen**, **apir**, **ipl**, **mces**, **pcbb**, **prbr**, **ptbr**, **scbb**, **sirr**, **sisr**, **tbchk**, **tbia**, **tbiap**, **tbis**, **esp**, **ssp**, **usp**, or **whami**.
pt:	PAL Temporary register set. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **PT:0** through **PT:31** or **PT0:** through **PT31:**.
pcicfg:	PCI configuration space.
pcidmem:	PCI dense memory space.
pcismem:	PCI sparse memory space.
pciio:	PCI I/O space.
eerom:	Environment variable and error log NVRAM.
ferom:	Intel 28F020 firmware FEPROM.
toy:	DS1386 registers, clock chip, and NVRAM.

address

Specifies the address to which the data is to be written. The address can be:

- Any valid hexadecimal offset in the address space of the specified device
- A symbolic address (if you omit the *device* argument)

For hexadecimal addresses that start with "f", you must add a leading zero (0) to prevent recognition as a floating-point register. For example, 0f0 is a valid memory address while f0 is not.

If you do not specify the *device* argument, you can specify one of the following symbolic addresses:

Value	Description
gpr	General purpose register 0.
fpr	Floating-point register 1.
ipr	Internal processor register.
pt or **pt0** -**pt31**	PAL Temporary registers 0 through 31. The data size defaults to quadword; the address space defaults to **pt**.
PC	Program counter (execution address register). The last address, size, and type are unchanged.
+	The location immediately following the last location referred to by the **examine** or **deposit** command. For references to physical or virtual memory, the location is the last address plus the size of the last reference. For other address spaces, the address is the last address referred to plus one.
–	The location immediately preceding the last location referred to by the **examine** or **deposit** command. For references to physical or virtual memory, the location is the last address minus the size of the last reference. For other address spaces, the address is the last address referred to minus one.
*****	The location last referred to by the **examine** or **deposit** command.
@	Uses the data at the last address referred to by the **examine** or **deposit** command.

data

The data to be written. If the specified data is larger than the specified size, the console firmware ignores the command and issues an error. If the data is smaller than the specified size, the firmware pads the data with leading zeros before writing it.

Options

–b

Use a data size of byte.

–w

Use a data size of word.

–l

Use a data size of longword.

–q

Use a data size of quadword.

–o

Use a data size of octaword.

–h

Use a data size of hexaword.

–physical

Write the data to physical memory. Using this option is the same as specifying **pmem:** for *device*.

–virtual

Write the data to virtual memory. Using this option is the same as specifying **vmem:** for *device*.

–gpr

Write the data to the general purpose registers. Using this option is the same as specifying **gpr:** for *device*.

–fpr

Write the data to the floating-point registers. Using this option is the same as specifying **fpr:** for *device*.

–ipr

Write the data to the internal processor registers. Using this option is the same as specifying **ipr:** for *device*.

–n *count*

Write to the specified number of consecutive locations. The console firmware deposits to the first address, then to the specified number of succeeding addresses. Specify *count* as a hexadecimal value.

–s *step*

Increment the address location by the specified size. By default, the address increment size is the data size. Use this option to override the default. This option is not inherited. Specify *step* as a hexadecimal value.

Examples

1. `>>> d -b -n 1FF pmem:0 0`
   Clears the first 512 bytes of physical memory.
2. `>>> d -l -n 3 vmem:1234 5`
   Deposits 5 into four longwords starting at virtual memory address 1234.
3. `>>> d -n 8 R0 FFFFFFFF`

Loads general purpose registers R0 through R8 with –1.

4. `>>> d -l -n 10 -s 200 pmem:0 8`
   Deposits 8 into the first longword of each of the first 17 pages in physical memory.

See Also

examine

dynamic – show memory

Shows the state of dynamic memory. Dynamic memory is split into two main heaps: the console's private heap and the remaining memory heap.

Syntax

dynamic [-c [-r]] [-h] [-p] [-v] [-extend *byte_count*]
[-z *heap_address*]

Options

–c

Perform a consistency check on the default heap or the heap specified with option **–z**.

–r

Repair a broken heap by flooding free blocks with DYN$K_FLOOD_FREE if and only if the free blocks have been corrupted. Repairing broken heaps is dangerous at best, as it masks underlying errors. This flag takes effect only if a consistency check is being done.

–h

Display the headers of the blocks in the default heap or the heap specified with option **–z**.

–p

Display dynamic memory statistics on a per process basis.

–v

Perform a validation test on the default heap or the heap specified with option **–z**.

–extend *byte_count*

Extend the default memory zone by the specified byte count at the expense of the main memory zone. The command assumes that the two memory zones are physically adjacent.

–z *heap_address*

Operate on the heap at the specified address.

Examples

```
1. >>> dynamic

zone     zone       used    used       free    free       utili-  high
address  size       blocks  bytes      blocks  bytes      zation  water
-------- ---------- ------- ---------- ------- ---------- ------- ------
---
00097740 1048576    389     358944     17      689664      34 %   371872
001D2B80 14805504   1       32         1       14805504     0 %   0


2. >>> dynamic -cv -z 97740

zone     zone       used    used       free    free       utili-  high
```

```
address  size       blocks  bytes      blocks  bytes      zation  water
-------- ---------- ------- ---------- ------- ---------- ------- ------
---
00097740 1048576    398     359520     17      689088      34 %
371872)


3. >>> dynamic -h
zone     zone       used    used       free    free       utili-  high
address  size       blocks  bytes      blocks  bytes      zation  water
-------- ---------- ------- ---------- ------- ---------- ------- ------
---
00097740 1048576    392     359136     17      689472      34 %   389280
a 00097740 000E1600_001E0600 000E1608_001BF628 00000000 00097740 32
f 000E1600 0017E600_00097740 00189E68_00097748 FFFFFFFF 000E1600 643072)
a 0017E600 001823C0_000E1600 001BF448_001B0D6C 00000023 0017E600 15808)
       .
       .
       .
>>>
```

See Also

alloc, **free**

echo – display text output

Sends a line of text that you enter on the command line to the standard output. By default, standard output is your console screen. The **echo** command separates arguments (words) in the line with blanks and adds a new line character to the end of the line.

Syntax

echo [-n] ***args*** **...**

Arguments

args ...

Specifies the character strings to be displayed.

The character strings can include pipes and I/O redirection. However, if you use them, enclose the characters strings within single quotes.

Options

–n

Suppress new lines characters from the output.

Examples

1.
```
>>> echo this is a test.
this is a test.
>>>
```
Sends a character string to your console screen.

2.
```
>>> echo -n this is a test.
this is a test.>>>
```
Sends a character string to your console screen, but with no new line separating the string from the next console prompt >>> .

3.
```
>>> echo 'this is a test' > foo
>>> cat foo
this is a test
>>>
```
Pipes a string to the file `foo`. Typing the contents of the file `foo` then shows the string.

4.
```
>>> echo > foo 'this is the simplest way
_>to create a long file. All characters will be echoed
_>to file foo until the closing single quote.'
>>> cat foo
this is the simplest way
to create a long file. All characters will be echoed
to file foo until the closing single quote.
>>>
```
Shows how you can use **echo** to create a file that is several lines long.

See Also

cat

eval – evaluate expression

Evaluates a postfix expression.

Syntax

eval [-ib -io -id -ix] [-b -o -d -x] ***operand1 operand2 operator***

Arguments

operand1

Specifies the first numeric value to be evaluated.

operand2

Specifies the second numeric value to be evaluated.

operator

Specifies one of the following:

Operator	Description
+	Adds the operands.
-	Subtracts *operand2* from *operand1*.
*	Multiplies the operands.
/	Divides *operand1* by *operand2.*

Options

–ib

Use the operands as binary values.

–io

Use the operands as octal values.

–id

Use the operands as decimal values.

–ix

Use the operands as hexadecimal values.

–b

Display the output as a binary value.

–o

Display the output as an octal value.

–d

Display the output as a decimal value.

–x

Display the output as a hexadecimal value.

Examples

1. `>>> **eval 5 10 +**`
   `15`

   Adds 5 and 10 and displays 15 as the result.

2. `>>> **eval -ix -d 5 10 +**`
   `21`

   Adds the hexadecimal values 0x5 and 0x10 and displays the result as decimal value 21.

examine – display memory data

Displays the content of a memory location, register, device, or file.

After initialization, if you have not specified a data address or size, the default address space is physical memory, the default data size is a quadword, and the default address is zero.

You specify an address or device by concatenating the device name with the address, for example, `pmem:0`, and by specifying the size of the data to be displayed.

If you do not specify an address, the data at the current address is displayed in the current data size (the last previously specified address and data size).

If you specify a conflicting device, address, or data size, the console ignores the command and issues an error.

The information that the command displays consists of the device name, the address (or offset within the device) in hexadecimal, and the examined data in hexadecimal.

The **examine** command uses the same options as the **deposit** command. Additionally, the **examine** command supports instruction decoding (see option **–d**), which disassembles instructions beginning at the current address.

Syntax

examine [-b -w -l -q -o -h -d] [physical -virtual -gpr -fpr -ipr] [-n *count*] [-s *step*] [*device*] *address*

Arguments

device

Specifies the device name or address space to access. The following devices are supported:

Value	Description
pmem:	Physical memory.
vmem:	Virtual memory. The console firmware checks on accessibility and protection. If the access would not be allowed to a program running with the current program stack, the firmware issues an error message. If memory mapping is not enabled, virtual addresses are equal to physical addresses.
gpr:	General purpose register. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **r0** through **r31**. The default data size is quadword.
fpr:	Floating-point register set. The data size defaults to quadword. When you specify this value, you can specify the following symbols for *address*: **f0** through **f31**. The default data size is quadword.
ipr:	Internal processor register set.
pt:	PAL Temporary register set. The data size defaults to quadword.
pcicfg:	PCI configuration space.
pcidmem:	PCI dense memory space.
pcismem:	PCI sparse memory space.
pciio:	PCI I/O space.
eerom:	Environment variable and error log NVRAM.
ferom:	Intel 28F020 firmware FEPROM.
toy:	DS1386 registers, clock chip, and NVRAM.

address

Specifies the address of the data that is to be examined. The address can be:

- Any valid hexadecimal offset in the address space of the specified device
- A symbolic address (if you omit the *device* argument)

For hexadecimal addresses that start with “f,” you must add a leading zero (0) to prevent recognition as a floating-point register. For example, 0f0 is a valid memory address while f0 is not.

If you do not specify the *device* argument, you can specify one of the following symbolic addresses:

Value	Description
gpr- *name*	Names a general purpose register. The data size defaults to quadword and the address space defaults to **gpr**. Symbols you can specify as valid names include **r0** through **r31**, **ai**, **ra**, **pv**, **fp**, **sp**, and **rz**.
fpr- *name*	Names a floating-point register. The data size defaults to quadword and the address space defaults to **fpr**. Symbols you can specify as valid names include **f0** through **f31**.
ipr- *name*	Names an internal processor register. The data size defaults to quadword and the address space defaults to **ipr**. Symbols you can specify as valid names include **ps**, **asn**, **asten**, **astsr**, **at**, **fen**, **ipir**, **ipl**, **mces**, **pcbb**, **prbr**, **ptbr**, **scbb**, **sirr**, **sisr**, **tbchk**, **tbia**, **tbiap**, **tbis**, **esp**, **ssp**, **usp**, and **whami**.
pt- *name*	Names a PAL Temporary register. The data size defaults to quadword and the address space defaults to **pt**. Symbols you can specify as valid names include **pt0** through **pt31**.
PC	Names the program counter (execution address register). The last address, size, and type are unchanged.
+	Names the location immediately following the last location referred to by the **examine** or **deposit** command. For references to physical or virtual memory, the location is the last address plus the size of the last reference. For other address spaces, the address is the last address referred to plus one.
–	Names the location immediately preceding the last location referred to by the **examine** or **deposit** command. For references to physical or virtual memory, the location is the last address minus the size of the last reference. For other address spaces, the address is the last address referred to minus one.
*****	Names the location last referred to by the **examine** or **deposit** command.
@	Uses the data at the last address referred to by the **examine** or **deposit** command as the address.

Options

–b

Use a data size of byte.

–w

Use a data size of word.

–l

Use a data size of longword.

–q

Use a data size of quadword.

–o

Use a data size of octaword.

–h

Use a data size of hexaword.

–d

Display the decoded macro instruction. This option does not recognize machine-specific PAL instructions.

–physical

Display data that is at an address in physical memory. Using this option is the same as specifying **pmem:** for *device*.

–virtual

Display data that is at an address in virtual memory. Using this option is the same as specifying **vmem:** for *device*.

–gpr

Display data that is in the general purpose registers. Using this option is the same as specifying **gpr:** for *device*.

–fpr

Display data that is in the floating-point registers. Using this option is the same as specifying **fpr:** for *device*.

–ipr

Display data that is in the internal processor registers. Using this option is the same as specifying **ipr:** for *device*.

–n *count*

Display the data at the specified number of consecutive locations.

–s *step*

Increment the address location by the specified size. By default, the address increment size is the data size. Use this option to override the default. This option is not inherited. Specify *step* as a hexadecimal value.

1.
```
>>> e r0
gpr:                0 (    R0) 0000000000000002
```
Examine general purpose register R0 by symbolic address.

2.
```
>>> e -g 0
gpr:                0 (    R0) 0000000000000002
```
Examine general purpose register R0 by address space (–gpr option).

3.
```
>>> e gpr:0
gpr:                0 (    R0) 0000000000000002
```
Examine R0 by device name.

4.
```
>>> examine pc
gpr: 0000000F (    PC) FFFFFFFC
```
Examine the program counter.

5.
```
>>> examine sp
gpr: 0000000E (    SP) 00000200
```

Examine the GPR stack pointer register.

6.
```
>>> examine -n 5 R7
gpr: 00000007 (    R7) 00000000
gpr: 00000008 (    R8) 00000000
gpr: 00000009 (    R9) 801D9000
gpr: 0000000A (   R10) 00000000
gpr: 0000000B (   R11) 00000000
gpr: 0000000C (    AP) 00000000
```
Examine register R7 plus the 5 following general purpose registers.

7.
```
>>> examine ipr:11
ipr: 00000011 (  SCBB) 2004A000
```
Examine the SCBB, internal processor register 17 (decimal).

8.
```
>>> examine scbb
ipr: 00000011 (  SCBB) 2004A000
```
Examine the SCBB using the symbolic name.

9.
```
>>> examine pmem:0
pmem: 00000000 00000000
```
Examine physical address 0.

10.
```
>>> examine -d 40000
pmem: 00040000      11 BRB     20040019
```
Examine address 40000 with macro instruction decode.

11.
```
>>> examine
pmem: 20040048   DB MFPR    S^#2B,B^48(R1)
```
Look at the next instruction.

See Also

deposit

exer – exercise devices

Exercises one or more devices by performing read, write, and comparison operations. Optionally, this command reports performance statistics.

A read operation reads data from a device and places the data in a buffer. A write operation writes data that resides in a buffer to a device. A comparison operation compares the contents of two buffers.

The **exer** command uses two buffers in "memzone" heap of main memory, *buffer1* and *buffer2*. A read or write operation can use either buffer. A compare operation uses both buffers.

The total number of bytes read or written on each pass of the exerciser is specified by the length (in blocks) or starting and ending block address options.

Syntax

exer [-sb *start_block*] [-eb *end_block*] [-p *pass_count*]
[-l *blocks*] [-bs *block_size*] [-bc *block_per_io*]
[-d1 *buf1_string*] [-d2 *buf2_string*] [-a *action_string*]
[-sec *seconds*] [-m] [-v] [-delay *milliseconds*]
[*device...*]

Arguments

device...

Specifies the names of one or more devices or file streams to be exercised.

Options

–sb *start_block*

Use the specified hexadecimal value as the starting block number within the file stream. The default is 0.

–eb *end_block*

Use the specified hexadecimal value as the ending block number within the file stream. The default is 0.

–p *pass_count*

Run the exerciser for the specified number of passes. If you specify 0, the exerciser runs forever or until you enter Ctrl/C. The default is 1.

–l *blocks*

Exercise the specified number of blocks. Specify the block value as hexadecimal. This option has precedence over the **–eb** option. If the exerciser is reading only, and you do not specify **–l** or **–eb**, the exerciser reads until it reaches the end-of-file (EOF). If the exerciser is writing, and you do not specify **–l** or **–eb**, the exerciser writes for the size of the device. The default is 1.

–bs *block_size*

Use the specified block size. Specify the block size in bytes as a hexadecimal value. The default is 0x200 except for tape drives, which default to 0x800. The maximum block size allowed with variable length block reads is 0x800 bytes.

–bc *block_per_io*

Use the specified number of blocks per I/O operation. Specify the number of blocks as a hexadecimal value. The default is 1.

–d1 *buf1_string*

Evaluate the specified character string and initialize *buffer1* with the results. By default, the console firmware loads the buffer with alternating 5s and As (hexadecimal).

–d2 *buf2_string*

Evaluate the specified character string and initialize *buffer2* with the results. By default, the console firmware loads the buffer with alternating 5s and As (hexadecimal).

–a *action_string*

Use the specified "action string," which determines the sequence of read, write, and comparison operations that are to be performed on various buffers. The console firmware processes each command code character in the action string from left to right. Each time the **exer** command completes all of the operations specified by the action string, the command reduces the remaining amount of device data to be processed by the size of the last packet processed by the action string. The **exer** command processes the action string repeatedly until the specified amount of device data has been processed.

Lowercase action string characters specify operations that use *buffer1*. Uppercase action string characters specify operations that use *buffer2*. The action string character c requires the use of both buffers. The action string characters "?" and "-" do not use a buffer.

Table 6–1 lists the action string characters and corresponding actions. The default action string is "?r.":

Table 6–1 Action String Characters

Character	Action
r	Read data from a device and place the data in *buffer1.*
w	Write data that is in *buffer1* to a device.
R	Read data from a device and place the data in *buffer2.*
W	Write data that is in *buffer2* to a device.
n	Write data that is in *buffer1* without using locking to maintain mutual exclusion.
N	Write data that is in buffer2 without using locking to maintain mutual exclusion.
c	Compare the contents of *buffer1* and *buffer2.*

Table 6–1 Action String Characters (Continued)

Character	Action
-	Seek to a file offset prior to performing the last read or write operation.
?	Seek to a random block offset within a specified range of blocks, call the `random` function to create each of a set of numbers once, and then choose a set that is a power of two and is greater than or equal to the block range. Each call to `random` results in a number that is then mapped to the set of numbers in the block range. The **exer** command seeks to that location in the file stream. Since the **exer** command starts with the same random number seed, the set of random numbers generated is always over the same set of block range numbers.
s	Sleep for the number of milliseconds specified by the **delay** option. If you do not specify the **delay** option, the console sleeps for 1 millisecond. **Note:** Times reported in verbose mode are not necessarily accurate when this action character is used.

The action string can specify any combination or sequence of read, write, and comparison operations on buffer1 and buffer2. Depending on the option arguments that you use, you can omit one or two of the three operations without affecting the execution of the other operations.

If the **exer** command writes to a file, the number of bytes processed per pass equals the allocation size of the file. The allocation size is usually larger than the length of the file for RAM disk files, but equal to the length for disk devices.

Note

Disk device I/O fails if the block size is not equal to 1 or a multiple of 512. Partial block read or write operations are not supported; therefore, a length that is not a multiple of the block size results in no errors, but the last partial block I/O operation on the data does not occur.

–sec *seconds*

Terminate the exercise after the specified number of seconds have elapsed. By default, the exerciser continues until the specified number of blocks or *`pass-count`* are processed.

–m

Use metrics mode and report throughput at the end of the exercise.

–v

Use verbose mode for read operations and write the data that is read to standard output (STDOUT). This option does not apply to write and comparison operations.

Delay processing by the specified number of milliseconds if "s" appears in the action string.

Examples

1. `>>> exer dk*.* -p 0 -secs 36000`

   Read all SCSI type disks for the entire length of each disk. Repeat this for 36000 seconds (10 hours). All disks are read concurrently. Each block read occurs at a random block number on each disk.

2. `>>> exer -l 2 dka0`

   Read block numbers 0 and 1 from device `dka0`.

3. `>>> exer -sb 1 -eb 3 -bc 4 -a 'w' -d1 '0x5a' dka0`

   Write 0x5as to every byte of blocks 1, 2, and 3. The packet size is bc * bs, 4 * 512, 2048 for all writes.

4.
```
>>> ls -l du*.* dk*.*
d**.* no such file
r---   dk                 0/0             0   dka0.0.0.0.0
>>> exer dk*.* -bc 10 -sec 20 -m -a 'r'
dka0.0.0.0.0 exer completed

packet                                    IOs           elapsed idle
size    IOs  bytes read  bytes written   /sec bytes/sec seconds secs
8192  3325    27238400              0    166   1360288     20    19
```

5. `>>> exer -eb 64 -bc 4 -a '?w-Rc' dka0`

   Perform a destructive write test on blocks 0 through 100 on disk `dka0`. The packet size is 2048 bytes. The action string specifies the following sequence of operations:

   a. Set the current block address to a random block number on the disk between 0 and 97. A 4-block packet, starting at block number 98, 99, or 100, will access blocks beyond the end of the length to be processed. Thus, 97 is the largest possible starting block address of a packet.

   b. Write from *buffer1*, which contains the previously read data, to the current block address.

   c. Set the current block address to what it was just prior to the previous write operation.

   d. From the current block address, read a packet into *buffer2*.

   e. Compare *buffer1* with *buffer2* and report any discrepancies.

   f. Repeat steps a through e until enough packets have been written to satisfy the length requirement of 101 blocks.

6. `>>> exer -a '?r-w-Rc' dka0`

   Perform a nondestructive write test with packet sizes of 512 bytes. The action string specifies the following sequence of operations:

   a. Set the current block address to a random block number on the disk.

   b. From the current block address on the disk, read a packet into *buffer1*.

   c. Set the current block address to the device address, where it was just before the previous read operation occurred.

d. Write a packet of 0x5as from *buffer1* to the current block address.

e. Set the current block address to what it was just prior to the previous write operation.

f. From the current block address on the disk, read a packet into *buffer2*.

g. Compare *buffer1* with *buffer2* and report any discrepancies.

h. Repeat the preceding steps until each block on the disk has been written once and read twice.

7.
```
>>> set myd 0
>>> exer -bs 1 -bc a -l a -a 'w' -d1 'myd myd ~ =' foo
>>> clear myd
>>> hd foo -l a
00000000  ff 00 ff 00 ff 00 ff 00 ff 00 ..........
```

Use an environment variable *myd* as a counter. Write 10 bytes of the pattern ff 00 ff 00... to RAM disk file `foo`, using a packet size of 10 bytes. Because the length specified is also 10 bytes, only one write occurs. Delete the environment variable *myd*.

The **hd**, hexadecimal dump, of `foo` shows the contents of `foo` after the **exer** command runs.

8.
```
>>> set myd 0
>>> exer -bs 1 -bc a -l a -a 'w' -d1 'myd myd 1 + =' foo
>>> hd foo -l a
00000000  01 02 03 04 05 06 07 08 09 0a ..........
```

Write a pattern of 01 02 03 ... 0a to file `foo`.

9.
```
>>> set myd 0
>>> exer -bs 1 -bc 4 -l a -a 'w' -d1 'myd myd 1 + =' foo -m
foo exer completed

packet                                    IOs           elapsed idle
size    IOs  bytes read  bytes written   /sec bytes/sec seconds secs
4     3           0             10   3001     10001      0     0
>>> hd foo
00000000  01 02 03 04 01 02 03 04 01 02 ..........
>>> show myd
myd                4
```

10.
```
>>> echo '0123456789abcdefghijklmnopqrstAB' -n > foo3
>>> exer -bs 1 -v -m foo3
b2lkfmp8jatsnA1gri54B69o3qdc7eh0foo3 exer completed

packet                                    IOs           elapsed idle
size    IOs  bytes read  bytes written   /sec bytes/sec seconds secs
1    32          32              0   5333      5333      0     0
```

See Also

memexer

exit – exit current shell process

Exits the current shell process with the specified status or returns the status of the last command executed.

Syntax

exit ***exit_value***

Argument

exit_value

Specifies the status code to be returned by the shell process.

Examples

1. `>>> exit`
   Exits and returns the status of the previously executed command.
2. `>>> exit 0`
   Exits with a success status.
3. `>>> test || exit`
   Runs `test` and exits if there is an error.

false – return a failure status

Returns a failure status.

Syntax

false

Example

```
>>> while false ; do echo foo; done
>>>
```

free – deallocate memory

Frees a block of memory that has been allocated from heap. The block is returned to the appropriate heap.

Syntax

free ***address...***

Argument

address ...

Specifies the addresses of blocks of memory that are to be returned to the heap. If you specify more than one address, separate the addresses with a space.

Example

```
>>> alloc 200
00FFFE00
>>> free fffe00
>>> free ‘alloc 10‘ ‘alloc 20‘ ‘alloc 30‘
>>>
```

See Also

alloc, **dynamic**

grep – search for regular expressions

Globally searches for regular expressions and displays any lines containing occurrences of those expressions. A regular expression is a shorthand way of specifying a wildcard type of string comparison. Since the **grep** command is line-oriented, it works only on ASCII files.

Syntax

grep [-c] [-i] [-n] [-v] {*expression* -f *file*} [*file ...*]

Arguments

expression

Specifies the regular expression for which to search. If the expression includes any of the metacharacters listed in the following table, enclose the expression within quotes to avoid interpretation by the shell.

Metacharacter	Description
^	Matches the beginning of a line.
$	Matches the end of a line.
.	Matches any single character.
[ ]	Matches a specified set of characters, for example, `[ABC]` matches A or B or C. The following rules also apply for these sets: • A dash other than the first or last character denotes a range of characters: `[A-Z]` matches any uppercase letter. • If the first character of the set is ^, then the sense of match is reversed: `[^0-9]` matches any non-digit. • You must precede the backslash (\), right square bracket (]), dash (-), and circumflex (^) characters with a backslash (\ ) if they occur in a set.
*****	Matches repeatedly. When you place an asterisk (*) after a pattern, the asterisk indicates that the pattern should match any number of times. For example, `[a-z][0-9]*` matches a lowercase letter followed by zero or more digits.
+	Matches repeatedly. When you place a plus sign (+) after a pattern, the plus sign indicates that the pattern should match one or more times. For example, `[0-9]+` matches any sequence of one or more digits.
?	Matches optionally. When you place a question mark (?) after a pattern, the question mark indicates that the pattern can match zero or one times. For example, `[a-z][0-9]?` matches a lowercase letter alone or followed by a single digit.
\x'	Prevents the character (denoted by *x*) following the backslash from having special meaning.

file ...

Specifies the files to be searched. If you do not specify a file, the command searches standard input (STDIN).

Options

–c

Print only the number of lines that matched.

–i

Ignore case during the search. By default, the **grep** command is case-sensitive.

–n

Print the line numbers of the matching lines.

–v

Print all lines that do not contain the specified expression.

–f *file*

Use the regular expression in the specified file instead of the expression specified on the command line.

Examples

1.
```
>>> ps | grep ewa0
0000001f 0019e220 3    2 ffffffff 0  mopcn_ewa0 waiting on
mop_ewa0_cnw
00000019 0018e220 2    1 ffffffff 0  mopid_ewa0 waiting on tqe
00000018 0018f900 3    3 ffffffff 0  mopdl_ewa0 waiting on
mop_ewa0_dlw
00000015 0019c320 5    0 ffffffff 0     tx_ewa0 waiting on
ewa0_isr_tx
00000013 001a2ce0 5    2 ffffffff 0     rx_ewa0 waiting on
ewa0_isr_rx
```

   Search the output of the **ps** command (standard input) for lines containing EWA0.

2.
```
>>> alloc 20
00FFFFE0
>>> deposit -q pmem:fffff0 0
>>> e -n 3 ffffe0
pmem:           FFFFE0 EFEFEFEFEFEFEFEF
pmem:           FFFFE8 EFEFEFEFEFEFEFEF
pmem:           FFFFF0 0000000000000000
pmem:           FFFFF8 EFEFEFEFEFEFEFEF
>>> e -n 3 ffffe0 | grep -v 0000000000000000
pmem:           FFFFE0 EFEFEFEFEFEFEFEF)
pmem:           FFFFE8 EFEFEFEFEFEFEFEF)
pmem:           FFFFF8 EFEFEFEFEFEFEFEF)
>>> free ffffe0
>>>
```

   Using **grep**, search for all quadwords in a range of memory that are non-zero.

hd – dump file contents

Dumps the contents of a file in hexadecimal and ASCII format.

Syntax

hd [-byte -word -long -quad] ***file*** **...**

Arguments

file ...

Specifies the files to be displayed.

Options

–byte

Print the data in bytes.

–word

Print the data in words.

–long

Print the data in longwords.

–quad

Print the data in quadwords.

Examples

```
1. >>> echo -n 'the quick brown fox jumped over the lazy dog' >foo
   >>> hd foo
00000000 74 68 65 20 71 75 69 63 6B 20 62 72 6P 77 6E 20  the quick
brown
00000010 66 6F 78 20 6A 75 6D 70 65 64 20 6F 76 65 72 20  fox jumped
over
00000020 74 68 65 20 6C 61 7A 79 20 64 6F 67              the lazy dog

2. >>> -byte foo
00000000 74 68 65 20 71 75 69 63 6B 20 62 72 6P 77 6E 20  the quick
brown
00000010 66 6F 78 20 6A 75 6D 70 65 64 20 6F 76 65 72 20  fox jumped
over
00000020 74 68 65 20 6C 61 7A 79 20 64 6F 67              the lazy dog

3. >>> -word foo
   00000000 6874 2065 7571 6369 206B 7262 776F 206E  the quick brown
   00000010 6F66 2078 756A 706D 6465 6F20 6576 2072  fox jumped over
   00000020 6874 2065 616C 797A 6420 676F            the lazy dog

4. >>> -long foo
   00000000 20656874  63697571 7262206B  206E776F    the quick brown
   00000010 20786F66  706D756A 6F206465  20726576    fox jumped over
   00000020 20656874  797A616C 676F6420              the lazy dog

5. >>> -quad foo
   00000000 6369757120656874   206E776F7262206B      the quick brown
   00000010 706D756A20786F66   207265766F206465      fox jumped over
   00000020 797A616C20656874   00000000676F6420      the lazy dog
   >>>
```

help – display help on commands

Displays the syntax for Alpha VME 5/352 and 5/480 SBC console firmware commands. If you do not specify a command, the **help** command displays information about itself and lists the commands for which additional information is available.

The following conventions are used for the command syntax that the **help** command displays:

Convention	Description
<*item*>	Angle brackets denote an item for which you must specify a value.
[<*item*>]	Square brackets enclose optional arguments, options, or values.
{a, b, c }	Braces enclosing items separated by commas indicate mutually exclusive items. Choose only one of a, b, or c.
{a | b | c }	Braces enclosing items separated by vertical bars indicate combinatorial items. Choose any combination of a, b, and c.

You can use the **help** and **man** commands interchangeably.

Syntax

help [*command-spec ...*]

Arguments

command-spec ...

Specifies the commands or topics for which you request help.

For each command specification that you specify, the **help** command tries to find all topics that match. For example, if you specify a `ex` as the command specification, **help** displays information about the **exit** and **examine** commands.

The **help** command supports wildcards. Use an asterisk (*) as the wildcard character. For example, enter **help *** to display help on all console commands.

The **help** command treats command specifications as regular expressions. For more information on regular expressions, see the **grep** command. Help command specifications are case-sensitive.

Examples

1. `>>> help`
   Display a list of console commands for which help is available.
2. `>>> help *`
   Display help on all console commands.
3. `>>> help ex`
   Display help on all commands that begin with "`ex`."
4. `>>> help boot`
   Display help on the **boot** command.

init_ev – initialize environment variables

Sets all environment variables to default values. For the new variable settings to take effect, you must reset the system or issue the **initialize** command.

Syntax

init_ev

Example

```
>>> init_ev

Note: A System Reset or init command must be issued immediately
      after this command to set all environment variables to their
      default values!!

>>>
```

Reset the system or issue the **init** command to ensure that the new default environment variable settings take effect.

init – initialize a device or the processor and console

Initializes a device or the processor and console.

Syntax

init [-d *device*]

Option

–d *device*

Initialize the specified device.

Example

1. `>>>` **`init`**
   Initialize the processor and console.
2. `>>>` **`init -d ewa0`**
   Initialize device ewa0.

kill – delete process

Deletes the specified processes.

Syntax

kill *pid* ...

Arguments

pid ...

Specifies the process IDs (PIDs) of the processes to be deleted. To acquire a listing of PIDs associated with your system, use the **ps** command.

Example

```
>>> memtest -p 0 &
>>> ps | grep memtest
000000f1 00217920 2       9357 ffffffff 0      memtest ready
>>> kill f1
>>> ps | grep memtest
```

Run **memtest** and display the test's PID (f1) with the **ps** and **grep** commands. Using the displayed data, delete the process with the **kill** command. Try to display the test process again. The command output shows that the process is gone.

See Also

ps

line – read a line

Copies a line (up to the new line character) from the standard input channel of the current process to the standard output channel of that process. This command always writes at least the new line character as output.

Use this command in scripts to read from the user's terminal, or to read lines from a pipeline while in a **for/while/until** loop.

Syntax

line

Examples

1.
```
>>> line
type a line of input followed by carriage return
type a line of input followed by carriage return
```
Copy the line of typed input to the terminal screen.

2.
```
>>> line >foo
type a line of input followed by carriage return
>>> cat foo
type a line of input followed by carriage return
```
Use the **line** command interactively.

3.
```
>>> echo -n 'continue [Y, (N)]? '
>>> line <tt >tee:foo/nl
>>> if grep <foo '[yY]' >nl; then echo yes; else echo no; fi
>>>
```
Use the **line** command within a script.

ls – list files

Lists the files or inodes in the system. Inodes are RAM disk files, open channels, and some drivers. RAM disk files include script files, diagnostics, and executable shell commands.

Syntax

ls [-l] [*file* ...]

Argument

file ...

Specifies the files and inodes to be listed. You can use an asterisk (*) as a wildcard character. If you use a wildcard, the command lists all files and inodes that match the specification. If you omit the argument, the command lists all files and inodes on the system.

Option

–l

Lists the files and inodes in long format. When using long format, the command lists each file and inode on a line with additional information. By default, the command lists just names.

Examples

1.
```
>>> ls examine
examine
```
Lists the file named `examine`.

2.
```
>>> ls d*
d            date         debug1       debug2       decode
deposit
dg_pidlist   dka0.0.0.0.0              dke100.1.0.4.0
dub0.0.0.1.0              dynamic
```
Lists files and inodes that start with `d`.

man – help on commands

Displays the syntax for Alpha VME 5/352 and 5/480 SBC console firmware commands. If you do not specify a command, the **man** command displays information about itself and lists the commands for which additional information is available.

The following conventions are used for the command syntax that the **man** command displays:

Convention	Description
<*item*>	Angle brackets denote an item for which you must specify a value.
[<*item*>]	Square brackets enclose optional arguments, options, or values.
{a, b, c }	Braces enclosing items separated by commas indicate mutually exclusive items. Choose only one of a, b, or c.
{a | b | c }	Braces enclosing items separated by vertical bars indicate combinatorial items. Choose any combination of a, b, and c.

You can use the **help** and **man** commands interchangeably.

Syntax

man [*command-spec ...*]

Arguments

command-spec ...

Specifies the commands or topics for which you request help.

For each command specification that you specify, the **man** command tries to find all topics that match. For example, if you specify a `ex` as the command specification, **man** displays information about the **exit** and **examine** commands.

The **man** command supports wildcards. Use an asterisk (*) as the wildcard character. For example, enter **man *** to display help on all console commands.

The **man** command treats command specifications as regular expressions. For more information on regular expressions, see the **grep** command. Help command specifications are case sensitive.

Examples

1. `>>> man`
   Display a list of console commands for which help is available.
2. `>>> man *`
   Display help on all console commands.
3. `>>> man ex`
   Display help on all commands that begin with "`ex`."
4. `>>> man boot`
   Display help on the **boot** command.

memexer – memory exerciser

Starts a specified number of graycode memory test processes running in the background. Each test randomly allocates and tests blocks of memory twice the size of the Bcache, using all available memory.

The command does not display any output unless an error occurs.

Syntax

memexer [*number_of_tests*]

Argument

number_of_tests

Specifies the number of memory test processes to start. The default is 1. To run tests indefinitely, specify 0.

Example

```
>>> memexer 2 &
>>>
```

Run two memory tests in the background. The tests run in blocks of two times the backup cache size across all available memory.

See Also

memtest

memtest – memory test

Tests memory with any or all of four tests:

Test	Description
Graycode memory test	Writes, reads, and verifies a graycode pattern and an inverse graycode pattern for the specified address range.
March memory test	Writes, reads, and verifies a marching pattern and an inverse marching pattern for the specified address range.
Random memory test	Exercises random addresses within the specified range with random data of random length.
Victim block test	Writes blocks of data to the specified address, victimizes the data, and then reads back and verifies the block.

Notes

If you use **memtest** to test large sections of memory, it might take a while for testing to complete.

If you issue a Ctrl/C or the **kill** command with a PID in the middle of testing, the **memtest** process might not abort right away. To increase speed of execution, check for a Ctrl/C or **kill** command done outside of any test loops. If this is not satisfactory, you can run concurrent **memtest** processes in the background with shorter lengths within the target range.

Syntax

```
memtest [-sa start_address] [-ea end_address] [-l
length]
        [-ba block_address] [-bs block_size]
        [-i address_inc] [-p pass_count]
        [-d data_pattern] [-rs random_seed] [-rb] [-f] [-
m]
        [-z] [-h] [-mb] [-t] [-g] [-se]
```

Options

–sa *start_address*

Use the specified address as the starting address for the test. The default is the first free space in the memory zone.

–ea *end_address*

Use the specified address as the ending address for the test. The default is *start_address* plus *length*.

–l *length*

Test the specified length (in bytes) of memory. The default length is equal to *block_size*, except with the **–rb** option, which uses the zone size. The **–l** option has precedence over the **–ea** option.

–ba *block_address*

Test the block of memory at the specified address using the victim eject memory test. This option applies victim eject memory test only.

–bs *block_size*

Test the specified amount of memory (in bytes). Specify the size as a hexadecimal value. The default is 8192 bytes. This option applies to the random block test only. For all other tests, the block size equals *length*.

–i *address_inc*

Test memory at increments specified by the increment. The default is 0, which implies no incrementation. This option applies to the graycode test only. The increment value is in quadwords (that is, an increment of one tests every other quadword). This option is useful if multiple CPUs test the same physical memory.

To test an unaligned starting address, you must also specify the **–z** option.

–d *data_pattern*

Use the specified test pattern. The default pattern is 5s.

–p *pass_count*

Execute the test the number of times specified by the pass count. If you specify 0, the command runs forever or until you enter Ctrl/C. The default is 1.

–rs *random_seed*

Use the specified random seed. Use this option only with the **–rb** option. The default is 0.

–rb

Randomly allocate and test all of the specified memory address range. Allocations are done of size *block_size*.

–f

Use fast mode. If you specify this option, the data comparison is omitted. The console firmware detects only ECC/EDC errors.

–m

Time the memory test and at the end of the test, display the elapsed time. By default, the timer is off.

–z

Use the specified memory address without an allocation. This bypasses all checking, but allows testing in addresses outside of the main memory heap. It also allows unaligned testing.

Caution

This flag allows you to test and corrupt *any* memory.

–h

Allocate the memory to be tested from the firmware heap.

–mb

Use memory barriers after each memory access. Use this option only for fast mode (**-f**) graycode tests. When you specify this flag, the console firmware executes an Alpha MB instruction after every memory access. This guarantees serial access to memory.

–t

Run the specified tests. By default, the command runs all tests in the group specified by the **-g** option. The individual tests are as follows:

Test	Test Number
Graycode test	1
March test	2
Random test	3
Victim eject test	4

–g

Use the specified group. Currently, the only group supported is MFG.

–se

Use a soft error threshold.

Examples

1. `>>> memtest -sa 200000 -l 1000`
   Test memory starting at address 0x200000 (–sa) for 0x1000 bytes (–l).
2. `>>> memtest -sa 200000 -l 1000 -f`
   Test memory starting at address 0x200000 for 0x1000 bytes, using fast mode. Fast mode eliminates data verification.
3. `>>> memtest -sa 300000 -p 10`
   Write a default block size of 8192 bytes starting at address 0x300000 for 10 passes (–p).
4. `>>> memtest -f -mb`
   Test memory in arbitrary 8192 byte blocks, without data verification. After each read and write operation, execute a memory barrier (MB) instruction.
5. `>>> memtest -sa 200000 -ea 400000 -rb`
   Test memory starting at address 0x200000 and ending at address 0x3fffff. Randomly allocate every block within this range.

Note

The **memtest** command does not generate an error with the **–rb** option if a block within the range cannot be allocated.

6. `>>> memtest -h -rb -bs 100`
   Test the console heap by randomly allocating memory in blocks of size 0x100 bytes.
7. `>>> memtest -rb -p 0`
   Test memory across all of the memory zone (all memory excluding the HWRPB, the PAL area, the console, and the console heap). The test runs in the foreground until you enter Ctrl/C.

See Also

memexer

net – perform MOP operations

Using a specified Ethernet port, performs basic maintenance operations protocol (MOP) operations, such as loopbacks, ID requests, and remote file loads. This command also allows you to observe the status of a network port. Specifically, when you use **net** with the **–s** option, the command displays the current status of a port, including the contents of MOP counters. This command is useful for monitoring port activity and trying to isolate network failures.

Syntax

```
net [-s] [-sa] [-ri] [-ic] [-id] [-l0] [-l1] [-rb] [-csr]
    [-els] [-kls] [-cm mode] [-da node_address]
    [-l file_name] [-lw wait_in_secs] [-sv mop_version]
   port
```

Arguments

port

Specifies the name of the Ethernet port on which to operate. If you do not specify a port, the command uses the default port, EWA0.

Options

–s

Display port status information, including MOP counters.

–sa

Display the port's Ethernet station address.

–ri

Reinitialize the port's drivers.

–ic

Initialize the MOP counters.

–id

Send a MOP request ID to the destination node specified with the **–da** option.

–l0

Send an Ethernet loopback to the destination node specified with the **–da** option. This option, **–l0**, is "l" for loopback and zero.

–l1

Request a MOP loopback.

–rb

Request a system reboot by sending a MOP V4 request boot message to the remote boot node specified with the **–da** option.

–csr

Display the values of the Ethernet port control/status registers (CSRs).

–els

Enable the extended design verification test (DVT) loop service.

–kls

Disable the extended DVT loop service.–cm *mode*

Change the mode of the port device. Valid modes and their corresponding values include the following:

Mode	Symbol
Normal	**nm**
Internal loopback	**in**
External loopback	**ex**
Normal filter	**nf**
Promiscuous	**pr**
Multicast	**mc**
Internal loopback and promiscuous	**ip**
Force collisions	**fc**
No force collisions	**nofc**
Default	**df**

–da *node_address*

Use the specified destination node address with the **–l0**, **–id**, or **–rb** option.

–l *file_name*

Broadcast a MOP load request for the specified load file.

–lw *wait_in_secs*

Wait the specified number of seconds for loop messages from the **–l1** option to return. If the messages do not return in the specified time period, the console firmware generates an error message.

–sv *mop_version*

Set the preferred version of MOP to be used. Valid version numbers are 3 and 4.

Examples

1.
```
>>> net -sa
-ewa0:  08-00-2b-1d-02-91
```
Display the local Ethernet port station address.

2. >>> **net -s**

```
DEVICE SPECIFIC:
TI: 203 RI: 42237 RU: 4 ME: 0 TW: 0 RW: 0 BO: 0
HF: 0 UF: 0 TN: 0 LE: 0 TO: 0 RWT: 39967 RHF: 39969  TC: 54

PORT INFO:
tx full: 0 tx index in: 10 tx index out: 10
rx index in: 11

MOP BLOCK:
Network list size: 0

MOP COUNTERS:
Time since zeroed (Secs): 2815

TX:
Bytes: 116588 Frames: 204
Deferred: 2 One collision: 52 Multi collisions: 14
TX Failures:
Excessive collisions: 0 Carrier check: 0 Short circuit: 0
Open circuit: 0 Long frame: 0 Remote defer: 0
Collision detect: 0

RX:
Bytes: 116564 Frames: 194
Multicast bytes: 13850637 Multicast frames: 42343
RX Failures:
Block check: 0 Framing error: 0 Long frame: 0
Unknown destination: 42343 Data overrun: 0 No system buffer: 22
No user buffers: 0
>>>
```

Display the EWA0 port status, including the MOP counters.

ps – show process

Displays the system state in the form of process status and statistics.

Syntax

ps

Example

```
>>> ps
                    CPU
ID       PCB      Pri Time   Affinity CPU Program   State
-------- -------- --- ------ -------- --- --------- --------------------
--
0000008f 0010e8a0 3        0 00000001 0          ps running
00000020 00110160 1        0 ffffffff 0    puc_poll waiting on tqe
0000001f 0013cb60 6        0 ffffffff 0 puc_receive waiting on
puu_receive
0000001c 0013ed00 1        0 ffffffff 0    pub_poll waiting on tqe
0000001b 0014fc00 6        0 ffffffff 0 pub_receive waiting on
puu_receive
0000001a 00111a20 3        0 00000001 0          sh ready
00000015 001176a0 2        0 ffffffff 0  mopcn_ewa0 waiting on
mop_ewa0_cnw
00000014 00119140 2        0 ffffffff 0  mopid_ewa0 waiting on tqe
00000013 0011ac20 2        0 ffffffff 0  mopdl_ewa0 waiting on
mop_ewa0_dlw
00000012 0011f6a0 6        0 ffffffff 0     tx_ewa0 waiting on
ewa0_isr_tx
00000011 00121140 6        0 ffffffff 0     rx_ewa0 waiting on
ewa0_isr_rx
00000010 00122ac0 1        0 ffffffff 0    pua_poll waiting on tqe
0000000f 001244e0 6        0 ffffffff 0 pua_receive waiting on
pua_receive
00000009 00147460 5        0 ffffffff 0    lad_poll waiting on tqe
00000008 00148f00 5        0 ffffffff 0    dup_poll waiting on tqe
00000007 0014a9a0 5        0 ffffffff 0   mscp_poll waiting on tqe
00000006 0014e1a0 5        0 00000001 0    entry_00 waiting on entry_00
00000004 001516e0 2        0 ffffffff 0  dead_eater waiting on dead_pcb
00000003 00153140 7 11759330 ffffffff 0       timer waiting on timer
00000002 00158740 6        0 ffffffff 0  tt_control waiting on
tt_control
00000001 0005cfd8 0        0 00000001 0        idle ready
>>>
```

See Also

sa, **sp**

pwrup – run power-on diagnostics

Runs the power-on diagnostics script. The **pwrup** command initializes network environment variables and runs diagnostic tests.

Syntax

```
pwrup
```

Example

```
>>> pwrup
```

Runs the power-on script.

rm – remove file

Removes specified files from the file system. Allocated memory associated with the removed files is returned to memory heap.

Syntax

```
rm file...
```

Arguments

file ...

Specifies the files to be removed.

Example

```
>>> ls foo
foo
>>> rm foo
>>> ls foo
foo no such file
>>>
```

List file `foo` to show that it exists, remove the file, and then try to list the file again to show that it is gone.

See Also

cat, **ls**

sa – set process affinity

Change the affinity mask of a process. The affinity mask specifies the processors on which the process can run.

Syntax

sa ***process_id affinity_mask***

Arguments

process_id

Specifies the process ID (PID) of the process for which the affinity mask is to be modified.

affinity_mask

Specifies the new affinity mask, which indicates on which processors the process can run. Bits 0 and 1 of the mask correspond to processors 0 and 1, respectively.

Example

```
>>> memtest -p 0 &
>>> ps | grep  memtest
00000025 001a9700 2      23691 00000001 0      memtest ready
>>> sa 25 2
>>> ps | grep  memtest
00000025 001a9700 2     125955 00000002 1      memtest running
>>>
```

See Also

ps, **sp**

semaphore – show system semaphores

Shows all the semaphores known to the system by traversing the semaphore queue.

Syntax

semaphore

Example

```
>>> semaphore

                Name              Value   Address  First Waiter
------------------------------- -------- -------- --------------------
---
                                   dyn_sync 00000001 00050378
                                dyn_release 00000001 000503A0
                               shell_iolock 00000001 0015D684
                                exit_iolock 00000001 0015D770
                                grep_iolock 00000001 0015DB20
                                eval_iolock 00000001 0015DC0C
                               chmod_iolock 00000001 0015DCF8
^C
>>>
```

set – set environment variable

Sets the value of an environment variable. Use environment variables to pass configuration information between the console firmware and the operating system.

Some environment variables are stored in nonvolatile memory.

For a listing of predefined environment variables, see Table 5–2.

Syntax

set *envar value* [-default] [-integer] [-string]

Arguments

envar

Specifies the name of the environment variable to be assigned a new value. For a listing of predefined environment variables, see Table 5–2.

value

Specifies the value to be assigned to the environment variable. Depending on the environment variable, the value must be a numeric value or an ASCII string.

Options

–default

Restore the environment variable to its default value.

–integer

Create an environment variable that is set to an integer value.

–string

Create an environment variable that is set to an ASCII string value.

Examples

1. `>>> set MODE FASTBOOT`

   Set the mode for controlling the level of testing done at power-on or after console initialization to FASTBOOT. The FASTBOOT value indicates that you want the system to execute minimal console diagnostics.

2.
```
>>> set VME_A16_BASE 0
>>> set VME_A24_BASE a00000
>>> set VME_A24_SIZE 400
>>> set VME_A32_BASE 80000000
>>> set VME_A32_SIZE 4000
```

   Set the following:

   - The base address of the VMEbus A16 address space to be %x0
   - The base address of the VMEbus A24 address space to be %x0xa00000
   - The size of the VMEbus A24 address space to be 1 MB
   - The base address of the VMEbus A32 address space to be %x80000000

- The size of the VMEbus A32 address space to be 16 MB

3. `>>> set EWA0_PROTOCOLS BOOTP`
   Set the network protocol for booting and other network functions to be BOOTP.

4. `>>> set BOOTDEF_DEV ewa0`
   Set the default device from which the system attempts to boot to EWA0.

5. `>>> set AUTO_ACTION BOOT`
   Set the system's default console action to boot after an error, halt, or power-on.
6. `>>> set BOOT_FILE avme.sys`
   Set the file name to be used when the system's boot requires a file name.
7. `>>> set BOOT_OSFLAGS 0,1`
   Set the system's default boot flags to 0,1.
8. `>>> set foo 5`
   Create environment variable `foo` and set its value to 5.

See Also

clear, **show**

set led – display char on LED

Displays a character on the front panel light emitting diode (LED).

Syntax

set led *char* [-b]

Argument

char

Specifies the character to be displayed on the front panel LED. Specify the character in quotation marks (""). You must specify a metacharacter with a backslash (\ ) prefix.

Options

–b

Display the character in bright mode. The default is dim mode.

Example

```
>>> set LED W -b
```

Display an uppercase W on the LED panel at full brightness.

See Also

show led

set reboot srom – set reboot mode to Serial ROM Mini-Console

Enters the Serial ROM (SROM) Mini-Console.

When you issue this command, the module enters the SROM Mini-Console the next time you reset or power on the system. Once issued, the command prevents you from rebooting from the console until you alter NVRAM bytes using the SROM Mini-Console.

Note

If the I/O module's debug jumper is installed, the system displays the SROM Mini-Debugger prompt every time you power on the system. While in the SROM Mini-Debugger, you can start the SRM console by entering the **st** command and then entering address 0x8000 at the address prompt as follows:

```
SROM> st
a> 8000
```

Syntax

```
set reboot srom
```

Example

```
>>> set reboot srom
```

Set the reboot flag to enter Serial ROM Mini-Console the next time you reset or power on the system.

set toy sleep – disable TOY clock's internal oscillator

Disables the DS1386 TOY clock's internal oscillator. When you execute this command, bit 8 of the MONTH register of the device is set to 1, disabling the TOY clock's oscillator. This prevents the TOY clock's time registers from advancing and lengthens the life of the device's internal lithium battery. The next time you power on the system, the console firmware automatically reenables the oscillator, enabling the clock to count time again.

This command is useful for final testing during manufacturing or for preparing the system for storage.

Note

Reset the time and date once the module is powered on after disabling the battery.

Syntax

```
set toy sleep
```

Example

```
>>> set toy sleep
```

Set the TOY clock into storage mode. The clock is automatically reenabled on subsequent initialization.

sh – create new shell process

Creates a new shell process. Each shell process implements most of the functionality of the Bourne shell.

Syntax

```
sh [-v] [-x] [-d] [-l] [-r] [-p] [arg ...]
```

Arguments

arg ...

Specifies one or more arguments that are to be passed to the new shell process. Specify the arguments as text strings terminated with white space.

Options

–v

Print lines as they are read.

–x

Show commands just before executing them.

–d

Delete standard input (STDIN) when the shell is done.

–l

Trace the lexical analyzer (show tokens as they are recognized).

–r

Trace the parser (show rules as they execute).

–p

Trace the execution engine (show routines called).

Example

```
>>> sh          # start a new shell\bold))
>>>              # the new shell's prompt\bold))

>>> sh -v <foo # execute command file "foo" and show lines as read in
>>> sh -x <foo # print out commands as they are executed and after
>>>             # all substitutions have been performed.
```

show – display system information

Displays the current value of an environment variable or other system parameter.

Syntax

show [*system_param*] [*envar*]

Arguments

system_param

Specifies the type of information that is to be displayed. Specify one of the following parameters:

Parameter	Description
config	Displays the system configuration.
device [*device-name*]	Displays information about devices and controllers in the system.
hwrpb	Displays the Alpha hardware restart parameter block (HWRPB).
led [-hex]	Displays a character on the LED panel. The **-hex** option displays the contents of the LED register instead of the character that is set to be displayed.
map	Displays system virtual memory map.
mode	Displays the current mode, FASTBOOT or NOFASTBOOT.
pal	Displays the version of PALcode for VMS and OSF (UNIX).
version	Displays the version of the console firmware.

You can specify a device name with the **device** parameter. The name that you specify can include abbreviations or an asterisk (*) as a wildcard character. The naming convention for system devices is as follows:

```
dka0.0.0.0.0
| || | | | |
| || | | | +-- Hose # : Always zero for Alpha VME 5/352 and 5/480 SBCs
| || | | +---- Slot # : On PCI System =
| || | |                <PCI bus * 1000>+<PCI function *100>+<PCI slot>
| || | +--- Channel # : Always zero.
| || +---- Bus Node # : Device's bus ID (i.e. SCSI node ID plug #).
| |+--- Device Unit # : Device's unique system unit number.
| +---- Controller ID : One letter controller designator.
+---------- Driver ID : Two letter port or class driver designator.
                           PK - SCSI port, DK - SCSI class
                           EW - Ethernet Port
```

envar

Displays the value of the specified environment variable. For a listing of predefined environment variables, see Table 5–2.

Examples

1.
```
>>> show version
version                V1.1-0 Jul 1 1996 10:16:59
>>>
```
Display the version of the firmware running on the system.

2.
```
>>> show auto_action
boot
>>>
```
Display the default system power-on action.

3.
```
>>> show bootdef_dev
ewa0
>>>
```
Display the system's default boot device. In this example, the default boot device is EWA0.

4.
```
>>> show config
                    Digital Equipment Corporation
                          AlphaVME 5/480

SRM Console T1.0-0  VMS PALcode V1.19-8, OSF PALcode V1.21-8

             MEMORY:  128 Meg of system memory
  System Controller:  VIC64 Enabled

Hose 0, PCI
        slot 0  DECchip 7407
        slot 1  DECchip 21040-AA  ewa0.0.0.1.0  00-00-F8-23-B7-8E
        slot 2  NCR 53C810       pka0.7.0.2.0  SCSI Bus ID 7
        slot 3  Intel 82378

>>>
```

5.
```
>>> show device
dva0.0.0.0.1             DVA0
ewa0.0.0.1.0             EWA0           08-00-2B-1D-27-AA
pka0.7.0.2.0             PKA0           SCSI Bus ID 7
>>>
```
Display all devices and controllers in the system. The display output includes the device name, device ID, device type, and device internal firmware revision information (if available).

6.
```
>>> show device e
ewa0.0.0.6.0               EWA0         08-00-2B-1D-27-AA
```
Display devices that start with "e."

7.
```
>>> show device dk           # Show SCSI disks.
dkc0.0.0.2.0               DKC0                          RZ57
```
Display all devices starting with "dk" (all SCSI disks).

8.
```
>>> show device mk           # Show SCSI tape drives.
mke0.0.0.4.0               MKE0                        TLZ04
>>>
```
Display all devices starting with "mk" (all SCSI tapes).

9.
```
>>> show hwrpb
HWRPB is at 2000
       .
       .
       .
display of the contents of HWRPB registers
```

```
            .
            .
            .
>>>
```

Display the system's HWRPB address and register data.

10. `>>> show led`

    Display the current character being displayed on the LED panel.

11. `>>> show led -hex`

    Display the contents of the LED register.

12.
```
>>> boot -halt
>>> show map
pte 00001020 v FFFFFC0902408000 p 00000000 V KR    SR            FR
FW
pte 00001028 v FFFFFC090240A000 p 00000000 V KR    SR
FW
pte 00001020 v FFFFFC0902C08000 p 00000000 V KR    SR            FR
FW
pte 00001028 v FFFFFC0902C0A000 p 00000000 V KR    SR
FW
pte 00001020 v FFFFFC0B02408000 p 00000000 V KR    SR            FR
FW
pte 00001028 v FFFFFC0B0240A000 p 00000000 V KR    SR
FW
pte 00001020 v FFFFFC0B02C08000 p 00000000 V KR    SR            FR
FW
pte 00001028 v FFFFFC0B02C0A000 p 00000000 V KR    SR
FW
>>>
```

Note

The map is empty after all console initialization. To fill in the page table entries, enter the **boot** command with the **–halt** option.

See Also

set, set led

show_log – display NVRAM error log information

Displays console-detected fault information that was previously stored in the error log area of NVRAM. If you do not specify command-line options, the command displays the most recent fault.

Before using the **show_log** command, you must initialize the error log by issuing the **clear_log** command.

Note

Console error logging is completely independent of the operating system's error logging.

Syntax

```
show_log [{-n [count]} -all -new]
```

Options

–n [*count*]

Display the specified number of most-recent faults that are logged into the NVRAM error log area. The default value for *count* is 1.

–all

Display all faults logged into the NVRAM error log area. All faults are marked as seen so you can display new faults easily by using the **–new** option. This option always causes the command to display all logged faults.

–new

Display new faults logged into the NVRAM error log area; displays faults that have *not* been previously displayed with the **–all** option.

Examples

1. >>> **show_log**

```
============================ F A U L T #1
============================

Time of Error: 13:08:39  9-AUG-1997
Diagnostic   : Interval Timer
Pass Count   : 1     Test Number: 4     Failing Point: 18
Error Message: Interrupt not invoked and should have been
>>>
```

Display the most recent fault.

2. >>> **show_log -n 3**

```
============================ F A U L T #1
============================

Time of Error: 13:10:06  9-AUG-1997
Machine Check: IOC Controller
SCB Vector   : 67
IOC Status 0 : 0400031604000316
IOC Status 1 : 0400000004000000
PC           : 0000000000064c40

============================ F A U L T #2
============================

Time of Error: 13:08:39  9-AUG-1997
Diagnostic   : Interval Timer
Pass Count   : 1     Test Number: 4     Failing Point: 18
Error Message: Interrupt not invoked and should have been

================================================================
==

No more faults found

================================================================
==

>>>
```

Display the two most-recent faults since they are the only ones logged into NVRAM.

See Also

clear_log

sleep – suspend execution

Suspends execution of a console process for a specified number of seconds. The console process temporarily wakes up every second to check for and kill pending bits.

Syntax

sleep [-v] ***time***

Argument

time

Specifies the number of seconds to sleep. The default is one second.

Option

–v

Use a time value of milliseconds. The default is 1000 milliseconds (one second).

Examples

```
1. >>> ((sleep 10; echo hi there)&)
   >>>
(10 seconds elapse...)
hi there
```

Sleep for 10 seconds, then execute the **echo** command.

```
2. >>> sleep -v 20
```

Sleep for 20 milliseconds.

This command does not function if **set toy sleep** has been issued.

sort – sort a file

Arranges the lines of a file in lexicographic order and writes the results to standard output (STDOUT). The size of the file that sort can handle is limited by the size of memory.

Syntax

sort *file*

Argument

file

Specifies the file to be sorted.

Example

```
>>> echo > foo 'banana
_>pear
_>apple
_>orange'
```

Create file `foo` with 4 lines.

```
>>> sort foo
apple
banana
orange
pear
```

Sort file `foo` and display the output.

sp – set priority

Modifies the priority of a process.

Note

Changing the priority of the process impacts the behavior of the process and the rest of the system.

Syntax

sp ***process_id priority***

Arguments

process_id

Specifies the process ID (PID) of the process for which the priority is being set.

priority

Specifies the new priority for the specified process. Priority values range from 0 to 7, with 7 being the highest priority.

Example

```
>>> memtest -p 0 &
>>> ps | grep  memtest
00000025 001a9700 2      23691 00000001 0      memtest ready
>>> sp 25 3
>>> ps | grep  memtest
00000025 001a9700 3     125955 00000001 0      memtest ready
>>>
```

Raise the priority of process 25 from 2 to 3.

See Also

ps, **sa**

start – start program

Starts program execution at the specified address or starts drivers.

Syntax

start [-drivers [*device_prefix*]] [*address*]

Argument

address

Specifies the PC address at which to start execution.

Options

–drivers [*device_prefix*]

Specifies the name of the device or class of devices to stop. If you do not specify a device prefix, the command starts all drivers.

Examples

1. >>> **start -driver ewa 400**
   Start program execution at address 400.
2. >>> **start -drivers**
   Start all the drivers in the system.

See Also

continue, **init**, **stop**

stop – stop CPU or device

Stops the CPU or a specified device.

Syntax

stop [-drivers [*device_prefix*]] [*processor_num*]

Argument

processor_num

Specifies the processor to stop. If you use this argument, specify 0.

Option

–drivers [*device_prefix*]

Stop the specified device or all devices of the specified device class. If you do not specify a device prefix, the command stops all drivers.

Example

```
>>> stop
```

Stop the processor.

See Also

continue, **init**, **start**

update – update flash ROMs

Loads updated firmware into the system's flash ROMs (FEPROMs). Prior to using this command, you must close DIP switch #2 on your Alpha VME 5/352 or 5/480 SBC's I/O module and you must issue the **boot** command.

During the update process, each byte of the FEPROM is verified. Each step provides for a certain number of retries to perform the operation successfully on a particular byte of the FEPROM. If a failure occurs in any of the steps, an error message is displayed on the console.

If the update is successful, a success message is displayed on the console.

Notes

You must reset or power on the system to run the new image in the FEPROMs; otherwise, the previous console image executes out of memory.

Be sure to disable FEPROM writing after completing the update process by setting switch #2 back to the open position.

For more information about updating firmware, see Section 5.7.

Syntax

```
update [-file filename] [-protocol transport]
       [-device source_device] [-target target_device]
```

Options

–file *filename*

Update the FEPROM with the specified image.

–protocol *transport*

Use the specified source transport protocol. Valid protocols are MOP and TFTP. See Section 5.4 for more information on using the TFTP protocol to read files across the network.

–device *source_device*

Load the new FEPROM update image from the specified device. Currently, the only valid device is EWA0.

–target *target_device*

Use the specified target device for the upgrade operation. Valid target devices are CONSOLE and USERFLASH.

Example

```
>>> boot -fi alphavme5_v1_0 -prot mop ewa0
(boot ewa0.0.0.1.0 -file alphavme5_v1_0 -flags 0)

Trying MOP boot.
.............

Network load complete.
Host name: oemert
Host address: aa-00-04-00-56-4b

bootstrap code read in
base = 1c2000, image_start = 0, image_bytes = db000
initializing HWRPB at 2000
initializing page table at 1b4000
initializing machine state
setting affinity to the primary CPU
jumping to bootstrap code
starting console on CPU 0
initialized idle PCB
initializing semaphores
initializing heap
initial heap 200c0
memory low limit= 1b2000
heap = 200c0, 17fc0
initializing driver structures
initializing idle process PID
XDELTA not enabled.
initializing file system
initializing timer data structures
lowering IPL
CPU 0 speed is 2.08 ns (481MHz)
create dead_eater
create poll
create timer
create powerup
128 Meg of system memory
2MB Bcache
probing hose 0, PCI
bus 0, slot 1 -- ewa -- DECdhip 21040-AA
bus 0, slot 2 -- pka -- NCR 53C810
entering idle loop
Skipping powerup tests...
AlphaVME 5/480 Common Console V1.0-0, built on Sep 24 1997 at
09:20:32

>>> update
(update -path noname -target console)

new: 1.0-0

       Note: Module DIP Switch #2 must be CLOSED to enable Updates!
```

```
                    FEPROM UPDATE UTILITY
                    --->CAUTION<---
          EXECUTING THIS PROGRAM WILL CHANGE YOUR CURRENT ROM!

Do you really want to continue[Y/N]?:y

          DO NOT ATTEMPT TO INTERRUPT PROGRAM EXECUTION!
          DOING SO MAY RESULT IN LOSS OF OPERARBLE STATE.

The program will take at most several minutes.

Erasing the target flash device...
.......
Erasure completed.
Programming...
.......
Programming completed
Verifying...
Update successfu

      Note: Module DIP Switch #2 should be OPENED to disable Updates!


>>>
```

Part III
Diagnostics

Part III discusses the diagnostics for DIGITAL Alpha VME 5/352 and 5/480 single-board computers (SBCs). This part consists of the following chapters:

- Chapter 7, Diagnostics and System Initialization
- Chapter 8, Console Mode Diagnostics

7

Diagnostics and System Initialization

Diagnostics for the Alpha VME 5/352 and 5/480 SBCs provide a fast, high coverage suite of power-on self-test (POST) diagnostics to be invoked automatically at power-on and system reset. In addition to the POST diagnostics, there are ROM-based console mode diagnostics that provide additional testing and fault isolation. You invoke the console mode diagnostics by entering commands at your terminal. You also have the option of using diagnostic environment variables to gain more control over your test environment.

This chapter introduces you to DIGITAL Alpha VME 5/352 and 5/480 SBC diagnostics by discussing the:

- POST diagnostics, Section 7.1
- System initialization sequence and countdown, Section 7.2
- POST NVRAM and memory diagnostics descriptions, Section 7.3

7.1 POST Diagnostics

Your SBC invokes POST diagnostics when you apply power to or reset the system. In this mode, a sequence of diagnostics is executed without operator intervention.

Once the SROM code has been loaded into the 8 KB internal instruction cache, a very basic system initialization is performed in preparation for starting the console firmware. After enough of the system has been initialized, the flash ROM-based console is loaded into system memory and execution is transferred to it. During this phase of console startup, the system automatically invokes several more diagnostics and executes them without operator intervention.

The system LED display indicates progress of the SROM initialization by showing a countdown from 8 to 1.

If a failure is detected by the SROM-based tests, the test sequence halts and the LED displays the number of the failing test. If the Intel SIO is successfully configured and the console UART test passes, the SROM does all I/O through the console UART.

Failures detected beyond the SROM do not halt the POST sequence. Instead, the display freezes at the first failing test, and the sequence attempts to continue to console mode. An attempt is also made to write the diagnostic log to the console terminal.

You can affect the POST sequence by using certain user-selectable, control parameters (implemented as environment variables) that allow the initialization to continue, despite the existence of some errors that you may not wish to treat as fatal.

7.2 System Initialization Sequence and Countdown

During SROM initialization and console tests, the LED display shows a countdown indicating progress. The console serial output also reports the countdown if the environment variable CONSOLE is set to SERIAL. The SROM initialization and console tests execute and display output as shown in Table 7–1.

Table 7–1 SROM Initialization and Console Tests

Initialization Procedures	LED Display	Console Display
Read the SROM and initialize the CPU, the CIA chip, the PCI bus, COMM1 port, and the SIO chip.		
	8	8..
Detect the CPU speed, initialize the CPU and CIA Bcache registers, and turn off the Bcache.		
	7	7..
Initialize CIA memory control registers, wake up the DRAMs, and determine the amount of memory that is installed.		
	6	6..
Enable the Dcache and Bcache. Disable ECC reporting, read from memory, and then write back to memory with a good ECC. Clear the CPU and CIA error registers.		
	5	5..
	4	4..
Write to memory (data=address), read from memory, and compare. Check the ECC error status. Load the SRM console and perform a checksum.		
	2	2..
Enable all Scache. Set and flush the Icache.		
	1	1..
Jump to the console.	0	starting console on CPU 0
Initialize console, test memory and NVRAM, and probe the PCI bus		See sample output below
Perform console SCSI test.	A	SCSI Tests...
Perform console heartbeat test.	B	Heartbeat Test...
Perform console interval timer tests.	C	Interval Timer Tests...
Perform console TOY clock tests.	D	Time-of-Year Test...
Perform console serial com port tests.	E	

Table 7–1 SROM Initialization and Console Tests (Continued)

Initialization Procedures	LED Display	Console Display
Perform console Ethernet ROM tests.	F	Ethernet ROM Tests...
Perform console internal loopback tests.	G	NI Loopback Test...
Perform console watchdog test.	H	Watchdog Test...
Perform console VIP/VIC65 tests.	I	VIP Tests...
		>>>

A sample of actual console output follows. Note that the SROM version, CPU speed, memory size, cache size, and SRM version appear in boldface type. You should record and store this information for safekeeping. You will be asked for this in the event that you call for support.

```
Alpha VME 5xxx V1.0
8..7..6..5..4..2..1..starting console on CPU 0
initialized idle PCB
initializing semaphores
initializing heap
initial heap 200c0
memory low limit = 12c000
heap = 200c0, 17fc0
initializing driver structures
initializing idle process PID
XDELTA not enabled.
initializing file system
initializing time data structures
lowering IPL
CPU 0 speed is 2.08 ns (481MHz)
64 Meg of system memory
2MB Bcache
probing hose 0, PCI
bus 0, slot 1 -- ewa -- DECchip 21040-AA
bus 0, slot 2 -- pka -- NCR 53C810
entering idle loop
SCSI Tests...
Heartbeat Tests...
Interval Timer Tests...
Time-of-Year Test...
Ethernet ROM Tests...
NI Loopback Test...
Watchdog Test...
VIP Tests...
.................
Alpha VME 5/480 Common Console V0.0-1, built on Feb 14 1997 at 12:55:07
```

7.3 POST NVRAM and Memory Diagnostics Descriptions

This section provides details on the POST NVRAM and memory diagnostics. These diagnostics run during system initialization testing.

POST Nonvolatile RAM Diagnostic

The POST NVRAM Diagnostic verifies the SBC's NVRAM. It performs a data integrity test, through power cycles, and performs write, read, and comparison operations on specific NVRAM locations used for diagnostics. This diagnostic also checks for uninitialized NVRAM by comparing the stored checksum with the calculated checksum.

Description

This test executes at the beginning of console boot before the console drivers and devices have been initialized.

Test Name: None; executes when the power is turned on

POST Memory Diagnostic

The POST Memory Diagnostic verifies system memory. It runs with ECC enabled. If the test detects a memory error that cannot be corrected with ECC, it logs the error in the error logging area of NVRAM.

Description

The POST Memory Diagnostic executes at the beginning of console boot before the console drivers and devices have been initialized. The test provides the following coverage:

Memory bits	Stuck bits, bit transition fault, or bit coupling fault.
Decoder logic	An address selects no memory, two or more addresses select the same memory cell, or one address selects more than one cell.
Sense amplifier logic	Stuck fault or coupling fault.
Component and path coverage	The CPU memory control logic, etch from the CPU to the daughter card connectors, etch from the CPU backup cache control to the backup cache and from backup cache to the memory bus. The daughter card is assumed good since it is tested separately in manufacturing.

See also the description of the **memtest** console command in Chapter 6.

Note

This test is dependent upon the setting of the environment variable MODE. Setting MODE to FASTBOOT results in a quick memory verification test. NOFASTBOOT results in a full memory test.

Test Name: None; executes when the power is turned on

8
Console Mode Diagnostics

This chapter describes the following console mode diagnostic tests, which might be run during system initialization testing or from the console:

- Heartbeat Timer Test
- Interval timer tests
- DECchip 21040 Ethernet controller tests
- DALLAS DS1386 RAMified watchdog timekeeper tests
- LAN Address ROM Test
- NCR 53C810 PCI-SCSI I/O processor tests
- Watchdog Timer Interrupt Test
- VME interface tests

Section 8.1 provides a summary of the dianostics.

8.1 Console Mode Diagnostics Summary

You can invoke some diagnostics directly from the console terminal, and you can control them by using command options and diagnostic environment variables. These tests may require operator intervention.

Table 8–1 shows the console mode diagnostic tests and the command you can use to invoke them. You can invoke the majority of these tests at the console prompt.

Table 8–1 Console Mode Diagnostic Tests

HW Under Test	Command	Description
Device Exerciser	**exer** *device*	Exercises one or more devices
Memory and Cache		
– Memory exerciser test	**memtest** or **mem_ex**	
Network Interface		
– DECchip 21040 internal loopback test	**niil_diag -t 1**	
– DECchip 21040 external loopback test	**niil_diag -t 2**	
–DECchip 21040	**nicsr_diag -t 1**	Reads the configuration register
–DECchip 21040 CSR tests	**nicsr_diag -t 2** **nicsr_diag -t 3**	Command/status register read test Register write/read test

Table 8–1 Console Mode Diagnostic Tests (Continued)

HW Under Test	Command	Description
NVRAM + TOY Clock		
– NVRAM tests	**ds1386_diag -t 1**	NVRAM write/read test
	ds1386_diag -t 2	NVRAM unique address test
	ds1386_diag -t 3	NVRAM march test
– Time-of-year (TOY) clock register tests	**ds1386_diag -t 4**	Bit pattern test on TOY clock register
	ds1386_diag -t 5	Time keeper test
SCSI		
– SCSI device tests	**ncr810 -t 1**	MBLT write/read test
	ncr810 -t 2	Prints command/status registers
	ncr810 -t 3	Register read/write test
	ncr810 -t 4	Chip reset test
	ncr810 -t 5	Internal loopback test
	ncr810 -t 6	External loopback test
	ncr810 -t 7	SCSI interrupt test (drives must be removed)
Timers		
– Heartbeat timer test	**hbeat_diag -t 1**	Verifies heartbeat frequency=1024 Hz
– Interval timer tests	**i8254_diag -t 1**	Timer 2 interrupt test
	i8254_diag -t 2	Data line test with Timer 2
	i8254_diag -t 3 *	P2 connector test using Timers 0, 1, and 2
	i8254_diag -t 4 *	Master/slave timer test
	i8254_diag -t 5	Timer 2 noninterrupt test
	i8254_diag -t 6	Periodic real-time test using VIC64 chip
– Watchdog timer test	**wdog_diag -t 1**	Timer interrupt test
*Requires external loopback connector configured as shown in Figure 8–1.		
VME InterfaceTests		
– VIP PCI configuration register test	**vip_diag -t 1**	
– VIP register write/read test	**vip_diag -t 2**	
– VIC64 register write/read test	**vip_diag -t 3**	
– Scatter-gather RAM test	**vip_diag -t 4**	
MISC		
– Ethernet hardware address test	**enet_diag -t 1**	Displays LAN address ROM
	enet_diag -t 2	LAN address ROM test

Heartbeat Timer Test

The Heartbeat Timer Diagnostic Test verifies that a heartbeat interrupt is generated at the correct interval (1024 Hz) and is properly dismissed by way of the module clear heartbeat register.

This test checks for the following logic:

- Heartbeat timer and interrupt delivery mechanism
- Module clear heartbeat register

Heartbeat Timer Test

Console Command: `hbeat_diag -t 1`

Command Option:

–dd: Print detailed test information on each pass.

Miscellaneous Notes

- This is a POST diagnostic.
- The test expects timer interrupts to be enabled. If they are not enabled, an interrupt count of zero results.
- You cannot run this test concurrently with other tests.

Interval Timer Tests

The Interval Timer Tests test the functionality of the 8254 interval timer chip and surrounding external circuitry, including latches, programmable-array logic (PAL) devices and printed circuit board module etch. The intent of the tests is to verify that timers 0, 1, and 2 can generate a CPU interrupt, if properly enabled, at the programmed frequency. Since all three interval timers of the 8254 chip have different external configurations, several tests are required for complete test coverage.

These tests require that you properly program *both* timer 0 and 1 and connect them externally for successful operation.

Timer 2 Terminal Count Test

The Timer 2 Terminal Count Test exercises Timer 2 with timer interrupts enabled. The gate input for Timer 2 is always enabled and the clock input is connected to a 10 MHz (100 ns period) clock source.

Timer 2 is programmed to mode 0, interrupt on terminal count. After the timer is initially programmed to mode 0 and loaded with a count value, the OUT output is low and remains low until the internal count value reaches zero. When the count value reaches zero, OUT output is asserted high and remains high until Timer 2 is reprogrammed. The event of OUT transitioning from low to high should generate a CPU interrupt.

The interrupt service routine (ISR) invoked due to the timer generated interrupt sets a global flag indicating the interrupt took place and that software was dispatched to the correct point.

Console Comman: `i8254_diag -t 1`

Miscellaneous Notes

- The interrupt enable bits for Timers 0 and 2 (bits 4 and 5 of the interrupt status register at address 0x4010) are not writable directly. You toggle bits 4 and 5 by writing to addresses 0x4010 and 0x4014, respectively. In both cases, the data written is Don't Care.
- A read of the interrupt status register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
- Due to hardware limitations on interrupt detection, the value programmed for Timer 2 must be greater than 2.
- See the Intel 8254 interval timer sheet for more details.

Timer 2 Square Wave Test

The Time 2 Square Wave Test exercises Timer 2. The gate input for Timer 2 is always enabled and the clock input is connected to a 10 MHz (100 ns period) clock source.

Timer 2 is programmed to mode 3, square wave mode. After the timer is initially programmed for mode 3 and then loaded with a count value, the OUT output produces a continuous, square wave output whose period is equal to the count value multiplied by the period of the clock input. The count values are chosen such that they check stuck NDATA lines.

The event of OUT transitioning from low to high should generate a CPU interrupt, provided the timer 2 interrupt enable bit is set.

The ISR invoked due to the timer generated interrupt increments an interrupt counter and sets a global flag indicating the interrupt took place and that software was dispatched to the correct point. The test verifies that the interrupt count is within a certain range, based on the count value the timer was programmed with and the duration of time that interrupts were enabled.

Console Command: `i8254_diag -t 2`

Miscellaneous Notes

- The interrupt enable bits for Timers 0 and 2 (bits 4 and 5 of the interrupt status register at address 0x4010) are not directly writable. You toggle bits 4 and 5 by writing to addresses 0x4010 and 0x4014, respectively. In both cases, the data written is Don't Care.
- A read of the interrupt status register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
- Due to hardware limitations on interrupt detection, the value programmed into Timer 2 must be greater than 2.
- See the Intel 8254 interval timer sheet for more details.

3 Timers Loopback Test

The 3 Timers Loopback Test exercises Timer 2, Timer 1, and Timer 0. The gate input for Timer 2 and Timer 1 is always enabled and the clock input is connected to a 10 MHz (100 ns period) clock source. Timer 0 accepts its input through a P2 loopback connector to which the output of Timer 1 and Timer 2 is tied. Timer 2 is the gate input and Timer 1 provides the clock.

This test essentially emulates the real-time time provider and slave scheme found in the Real-Time Clock and Interval Device Driver functional specification.

Note

A VMEbus P2 loopback connector is required. See Figure 8–1 for a description of the loopback connections.

The **–lp** option enables the timers indefinitely, making the SBC the master time provider for Test #4.

Timer 2 and Timer 1 are programmed to mode 3, square wave mode. Timer 0 is programmed to mode 1. After you program the timers with the appropriate mode and load them with a count value, the OUT output produces a continuous, square wave output that has a period equal to the count value multiplied by the period of the clock input. In this test Timer 2 provides a major clock, which basically provides the start time of Timer 0, and Timer 1 produces a much faster clock called the minor clock, which controls the rate that Timer 0 counts down.

Timer 0 is the only interrupt that is enabled during this test. The event of OUT transitioning from low to high should generate a CPU interrupt.

The ISR invoked due to the timer generated interrupt increments an interrupt counter and sets a global flag indicating the interrupt took place and that software was dispatched to the correct point. The test verifies that the interrupt occurs, and that no more than one interrupt occurs per major clock cycle.

Console Command: `i8254_diag -t 3`

Command Options:

- **–np**: Do not print a P2 connector message.
- **–lp**: Prevent timers from being stopped at the end of the test. This option is required before you invoke Test #4.

Timer 0 Loopback Test

The Timer 0 Loopback Test exercises only Timer 0. Timer 0 accepts its clock and gate input from the P2 loopback connector. In this test, you can cause Timer 0 inputs on the P2 connector to be driven by a master Alpha VME SBC running Test 3 by specifying the **–lp** option on the command line.

This test essentially emulates the slave system found in the Real-Time Clock and Interval Device Driver functional specification.

This test enables only Timer 0 as is done in Test 3, but does not use Timer 1 or Timer 2. The clock and gate come from the timers on the master Alpha VME SBC. Timer 0 interrupts when the gate is received and its count is decremented to 0.

Note

A VMEbus P2 loopback connector is required. See Figure 8–1 for a description of the loopback connections.

Console Command: `i8254_diag -t 4`

Command Option:

- **–np**: Do not print a P2 connector message.

Miscellaneous Notes

Test #3 must be invoked with the **–lp** option on the master Alpha VME SBC prior to invoking this test.

Timer 2 Interrupt Test

The Timer 2 Interrupt Test exercises Timer 2 with the timer interrupt disabled. The gate input for Timer 2 is always enabled and the clock input is connected to a 10 MHz (100 ns period) clock source.

Timer 2 is programmed to mode 0, interrupt on terminal count. After the timer is initially programmed to mode 0 and loaded with a count value, OUT output is low and remains low until the internal count value reaches zero. When the count value reaches zero, OUT output is asserted high and remains high until Timer 2 is reprogrammed. The event of OUT transitioning from low to high should set the Timer 2 status bit and not generate a CPU interrupt.

The ISR global flag is checked verifying that the ISR was not invoked. The Timer 2 status bit is checked to indicate the interrupt took place.

Console Command: `i8254_diag -t 5`

Miscellaneous Notes

- The interrupt enable bits for Timers 0 and 2 (bits 4 and 5 of the interrupt status register at address 0x4010) are not directly writable. Bits 4 and 5 are toggled by writing to addresses 0x4010 and 0x4014, respectively. In both cases, the data written is Don’t Care.
- A read of the interrupt status register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
- Due to hardware limitations on interrupt detection, the value programmed into Timer 2 must be greater than 2.
- See the Intel 8254 interval timer sheet for more details.

Timer 1 Interrupt Test

The Timer 1 Interrupt Test verifies the interrupt path of Timer 1 (periodic real-time timer). Timer 1 is programmed to mode 3, square wave mode. After the timer is initially programmed to mode 3 and loaded with a count value, OUT output is low and remains low until the internal count value reaches zero. When the count value reaches zero, OUT output is asserted high and remains high until timer 1 is reprogrammed.

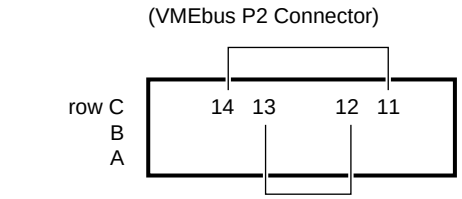
A global interrupt count flag is checked verifying whether the ISR was invoked.

Console Command: `i8254_diag -t 6`

Figure 8–1 Loopback Descriptions for Interval Timer Test 3 and 4

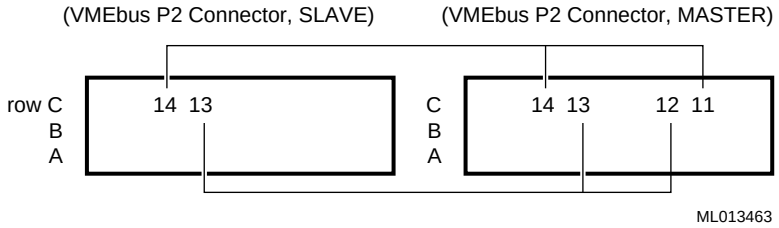
Configuration for Interval Timer test 3

To make a loopback for test 3 connect pin C11 to C14. With a second jumper, connect C12 to C13.



Configuration for Interval Timer test 4 (MASTER/SLAVE Alpha VME)

For test 4, the MASTER signals must be the input for the second Alpha VME module. Connect pins C11 and C14 of the MASTER to C14 of the SLAVE. With a second jumper, connect C12 and C13 of the MASTER to C13 of the SLAVE.



DECchip 21040 Ethernet Controller Tests

The DECchip 21040 Ethernet Controller diagnostics verify that the internal and external loopback mechanisms of the DECchip 21040 Ethernet controller chip are operating properly and are performing write and read operations on behalf of all configuration registers.

Ethernet Internal Loopback Test

The Ethernet Internal Loopback Test transmits Ethernet packets from the transmit ring in main memory, loops them back at the MAC layer, and returns them to the receive ring in main memory. No traffic is put on the network cable.

This test transmits Ethernet packets from the transmit ring in main memory and places them on the network medium (twisted-pair cable). It concurrently listens to the line that carries its own transmissions and returns them to the receive ring in main memory. Received packets not identified as test packets are discarded for the duration of the test.

Note

To run the external loopback test, you must use a 10baseT loopback connector (H4082-AA). The external loopback test does not run if the device is connected to an open network.

This test checks the following logic respectively:

- The device's internal logic up to but not including the Ethernet transmission logic
- The on-chip transmit/receive circuitry and the passive external components that connect to the twisted-pair interface

Console Command

- For internal loopback: **`niil_diag -t 1`**
- For external loopback: **`niil_diag -t 2`**

Command Option:

–dd: Print detailed test information on each pass.

DECchip 21040 PCI Configuration Register Dump

The DECchip 21040 PCI Configuration Register Dump Test reads the PCI configuration registers of the DECchip 21040 and prints them to the standard output.

Console Command: `nicsr_diag -t 1`

DECchip 21040 Control/Status Register Dump

The DECchip 21040 Control/Status Register (CSR) Dump Test reads the CSRs of the DECchip 21040 and prints them to standard output.

Console Command: `nicsr_diag -t 2`

DECchip 21040 Configuration Register Test

The DECchip 21040 Configuration Register Test performs write and read operations on the chip's configuration registers with data patterns of all 1s, all 0s, and alternating 1s and 0s. Upon exiting, the test returns the configuration registers to their initial values.

Console Command: `nicsr_diag -t 3`

Command Option:

–dd: Print detailed test information on each pass.

Miscellaneous Notes

This test runs only when you power on the system.

DALLAS DS1386 NVRAM Watchdog Timekeeper Tests

The DALLAS DS1386 NVRAM Watchdog Timekeeper tests verify the 32 KB of NVRAM and the real-time clock of the DALLAS DS1386. Tests 1 through 3 exercise the NVRAM and Tests 4 and 5 exercise the real-time clock. The tests test the DS1386, and decoders.

The functionality of the watchdog feature is tested by a separate diagnostic test. No alarm features are tested, since the alarms are not used.

The NVRAM is tested on a page basis; there are 128 pages each containing 256 bytes. However, the first page has reserved addresses for the real-time clock registers.

NVRAM March I Test

The NVRAM March I Test writes, reads, and compares all 32 KB of NVRAM with data patterns of all 1s, all 0s, alternating 1s and 0s, and shifting 1s and 0s. If the quick verify option is set (default), only the first location of each page is tested. The no quick verify option tests every location (32 KB) of the NVRAM.

Note

The contents of the NVRAM are overwritten by this diagnostic and restored on test completion. If the module is reset during this test, the NVRAM contents are undefined.

Console Command: `ds1386_diag -t 1`

Command Options:

- **–dd**: Print detailed test information on each pass.
- **–nqv**: Test every location in NVRAM. The default is to test one location per page.

Miscellaneous Notes

This diagnostic is an extended test.

NVRAM Address-On-Address Test

The NVRAM Address-On-Address Test writes, reads, and compares all 32 KB of NVRAM using this unique page offset for test data. Locations in the DS1386 are byte wide. Therefore, you do not have enough room to write the unique address into each corresponding location. However, this test writes the unique page offset to its corresponding location in NVRAM.

If you set the quick verify option (default), only the first location of each page is tested. The no quick verify option tests every location (32 KB) of the NVRAM.

Note

The contents of the NVRAM are overwritten by this diagnostic and restored on test completion. If the module is reset during this test the NVRAM contents are undefined.

Console Command: `ds1386_diag -t 2`

Command Options:

- **–dd**: Print detailed test information on each pass.
- **–nqv**: Test every location in NVRAM. The default is to test one location per page.

Miscellaneous Notes

This diagnostic is an extended test.

NVRAM March II Test

The NVRAM March II Test verifies NVRAM addressing by marching (writing, reading, and comparing) a 0x00 byte value through a field of 0xFF. Each iteration reads the entire 32 KB for background pattern of 0xFF. If you set the quick verify option (default), only the first location of each page is tested. The no quick verify-option, -**nqv**, tests every location (32 KB) of the NVRAM.

Note

The contents of the NVRAM are overwritten by this diagnostic test and restored on test completion. If the SBC is reset during this test, the NVRAM contents are undefined.

Console Command: `ds1386_diag -t 3`

Command Options:

- **–dd**: Print detailed test information on each pass.
- **–nqv**: Test every location in NVRAM. The default is to test one location per page.

Miscellaneous Notes

This diagnostic is an extended test.

TOY Clock Bitwalk Test

The TOY Clock Bitwalk Test does a walking 1, walking 0, and A5 on the TOY clock registers. It also tests the rollover cases associated with keeping time.

The watchdog reset enable bit in the module control register is set to zero to ensure that a watchdog expiration does not cause a hardware reset to occur. Secondly, the contents of the command register is saved and the transfer enable bit is set to 0 to disable updates to the registers while the diagnostic is in progress.

The diagnostic bit patterns are then walked through all 14 registers. Next, the seconds, minutes, hours, day, month, and year registers are programmed such that the next clock tick rolls over for each of these parameters. The updates to the registers are started and updated for a three second time period. After the three second update period, the registers are then examined to verify that each parameter did indeed roll over to the appropriate value.

The diagnostic test cleans up by reenabling the watchdog reset bit in the module control register and restoring the original contents of the TOY clock command register.

Note

The current date and time has to be reset after invoking this diagnostic test since approximately 3 seconds of time is lost for each pass.

Console Command: `ds1386_diag -t 4`

Command Option:

–dd: Print detailed test information on each pass.

Miscellaneous Notes

This diagnostic is an extended test.

TOY Clock Time Advancement Test

The TOY Clock Time Advancement Test is a power-on diagnostic. It verifies that the TOY clock registers are advancing with clock ticks.

The test reads the current value of the seconds register. Then the test sleeps for 1.2 seconds and reads the seconds register again expecting it to have incremented with the exception of the rollover case. The rollover case is where the seconds register advances from 59 to 0. If the rollover case is encountered, the test sleeps for another second and reads the register again. This is repeated four times.

Console Command: `ds1386_diag -t 5`

Command Option:

–dd: Print detailed test information on each pass.

Miscellaneous Notes

This diagnostic is a POST diagnostic.

Local Area Network Address ROM Tests

The Local Area Network (LAN) Address ROM tests test the integrity of the LAN address ROM, decoders, and printed circuit board module etch. The LAN address ROM contains the Ethernet station address of the module.

LAN Address ROM Dump Test

The LAN Address ROM Dump Test dumps the contents of the 32 octets within the LAN address ROM to the screen. No verification of the data is performed.

Console Command: `enet_diag -t 1`

Command Options:

–dd: Print the LAN address ROM to screen.

–np: Do not print the LAN address ROM to screen.

Miscellaneous Notes

- The LAN address ROM octets must be read by using longword aligned byte accesses.
- This diagnostic is an extended test.

LAN Address ROM Verification Test

The LAN Address ROM Verification Test verifies the format of the data in the LAN address ROM. It verifies that the octets are ordered appropriately and that the checksums are correctly calculated based on the LAN address.

Console Command: `enet_diag -t 2`

Command Option:

–dd: Print the LAN ROM address to screen.

Miscellaneous Notes

- The LAN address ROM octets must be read by using longword aligned byte accesses.
- This test is considered a POST diagnostic.

Figure 8–2 LAN Address ROM Format

NCR 53C810 PCI-SCSI I/O Processor Tests

The NCR 53C810 PCI-SCSI I/O processor tests check the NCR810 SCSI controller chip. The tests do not require a drive to be attached to the SCSI port and are meant to be a power-on check of the NCR810 chip's low-level modes through programmed I/O issued from the CPU. No NCR810 scripts execute during these tests.

All tests set up the diagnostic support environment, allocate memory, set up the PCI configuration registers, and check for the default values in the command/status registers as defined by the NCR810 53C810 chip specification.

Note

If any of these tests fails, the console SCSI driver does not restart after the test. This causes SCSI devices connected to the system to be removed from the device list, and any attempts to run the disk exerciser or boot from a disk fails. (The console command **show device** lists the currently installed devices.)

NCR810 PCI Configuration Register Test

The NCR810 PCI Configuration Register Test prints the current setting of the NCR810 PCI configuration registers to the console screen using formatted output.

Console Command: `ncr810_diag -t 1`

Command Option:

–np: Do not print the contents of the configuration register.

NCR810 Command/Status Register Dump

The NCR810 Command/Status Register Dump Test displays the contents of all of the control/status registers (CSRs) on your screen. No test of the contents is performed.

Console Command: `ncr810_diag -t 2`

Command Option:

–np: Do not print the contents of the configuration register.

NCR810 Command/Status Register Test

The NCR810 Command/Status Register Test writes, reads, and compares the contents of all NCR810 CSRs that can be tested. When the test finishes, it returns the registers to their initialized values.

Console Command: `ncr810_diag -t 3`

Command Option:

–lp: Loop on write and read operations.

NCR810 Command/Status Register Reset Value Test

The NCR810 Command/Status Register Test checks that a reset of the NCR810 sets the CSRs to their default values as defined by the NCR810 53C810 chip specification.

Console Command: `ncr810_diag -t 4`

NCR810 Internal Loopback Test

The NCR810 Internal Loopback Test performs a SCSI loopback internal to the NCR810 chip. The following data patterns are used: all 1s, all 0s, alternating 1s and 0s. The test also verifies parity checking and that the SCSI reset control lines can be toggled internally.

Console Command: `ncr810_diag -t 5`

NCR810 Internal Live Bus Loopback Test

The NCR810 Internal Live Bus Loopback Test performs an internal SCSI loopback that also drives the signal lines on the SCSI bus.

You must remove all devices from the SCSI bus before running this test. Devices on the bus interfere with the test and cause false error reports. Also, the test data may produce invalid device instructions and cause the devices to hang.

First, the test places the SCSI bus in a high impedance state by loading a data pattern that causes the output drivers to draw no current. Then the test checks the output latches for the correct data. The test also verifies parity checking and that the SCSI reset control lines can be toggled internally. The following data patterns are used: all 1s, all 0s, alternating 1s and 0s.

Console Command: `ncr810_diag -t 6`

NCR810 Interrupt Test

The NCR810 Interrupt Test verifies the interrupt connection between the NCR810 and the SIO controller to the CPU. The test enables a general-purpose timer, which generates an interrupt that is dispatched to the CPU through the SIO controller. The console PALcode dispatches to the NCR810_diag ISR, which clears the interrupt.

Console Command: `ncr810_diag -t 7`

Miscellaneous Notes

- These tests do not run in parallel with the SCSI exerciser tests.

- No external loopback connectors are needed for the loopback tests.
- References - NCR 53C810 PCI-SCSI I/O Processor specification Revision 2.1

Watchdog Timer Interrupt Test

The Watchdog Timer Interrupt Test verifies the functionality of the watchdog timer by checking its ability to handle a user programmed watchdog timer reset. The test checks logic associated with the:

- Watchdog timer
- Some reset logic
- DS1386 TOY clock

Watchdog Timer Interrupt Test

The Watchdog Timer Interrupt Test sets the diagnostic-in-progress bit and invokes a watchdog timeout by loading a short time value into the watchdog timeout register. The test queries you to be sure the watchdog LED is off. Upon expiration of the timeout value, a HALT interrupt is expected. After the expected time, the test evaluates the reset reason register. If the HALT interrupt did not occur, or the watchdog reason was not set, the test calls out an error. Also, the test asks you to verify that the watchdog LED is now on. At the end of the test, the watchdog timer and diagnostic-in-progress bit are disabled.

Console Command: `wdog_diag -t 1`

Command Options:

- **–dd**: Print detailed test information on each pass.
- **–nc**: Do not prompt user to verify the state of the LED.
- **–np**: Override the **-nc** option by prompting user to verify the state of the LED.

Miscellaneous Notes

The purpose of setting the diagnostic-in-progress bit is to avoid an actual system reset when the watchdog timer expires. The watchdog expiration first causes a HALT interrupt. Approximately 300 ms later an actual system reset occurs, unless the diagnostic-in-progress bit is set. The reset reason register shows a watchdog reset reason whether or not the diagnostic-in-progress bit is set. The HALT interrupt and the reset reason are used for this diagnostic. User interaction can be suppressed with the **–nc** option.

VME Interface Tests

The VME Interface Tests verify the VME interface logic on the Alpha VME 5/352 and 5/480 SBCs, including the VME interface processor (VIP), the Cypress VIC064 chip, the scatter/gather RAMs, and some of the interrupt paths from the VME corner to the Alpha processor. These tests perform no VMEbus transactions and, therefore, require no additional VMEbus modules.

VIP PCI Configuration Register Test

The VIP PCI Configuration Register Test reads the first 8 longwords of the VIP PCI configuration space. Only the device and vendor ID, and base addresses 0, 1, 2, and 3 are compared to an expected value. The remaining longwords are always read and displayed only if you specify the **–dd** option.

Console Command: `vip_diag -t 1`

Command Option:

–dd: Print detailed test information.

VIP Register Write/Read Test

The VIP Register Write/Read Test ensures that the bits of a VIP register can be written and read correctly; verifying the data path and internal access.

Console Command: `vip_diag -t 2`

Command Option:

–dd: Print detailed test information.

VIC Register Write/Read Test

The VIC Register Write/Read Test ensures that the bits of a VIC register can be written and read correctly; verifying the data path and internal access.

Console Command: `vip_diag -t 3`

Command Option:

–dd: Print detailed test information.

VME Scatter/Gather RAM Test

The VME Scatter/Gather RAM Test verifies the integrity of the scatter/gather RAM by performing write, read, and verify operations of various patterns to the entire scatter/gather RAM.

Console Command: `vip_diag -t 4`

Command Option:

–dd: Print detailed test information on each pass.

Part IV
Appendixes

Part IV consists of the following appendixes:

- Appendix A, Console Command Summary
- Appendix B, Troubleshooting
- Appendix C, Module Connector Pin Assignments

A

Console Command Summary

Table A–1 summarizes the DIGITAL Alpha VME 5/352 and 5/480 SBC console commands.

Table A–1 Console Command Summary

Command	Options	Arguments
alloc	[**-flood**] [**–z** *heap_address*]	*size* [*modulus*] [*remainder*]
boot	[**–file** *boot_file*] [**–flags** *longword,...*] [**-protocols** *enet_protocol*] [**–halt**]	[*boot_device*]
break		[*break_level*]
cat	[**-l** *length*]	[*file...*]
chmod	[[{– + =} {**r w x b z**}]...]	*file...*
chown		*pid address ...*
clear		*envar*
clear_log		
date		[[[[*yyyy*]*mm*]*dd*]*hhmm*[.*ss*]]
deposit	[**-b -w -l -q -o -h**] [**-physical -virtual -gpr -fpr -ipr**] [**–n** *count*] [**–s** *step*]	[*device*] *address data*
dynamic	[**–c** [**-r**]] [**–h**] [**-p**] [**–v**] [**-extend** *byte_count*] [**–z** *heap_address*]	
echo	[**–n**]	*args...*
eval	[**-ib -io -id -ix**] [**-b -o -d -x**]	*operand1 operand2 operator*
examine	[**-b -w -l -q -o -h -d**] [**-physical -virtual -gpr -fpr -ipr**] [**–n** *count*] [**–s** *step*]	[*device*] *address*
exer	[**–sb** *start_block*] [**–eb** *end_block*] [**–p** *pass_count*] [**–l** *blocks*] [**–bs** *block_size*] [**–bc** *block_per_io*] [**–d1** *buf1_string*] [**–d2** *buf2_string*] [**–a** *action_string*] [**–sec** *seconds*] [**–m**] [**–v**] [**-delay** *milliseconds*]	[*device...*]
exit		*exit_value*
false		
free		*address...*
grep	[**-c**] [**-i**] [**-n**] [**-v**] [**–f** *file*]	*expression* [*file...*]
hd	[**-byte -word -long -quad**]	*file...*

Table A–1 Console Command Summary (Continued)

Command	Options	Arguments
help		[*command-spec...*]
init_ev		
init	[**–d** *device*]	
kill		*pid...*
line		
ls	[**–l**]	[*file...*]
man		[*command-spec...*]
memexer		[*number_of_tests*]
memtest	[**-sa** *start_address*] [**-ea** *end_address*] [**-l** *length*] [**-ba** *block_address*] [**-bs** *block size*] [**-i** *address_inc*] [**-p** *pass_count*] [**-d** *data_pattern*] [**-rs** *random_seed*] **-rb**] [**-f**] [**-m**] [**-z**] [**-h**] [**-mb**] [**-t**] [**-g**] [**-se**]	
net	[**-s**] [**-sa**] [**-ri**] [**-ic**] [**-id**] [**-l0**] [**-l1**] [**-rb**] [**-csr**] [**-els**] [**-kls**] [**-cm** *mode*] [**-se**] [**-da** *node_address*] [**-l** *file_name*] [**-lw** *wait_in_seconds*] [**-sv** *mop_version*]	[*port*]
ps		
pwrup		
rm		*file...*
sa		*process_id affinity_mask*
semaphore		
set	[**-default**] [**-integer**] [**-string**]	*envar value*
set led	[**-b**]	*char*
set reboot srom		
set toy sleep		
sh	[**-v**] [**-x**] [**-d**] [**-l**] [**-r**] [**-p**]	*[arg ...]*
show		*system_param envar*
show log	[{**-n** [*count*]} **-all -new**]	
sleep	[**-v**]	*time*
sort		*file*
sp		*process_id priority*
start	[**-drivers** [*device_prefix*]]	*address*
stop	[**-drivers** [*device_prefix*]]	*processor_num*
update[1]	[**-file** *filename*] [**-protocol** *transport*] [**-device** *source_device*] [**-target** *target_device*]	

[1] You must issue the **boot** command before using **update**.

B
Troubleshooting

The DIGITAL Alpha VME 5/352 and 5/480 SBCs include extensive diagnostic capabilities that execute when you power on the system. These include both SROM and flash ROM code. This appendix:

- Briefly discusses SROM and flash ROM diagnostics, Sections B.1 and B.2
- Provides guidance on troubleshooting systems that include a PMC I/O companion card, B.3
- Briefly discusses use of the dot matrix display by operating systems and applications, B.4
- Provides troubleshooting tips, B.5

For details about system diagnostics, see Chapter 7.

B.1 SROM Diagnostics

SROM diagnostics execute when you power on the system and display decreasing numeric codes (8, 7, ...1) on the dot matrix display to indicate status. All SROM tests must pass successfully before the flash ROM and console diagnostics run. If one or more SROM diagnostics fail, the flash ROM and console diagnostics are not loaded and a single right angle bracket prompt (SROM>) appears on the console terminal. The code of the failing diagnostic appears on the dot matrix display. In some cases, additional information appears on the console terminal.

B.2 Flash ROM Diagnostics

When the SROM diagnostics complete successfully, the flash ROM diagnostics are loaded, decompressed, and executed. Flash ROM diagnostics use an ascending (A, B,..., I) character-based code to indicate progress. If one or more flash ROM-based diagnostics fail, the code representing the first error remains on the dot matrix display and alternates between dim and bright intensity.

If all SROM and flash ROM diagnostics pass, and you have not set any AUTO_ACTION environment variables, the console prompt (>>>) appears on the console terminal, and a “rotating bar” appears on the dot matrix display.

B.3 Troubleshooting Systems that Include a PMC I/O Companion Card

A problem in the PMC I/O companion card that hangs the PCI bus signal lines could cause diagnostics to report problems throughout the I/O subsystem and in the PCI controller of the processor chip. If you have a PMC I/O companion card installed and you are experiencing diagnostic failures, remove it and rerun the POST diagnostics.

B.4 Operating System and Application Use of the Dot Matrix Display

Operating system and application software can use the dot matrix display. Once the system boots, the dot matrix display is no longer under control of the console code and can change. The console automatically clears the display before booting an image.

B.5 Troubleshooting Your SBC

Table B–1 lists symptoms and corrective actions you can use to troubleshoot Alpha VME 5/352 and 5/480 SBCs. See the *DIGITAL Alpha VME 5/352 and 5/480 Single-Board Computers User Manual* for more information on system diagnostics.

Table B–1 Troubleshooting Your SBC

Symptom	Corrective Action
No LEDs are lit and a prompt does not appear on the console.	Check the power source. If 5 V power is out of specification, the SBC is held in reset. Check that all modules are seated properly.
The green LED is lit and the number 4 appears on the dot matrix display when you power on the system.	Check the seating of the memory modules.
The green LED is lit and the number 0 appears on the dot matrix display when you power on the system.	Ensure that the console terminal is not in “hold screen” mode.
The green LED is lit and a flashing A appears on the dot matrix display when you power on the system.	Check the SCSI termination, the seating of the Alpha VME 5/352 and 5/480 SBC CPU and I/O module assembly, the seating of the breakout modules, the seating of the SCSI cable, and the seating of all SCSI devices.
The green LED is lit and a flashing D appears on the dot matrix display when you power on the system.	Check that the TOY Clock/NVRAM device is seated properly.
The green LED is lit and a flashing F appears on the dot matrix display when you power on the system.	Check the seating of the network address ROM.

Table B–1 Troubleshooting Your SBC (Continued)

Symptom	Corrective Action
The green LED is lit and a flashing G appears on the dot matrix display when you power on the system.	Check the seating of the twisted-pair cable and the nearest network transceiver.
The green LED is lit and a flashing I appears on the dot matrix display when you power on the system.	Check the seating of the Alpha VME 5/352 and 5/480 SBC CPU and I/O module assembly, the seating of the breakout modules, and the seating of all VME devices.
Diagnostics pass but the SCSI tests take more than 10 seconds to complete.	Check the SCSI termination, the seating of the Alpha VME 5/352 and 5/480 SBC CPU and I/O module assembly, the seating of the breakout modules, the seating of the SCSI cable, and the seating of all SCSI devices.
Diagnostics pass but there are no (or unreadable) characters displayed on the console.	Check the console terminal connections and settings (9600 baud, 8-bits, no parity). The terminal should be plugged into the console (CON) port.

C

Module Connector Pin Assignments

Sections C.1 through C.5 provide pin assignment information for the Alpha VME 5/352 and 5/480 SBC:

- CPU module connector, C.1
- I/O Type 1 module connector, C.2
- Primary breakout module connector, C.3
- Secondary breakout module connector, C.4
- PMC I/O companion card connector, C.5

C.1 CPU Module Connector Pin Assignments

The CPU module (54–24827–*xx*) P2 connector has the following power/ground pin assignments:

	Row A	Row B	Row C
Ground	1, 2, 4, 5, 7, 8, 10, 11, 13, 15, 16, 18-23, 28-30	2, 12, 22, 31	3, 4, 7-11, 14-17, 20-22, 24-27, 30
VCC	3, 6, 9, 12, 14, 17, 24-27, 31, 32	1, 13, 32	1, 2, 5, 6, 12, 13, 18, 19, 23, 28, 29, 31, 32

C.2 I/O Module Connector Pin Assignments

Sections C.2.1 through C.2.4 show the pin assignments for the VMEbus connector, console and serial connectors, and the Ethernet connector on the I/O module (54-24319-01).

C.2.1 P1 VMEbus Connector Pin Assignments

Table C–1 lists the pin assignments for the P1 VMEbus connector.P2 Connector

Table C–1 P1 VMEbus Connector Pin Assignments

Pin	Row A	Row B	Row C
1	VME_D0	VME_BBSY_L	VME_D08
2	VME_D1	VME_BCLR_L	VME_D09
3	VME_D2	VME_ACFAIL_L	VME_D10
4	VME_D3	VME_BGIN0_L	VME_D11
5	VME_D4	VME_BGOUT0_l	VME_D12
6	VME_D5	VME_BGIN1_L	VME_D13

Table C–1 P1 VMEbus Connector Pin Assignments (Continued)

Pin	Row A	Row B	Row C
7	VME_D6	VME_BGOUT1_L	VME_D14
8	VME_D7	VME_BGIN2_L	VME_D15
9	Ground	VME_BGOUT2_L	Ground
10	VME_SYSCLK	VME_BGIN3_L	VME_SYSFAIL_L
11	Ground	VME_BGOUT3_l	VME_BERR_L
12	VME_DS1_L	VME_BR0_L	VME_SYSRESET_L
13	VME_DS0_L	VME_BR1_L	VME_LWORD_l
14	VME_WRITE_L	VME_BR2_L	VME_AM5
15	Ground	VME_BR3_L	VME_A23
16	VME_DTACK_L	VME_AM0	VME_A22
17	Ground	VME_AM1	VME_A21
18	VME_AS_L	VME_AM2	VME_A20
19	Ground	VME_AM3	VME_A19
20	VME_IACK_L	Ground	VME_A18
21	VME_IACKIN_L	N/C	VME_A17
22	VME_IACKOUT_L	N/C	VME_A16
23	VME_AM4	Ground	VME_A15
24	VME_A7	VME_IRQ7_L	VME_A14
25	VME_A6	VME_IRQ6_L	VME_A13
26	VME_A5	VME_IRQ5_L	VME_A12
27	VME_A4	VME_IRQ4_L	VME_A11
28	VME_A3	VME_IRQ3_L	VME_A10
29	VME_A2	VME_IRQ2_L	VME_A09
30	VME_A1	VME_IRQ1_L	VME_A08
31	PWRN12	VME_5VSTBY	PWRP12
32	VCC	VCC	VCC

C.2.2 P2 VMEbus Connector Pin Assignments

Table C–2 lists the pin assignments for the P2 VMEbus connector.

Table C–2 P2 VMEbus Connector Pin Assignments

Pin	Row A	Row B	Row C
1	SCSI_DATA0_L	VCC	MSDATA
2	SCSI_DATA1_L	Ground	MSCLK
3	SCSI_DATA2_L	N/C	Ground

Table C–2 P2 VMEbus Connector Pin Assignments (Continued)

Pin	Row A	Row B	Row C
4	SCSI_DATA3_L	VME_A24	KBDATA
5	SCSI_DATA4_L	VME_A25	KBCLK
6	SCSI_DATA5_L	VME_A26	WD_STATUS_OC
7	SCSI_DATA6_L	VME_A27	BREAKOUT0
8	SCSI_DATA7_L	VME_A28	BREAKOUT1
9	SCSI_DP_L	VME_A29	Ground
10	SCSI_ATN_L	VME_A30	EXT_RESET_L
11	SCSI_BSY_L	VME_A31	TMR2_EXT_OP_L
12	SCSI_ACK_L	Ground	TMR1_EXT_OP_L
13	SCSI_RST_L	VCC	TMR_MINOR_IP_L
14	SCSI_MSG_L	VME_D16	TRM_MAJOR_IP_L
15	SCSI_SEL_L	VME_D17	Ground
16	SCSI_CD_L	VME_D18	PP_STB_L
17	SCSI_REQ_L	VME_D19	PP_ERR_L
18	SCSI_IO_L	VME_D20	PP_DATA0
19	Ground	VME_D21	PP_DATA1
20	Ground	VME_D22	PP_DATA2
21	Ground	VME_D23	PP_DATA3
22	Ground	Ground	PP_DATA4
23	VME_MASTER_SW_L	VME_D24	PP_DATA5
24	VCC	VME_D25	PP_DATA6
25	VCC	VME_D26	PP_DATA7
26	VCC	VME_D27	PP_SLCT
27	VCC	VME_D28	PP_PE
28	Ground	VME_D29	PP_BUSY
29	Ground	VME_D30	PP_ACK_L
30	Ground	VME_D31	PP_AFD_L
31	VCC	Ground	PP_INIT_L
32	VCC	VCC	PP_SLIN_L

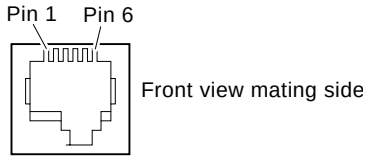
C.2.3 Console and Auxiliary Connector Pin Assignments

Table C–3 lists the pin assigments for the console and auxiliary connectors. Figure C–1 shows a pin assignment diagram.

Table C–3 Console and Auxiliary Connector Pin Assignments

Pin	Signal
1	Ready out (always asserted, tied high with a 150Ω resistor)
2	Transmit +
3	Transmit – (send common, tied to ground)
4	Receive +
5	Receive –
6	Ready in (tied to ground with 3.01KΩ resistor)

Figure C–1 Console and Auxiliary Connector Pin Assignments



MLO-013549

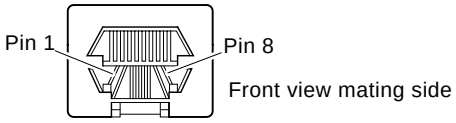
C.2.4 Ethernet Connector Pin Assignments

Table C–4 lists the pin assignments for the Ethernet connector. Figure C–2 shows a pin assignment diagram.

Table C–4 Ethernet Connector Pin Assignments

Pin	Signal
1	Transmit +
2	Transmit –
3	Receive +
4	No connection
5	No connection
6	Receive –
7	No connection
8	No connection

Figure C–2 Ethernet Connector Pin Assignments



MLO-013550

C.3 Primary Breakout Module Connector Pin Assignments

Table C–5 lists the pin assignments for the primary breakout module (54-24663-01). Figure C–3 shows a pin assignment diagram.

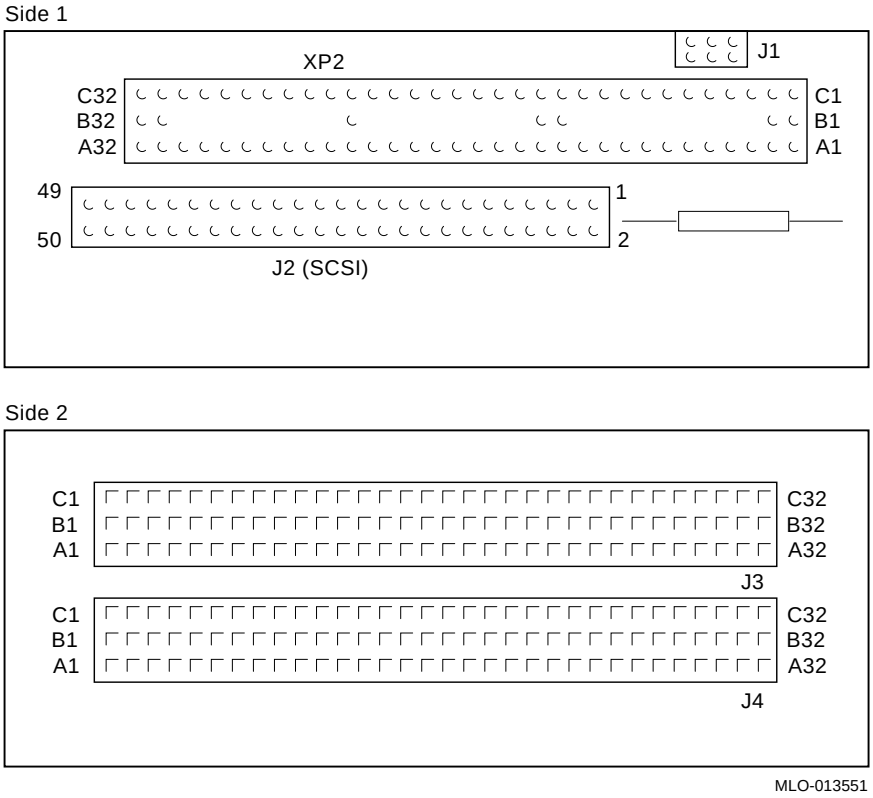
Table C–5 Primary Breakout Module Connector Pin Assignments

Pin	Row A	Row B	Row C
1	SCSI_DATA0_L	VCC	MSDATA
2	SCSI_DATA1_L	Ground	MSCLK
3	SCSI_DATA2_L	N/C	Ground
4	SCSI_DATA3_L	N/C	KBDATA
5	SCSI_DATA4_L	N/C	KBCLK
6	SCSI_DATA5_L	N/C	WD_STATUS_OC
7	SCSI_DATA6_L	N/C	BREAKOUT0
8	SCSI_DATA7_L	N/C	BREAKOUT1
9	SCSI_DP_L	N/C	Ground
10	SCSI_ATN_L	N/C	EXT_RESET_L
11	SCSI_BSY_L	N/C	TMR2_EXT_OP_L
12	SCSI_ACK_L	Ground	TMR1_EXT_OP_L
13	SCSI_RST_L	VCC	TMR_MINOR_IP_L
14	SCSI_MSG_L	N/C	TRM_MAJOR_IP_L
15	SCSI_SEL_L	N/C	Ground
16	SCSI_CD_L	N/C	PP_STB_L
17	SCSI_REQ_L	N/C	PP_ERR_L
18	SCSI_IO_L	N/C	PP_DATA0
19	Ground	N/C	PP_DATA1
20	Ground	N/C	PP_DATA2
21	Ground	N/C	PP_DATA3
22	Ground	Ground	PP_DATA4
23	VME_MASTER_SW_L	N/C	PP_DATA5
24	VCC	N/C	PP_DATA6
25	VCC	N/C	PP_DATA7
26	VCC	N/C	PP_SLCT
27	VCC	N/C	PP_PE
28	Ground	N/C	PP_BUSY
29	Ground	N/C	PP_ACK_L

Table C–5 Primary Breakout Module Connector Pin Assignments (Continued)

Pin	Row A	Row B	Row C
30	Ground	N/C	PP_AFD_L
31	VCC	Ground	PP_INIT_L
32	VCC	VCC	PP_SLIN_L

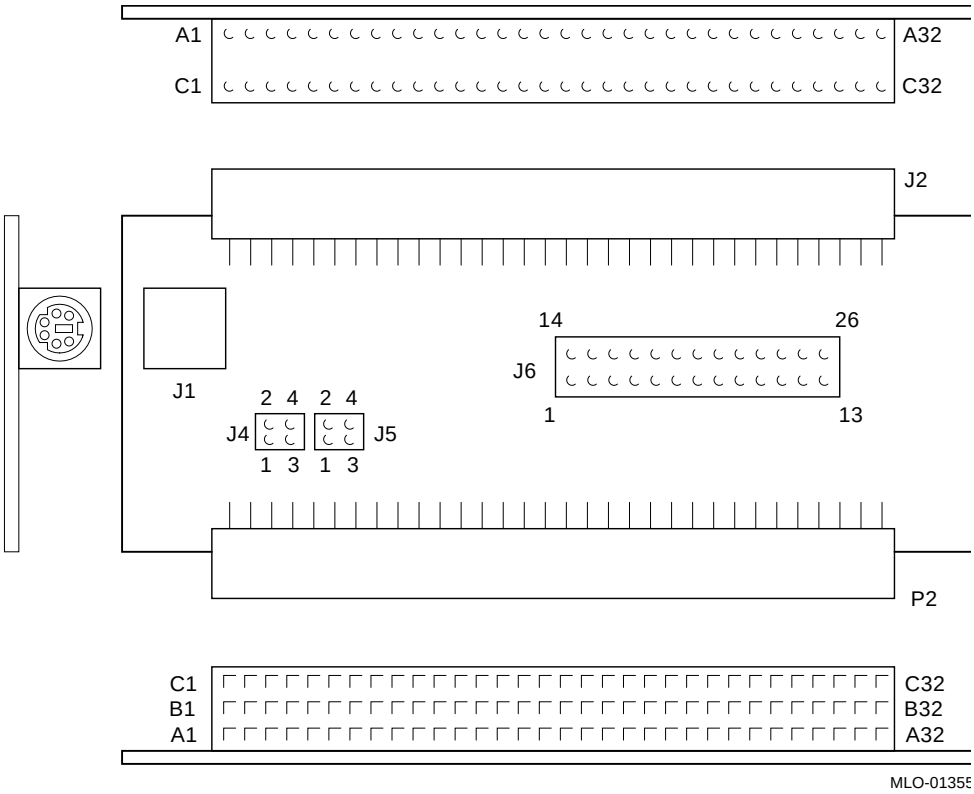
Figure C–3 Primary Breakout Module Connector Pin Assignments



C.4 Secondary Breakout Module Connector Pin Assignments

Figure C–4 shows the layout of the pin assignments for the secondary breakout module. Note the positions of the J1 (keyboard and mouse) and J6 (parallel port) connectors.

Figure C–4 Secondary Breakout Module Connector Pin Assignments



Sections C.4.1 and C.4.2 provide more detail on the J1 and J6 connectors, respectively.

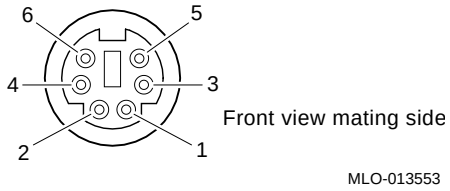
C.4.1 Keyboard and Mouse Connector Pin Assignments

Table C–6 lists the pin assignments for the keyboard and mouse (J1) connector. Figure C–5 shows a pin assignment diagram.

Table C–6 Keyboard and Mouse Connector Pin Assignments

Pin	Signal
1	MOUSE_DATA
2	KBRD_DATA
3	Ground
4	VCC
5	MOUSE_CLOCK
6	KBRD_CLOCK

Figure C–5 Keyboard and Mouse Pin Assignments



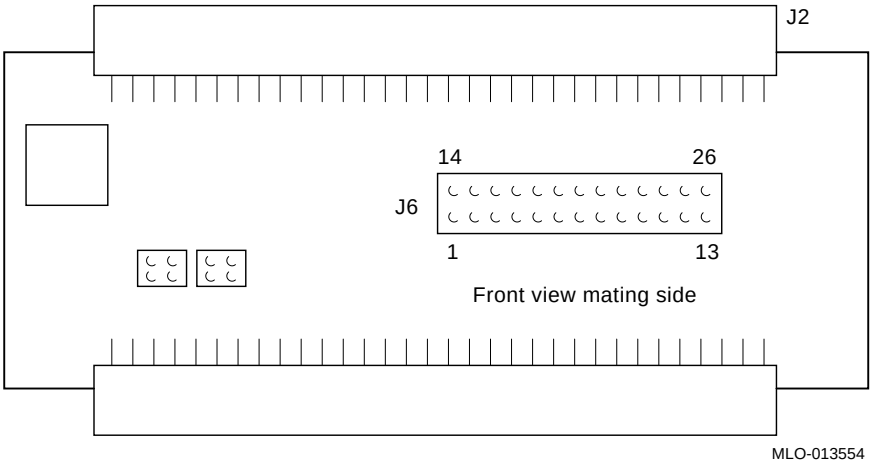
C.4.2 Parallel Port Connector Pin Assignments

Table C–7 lists the pin assignments for the parallel port (J6) connector. Figure C–6 shows a pin assignment diagram.

Table C–7 Parallel Port Connector Pin Assignments

Pin	Signal
1	PP_STB_L
2	PP_DATA0
3	PP_DATA1
4	PP_DATA2
5	PP_DATA3
6	PP_DATA4
7	PP_DATA5
8	PP_DATA6
9	PP_DATA7
10	PP_ACK_L
11	PP_BUSY
12	PP_PE
13	PP_SLCT
14	PP_AFD_L
15	PP_ERR_L
16	PP_INIT_L
17	PP_SLIN_L
18–25	Ground
26	N/C

Figure C–6 Parallel Port Connector Pin Assignments



C.5 PMC I/O Companion Card Connector Pin Assignments

Sections C.5.3 through C.5.3 identify the pin assignments for the following PMC I/O companion card (54-24665-01) connectors:

- PMC option 1 connectors
- PMC option 2 connectors
- Diskette drive connector
- Mouse and keyboard connector

C.5.1 PMC Option 1 Connector Pin Assignments

Figure C–7 shows the locations of the PMC option 1 connectors J11, J12, and J14(the P2 VMEbus signal connector). Tables C–8 through C–10 list the pin assignments for the connectors.

Figure C–7 PMC Option 1 Connectors

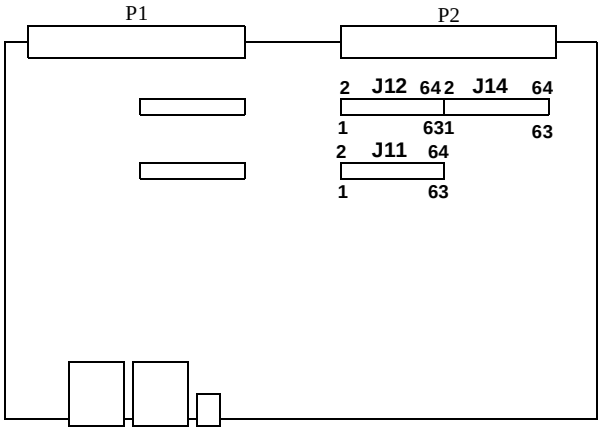


Table C–8 PMC Option 1 J11 Pin Assignments

Signal	Pin	Pin	Signal
Ground	1	2	PWRN12
Ground	3	4	BPCIOPT0_IRQA_L
BPCIOPT0_IRQB_L	5	6	BPCIOPT0_IRQC_L
N/C	7	8	VCC
BPCIOPT0_IRQD_L	9	10	N/C
Ground	11	12	N/C
PCICLK_OPT0_L	13	14	Ground
Ground	15	16	SGNT0_L
SSREQ0_L	17	18	VCC
SVIO	19	20	BPCI_AD31
BPCI_AD28	21	22	BPCI_AD27
PBCI_AD25	23	24	Ground
Ground	25	26	BPCI_CBE3_L
BPCI_AD22	27	28	BPCI_AD21
BPCI_AD19	29	30	VCC
SVIO	31	32	BPCI_AD17
BPCI_FRAME_L	33	34	Ground
Ground	35	36	BPCI_IRDY_L
BPCI_DEVSEL_L	37	38	VCC
Ground	39	40	BPCI_LOCK_L
SVIO	41	42	SVIO
BPCI_PAR	43	44	Ground
SVIO	45	46	BPCI_AD15
BPCI_AD12	47	48	BPCI_AD11
BPCI_AD9	49	50	VCC
Ground	51	52	BPCI_CBE0_L
BPCI_AD6	53	54	BPCI_AD5
BPCI_AD4	55	56	Ground
SVIO	57	58	BPCI_AD3
BPCI_AD2	59	60	BPCI_AD1
BPCI_AD0	61	62	VCC
Ground	63	64	SVIO

Table C–9 PMC Option 1 J12 Pin Assignments

Signal	Pin	Pin	Signal
PWRP12	1	2	Ground
SVIO	3	4	N/C
SVIO	5	6	Ground
Ground	7	8	N/C
N/C	9	10	N/C
VCC	11	12	+3V
S_RST_L	13	14	Ground
+3V	15	16	Ground
N/C	17	18	Ground
BPCI_AD30	19	20	BPCI_AD29
Ground	21	22	BPCI_AD26
BPCI_AD24	23	24	+3V
BPCI_AD17	25	26	BPCI_AD23
+3V	27	28	BPCI_AD20
BPCI_AD18	29	30	Ground
BPCI_AD16	31	32	BPCI_CBE2_L
Ground	33	34	N/C
BPCI_TRDY_L	35	36	+3V
Ground	37	38	BPCI_STOP_L
BPCI_PERR_L	39	40	Ground
+3V	41	42	BPCI_SERR_L
BPCI_CBE1_L	43	44	Ground
BPCI_AD14	45	46	BPCI_AD13
Ground	47	48	BPCI_AD10
BPCI_AD8	49	50	+3V
BPCI_AD7	51	52	N/C
+3V	53	54	N/C
N/C	55	56	Ground
N/C	57	58	N/C
Ground	59	60	N/C
SVIO	61	62	+3V
Ground	63	64	N/C

Table C–10 PMC Option 1 VMEbus P2 Signal Connector (J14) Pin Assignments

Signal	Pin	Pin	Signal
P2_1C	1	2	P2_1A
P2_2C	3	4	P2_2A
P2_3C	5	6	P2_3A
P2_4C	7	8	P2_4A
P2_5C	9	10	P2_5A
P2_6C	11	12	P2_6A
P2_7C	13	14	P2_7A
P2_8C	15	16	P2_8A
P2_9C	17	18	P2_9A
P2_10C	19	20	P2_10A
P2_11C	21	22	P2_11A
P2_12C	23	24	P2_12A
P2_13C	25	26	P2_13A
P2_14C	27	28	P2_14A
P2_15C	29	30	P2_15A
P2_16C	31	32	P2_16A
P2_17C	33	34	P2_17A
P2_18C	35	36	P2_18A
P2_19C	37	38	P2_19A
P2_20C	39	40	P2_20A
P2_21C	41	42	P2_21A
P2_22C	43	44	P2_22A
P2_23C	45	46	P2_23A
P2_24C	47	48	P2_24A
P2_25C	49	50	P2_25A
P2_26C	51	52	P2_26A
P2_27C	53	54	P2_27A
P2_28C	55	56	P2_28A
P2_29C	57	58	P2_29A
P2_30C	59	60	P2_30A
P2_31C	61	62	P2_31A
P2_32C	63	64	P2_32A

C.5.2 PMC Option 2 Connector Pin Assignments

Figure C–8 shows the locations of the PMC option 2 connectors J21 and J22. Tables C–11 and C–12 list the pin assignments for the connectors.

Figure C–8 PMC Option 2 Connectors

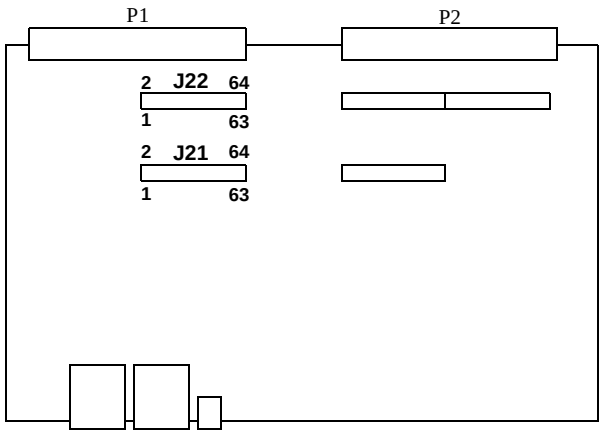


Table C–11 PMC Option 2 J21 Pin Assignments

Signal	Pin	Pin	Signal
Ground	1	2	PWRN12
Ground	3	4	BPCIOPT1_IRQA_L
BPCIOPT1_IRQB_L	5	6	BPCIOPT1_IRQC_L
N/C	7	8	VCC
BPCIOPT1_IRQD_L	9	10	N/C
Ground	11	12	N/C
PCICLK_OPT1_L	13	14	Ground
Ground	15	16	SGNT1_L
SSREQ1_L	17	18	VCC
SVIO	19	20	BPCI_AD31
BPCI_AD28	21	22	BPCI_AD27
PBCI_AD25	23	24	Ground
Ground	25	26	BPCI_CBE3_L
BPCI_AD22	27	28	BPCI_AD21
BPCI_AD19	29	30	VCC
SVIO	31	32	BPCI_AD17
BPCI_FRAME_L	33	34	Ground
Ground	35	36	BPCI_IRDY_L
BPCI_DEVSEL_L	37	38	VCC

Table C–11 PMC Option 2 J21 Pin Assignments (Continued)

Signal	Pin	Pin	Signal
Ground	39	40	BPCI_LOCK_L
SVIO	41	42	SVIO
BPCI_PAR	43	44	Ground
SVIO	45	46	BPCI_AD15
BPCI_AD12	47	48	BPCI_AD11
BPCI_AD9	49	50	VCC
Ground	51	52	BPCI_CBE0_L
BPCI_AD6	53	54	BPCI_AD5
BPCI_AD4	55	56	Ground
SVIO	57	58	BPCI_AD3
BPCI_AD2	59	60	BPCI_AD1
BPCI_AD0	61	62	VCC
Ground	63	64	SVIO

Table C–12 PMC Option 2 J22 Pin Assignments

Signal	Pin	Pin	Signal
PWRP12	1	2	Ground
SVIO	3	4	N/C
SVIO	5	6	Ground
Ground	7	8	N/C
N/C	9	10	N/C
VCC	11	12	+3V
S_RST_L	13	14	Ground
+3V	15	16	Ground
N/C	17	18	Ground
BPCI_AD30	19	20	BPCI_AD29
Ground	21	22	BPCI_AD26
BPCI_AD24	23	24	+3V
BPCI_AD17	25	26	BPCI_AD23
+3V	27	28	BPCI_AD20
BPCI_AD18	29	30	Ground
BPCI_AD16	31	32	BPCI_CBE2_L
Ground	33	34	N/C

Table C–12 PMC Option 2 J22 Pin Assignments (Continued)

Signal	Pin	Pin	Signal
BPCI_TRDY_L	35	36	+3V
Ground	37	38	BPCI_STOP_L
BPCI_PERR_L	39	40	Ground
+3V	41	42	BPCI_SERR_L
BPCI_CBE1_L	43	44	Ground
BPCI_AD14	45	46	BPCI_AD13
Ground	47	48	BPCI_AD10
BPCI_AD8	49	50	+3V
BPCI_AD7	51	52	N/C
+3V	53	54	N/C
N/C	55	56	Ground
N/C	57	58	N/C
Ground	59	60	N/C
SVIO	61	62	+3V
Ground	63	64	N/C

C.5.3 PMC I/O Companion Card Diskette Drive Connector Pin Assignments

Table C–13 lists the pin assignments for the PMC I/O companion card diskette drive connector. Figure C–9 shows a pin assignment diagram for the connector.

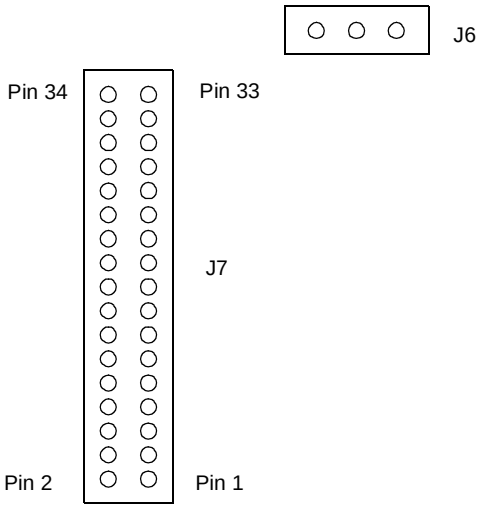
Table C–13 PMC I/O Companion Card Diskette Drive Connector Pin Assignments

Pin	Signal
1	Ground
2	DENSEL
3	Ground
4	No connection
5	Ground
6	DRATE0_L
7	Ground
8	INDEX_L
9	Ground
10	MTR0_L
11	Ground
12	DS1_L

Table C–13 PMC I/O Companion Card Diskette Drive Connector Pin Assignments (Continued)

Pin	Signal
13	Ground
14	DS0_L
15	Ground
16	MTR1_L
17	Ground
18	DIR_L
19	Ground
20	STEP_L
21	Ground
22	WRDATA_L
23	Ground
24	WGATE_L
25	Ground
26	TR0_L
27	Ground
28	WRTPRT_L
29	Ground
30	RDATA
31	Ground
32	HDSEL_L
33	Ground
34	RDSKCHG

Figure C–9 PMC I/O Companion Card Diskette Connector Pin Assignments



C.5.4 PMC I/O Companion Card Keyboard and Mouse Connector Pin Assignments

Tables C–14 and C–15 list the pin assignments for the PMC I/O companion card mouse and keyboard connectors, respectively. Figure C–10 shows a pin assignment diagram for the connectors.

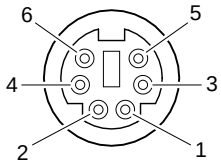
Table C–14 PMC I/O Companion Card Mouse Connector Pin Assignments

Pin	Signal
1	MOUSE_DATA
2	KBRD_DATA
3	Ground
4	VCC
5	MOUSE_CLOCK
6	KBRD_CLOCK

Table C–15 PMC I/O Companion Card Keyboard Connector Pin Assignments

Pin	Signal
1	KBRD_DATA
2	MOUSE_DATA
3	Ground
4	VCC
5	KBRD_CLOCK
6	N/C

Figure C–10 PMC I/O Companion Card Mouse and Keyboard Connector Pin Assignments



Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T