AXPvme
Single-Board Computer

Technical Description

Order Number: EK–EBV1X–TD. B01

Digital Equipment Corporation
Maynard, Massachusetts

July 1995
Printed in U.S.A.

The prepaid Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

S2923

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

2 Console Primer

3 Console Commands

Tables

Preface

Purpose

This guide describes the AXPvme module and its built-in features including the console code and diagnostics.

Audience

This guide is intended for users who wish to have in-depth information about the AXPvme module. This guide does not provide programming information. For programming information, refer to the operating system calls section of your operating system documentation.

Related Documents

Document	Part Number
AXPvme Single-Board Computer Installation/User Guide	EK-EBV1X-IN
VxWorks Digital AXPvme Single-Board Computers Hardware Supplement	AA-QA5HA-TE
VxWorks Programmer's Guide	AA-Q3YLB-TE
DEC OSF/1 Installation Guide	AA-PS2DD-TE

Conventions

This manual uses the following conventions:

Convention	Meaning
Return	Press the key that executes commands or terminates a sequence. This key is labeled Return, Enter, or ↩, depending on your keyboard.
Ctrl/x	While you hold down the Ctrl key, press any other key.
	OSF and VxWorks commands are case sensitive. You must enter commands in the correct case, as printed in the text.
`Monospace type`	Indicates examples of system output or user input.
italics	In commands and examples, italics indicates a value that you should supply.
U*X	Is used in place of UNIX and ULTRIX in this document.
[ ]	Square brackets in command descriptions enclose the optional command qualifiers. Do not type the brackets when entering information enclosed in the brackets.

Convention	Meaning
|	A vertical bar in command descriptions indicates that you have a choice between two or more entries. Select one entry unless the entries are optional.
{ }	Braces indicate that you are required to specify one (and only one) of the enclosed options. Do not type the braces when you enter the command.
( )	Parentheses enclose a set of options that must be specified together.

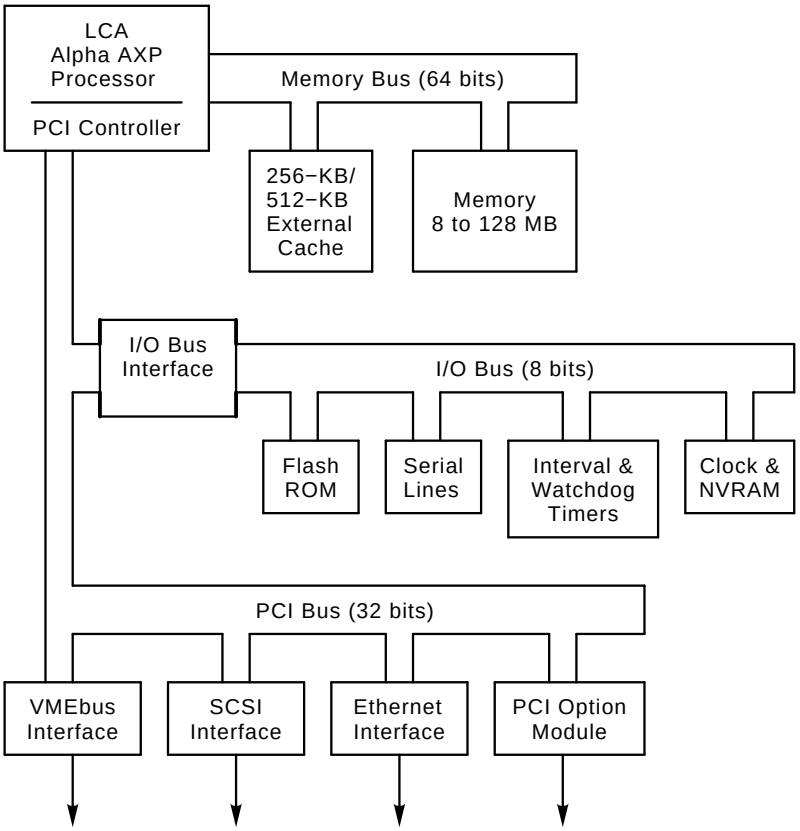
1

Technical Hardware Specifications

1.1 Overview

The AXPvme single-board computer is a 6U-sized Versa Module Eurocard bus (VMEbus) device with the low-cost Alpha (21066A) processor chip, memory, and peripheral device controllers on a single module. The AXPvme single-board computer is for VME real-time and embedded systems, where high performance with a Digital supported operating system is desirable. The supported operating systems are OSF/1 and VxWorks. Figure 1–1 shows a block diagram of the systems on the AXPvme module.

Figure 1–1 AXPvme Module Block Diagram



1.1.1 Related Hardware Specification Documents

The following documents should be used with the hardware specifications provided in this chapter:

- *Alpha Architecture Reference Manual*
- DECchip 21066-AA (Low-cost Alpha) Engineering Specification
- PCI Interface Specification 2.0
- DECchip 21040-AA Specification
- NCR 53C810 Documentation
- Intel 82378ZB Chip Specification
- Chip Specifications for 85C30, 8254, and VIC068/64

1.1.2 Conventions and Terms

The following conventions and terms are used in this chapter:

- All address numbering is in hexidecimal
- Other hexidecimal numbers are suffixed by "h," for example, 4C00h
- RO = Read only
- W1C = Write one to clear
- MBZ = Must be zero, for example, must be written as zero and expected as zero on reads

1.1.3 System Description

The AXPvme system is based on the LCA chip with the PCI as the onboard I/O bus.

1.1.3.1 LCA Processor

The Low-Cost Alpha chip (LCA) is a dual-issue Alpha implementation with full floating-point support, internal 8 Kbyte Instruction and Data caches, full memory, and I/O bus controllers, as well as a PLL clock generator.

1.1.3.2 Memory Controller

The LCA's integral memory controller handles all interface operations to the main memory and external write-back backup cache.

The AXPvme will support 8/16/32/64/128 Mbytes of DRAM memory.

The external backup cache is a fixed 256 Kbytes or 512 Kbytes depending on the module version. Table 1–3 shows the cache size of each AXPvme module.

Table 1–3 AXPvme Backup Cache Memory Size

AXPvme Module	Cache Size
64	256 Kbytes
64LC	256 Kbytes
160	256 Kbytes
100	512 Kbytes
166	512 Kbytes
230	512 Kbytes

1.1.3.3 I/O Controller

The LCA's integral I/O controller implements the interface to the peripheral component interconnect (PCI) bus. The PCI is used as the system I/O bus. All I/O is mapped into the 32-bit PCI address space. Byte, Word, Longword, and Quadword accesses over the PCI are support for CPU initiated cycles. The I/O controller also acts as a PCI target and PCI to main memory bridge for I/O initiated transfers and DMAs.

1.1.3.4 VME Interface

AXPvme supports a full master/slave VME interface to IEC 821, IEEE1014-1987 standards. The module can act as VME system controller (in slot 1), and as such, handles bus arbitration, interrupt handling, and so forth.

The VME interface operates in A16, A24, and A32 addresses spaces with D08, D16, D32, and D64 data transfers.

As a VME master, AXPvme can be programmed to any of the four bus request levels. The system will be a fair requester and can be programmed to release-on-request or when done. An outbound programmable scatter/gather (S/G) is used to facilitate flexible VME access. This S/G allows access to 2048 256-Kbyte pages of VME space and is used to specify address modifier and byte swapping information, and so forth. Full outgoing DMA bursts are supported for fast memory to VME target data transfer. Support for D64 transfers offer data rates of up to >24 Mbytes/s.

As a slave, AXPvme responds to programmed addresses in the A24 and A32 VME address spaces. D08, D16, D32, and D64 transfers including block modes (no D08 block mode) and read-modify-write cycles are supported (though not atomic to main memory). An inbound S/G is employed to give flexibility in slave addressing of the 8-128 Mbytes of AXPvme main memory. The S/G also allows write-protection of designated pages and the implementation of interrupt-based inbound S/G page monitoring.

Byte swapping (selectable via S/G entries) is provided to ease data interchange between the little-endian Alpha AXP and other architectures without software overhead.

Interprocessor communication registers in A16 space are provided for limited message passing and signaling.

1.1.3.5 Network Interface (IEEE 802.3)

A PCI-based network interface is standard on the AXPvme system. Connection to a network is via AUI at the front panel.

1.1.3.6 SCSI

A PCI-based SCSI interface is embedded on the AXPvme computer modules. This interface will provide SCSI-2 support (hardware will support either target and/or initiator). Connection to the SCSI interface is via a P2 breakout module using a standard nonshielded 50-pin low-density connector.

1.1.3.7 ROM

4 Mbyte of flash ROM is provided on processors with 512 Kbyte of cache while 1 Mbyte of flash ROM is provided on processors with 256 Kbyte cache for console, diagnostics, PAL code store, and user-specific applications. This ROM is implemented in flash EPROM to allow in-place ROM updates. Updates are enabled/disabled via software and a module DIP switch. For full details of the ROM usage, see Section 1.11.

The 4 Mbyte of flash ROM is accessed in the bottom 1 Mbyte of PCI memory space. Two bits in the MODULE CONTROL2 REGISTER select the current 1 Mbyte window mapped into PCI space. The systems with 1 Mbyte of flash ROM directly access this ROM without mapping.

1.1.3.8 Console UART

The AXPvme console is driven by a serial line DUART port. This DEC423 port is one of two asynchronous serial lines on the module.

1.1.3.9 Watchdog

A watchdog timer is included as a programmable fail-safe on system software. If enabled and allowed to expire, the watchdog first halts the CPU (causing a jump to console firmware) and then resets the entire AXPvme. On watchdog reset an open-collector signal, accessible via P2, is asserted and the amber front panel LED lights.

1.1.3.10 Interval Timers

AXPvme features three programmable 16-bit interval timer/counters. Two of the timers are driven from a fixed 10 MHz clock, and can be used to generate rate outputs (via P2 pins) or onboard interval interrupts. The third counter /timer is clocked and gated by external inputs (via P2 pins) for event counting or synchronization. The output of this final counter can be enabled to generate an onboard interrupt request.

1.1.3.11 TOY Clock

A basic time-of-year function with battery backup is included on the AXPvme module. This device keeps time and date information with a resolution of 0.01 seconds and an accuracy of +/- 1 minute per month, at 25°C.

1.1.3.12 Nonvolatile RAM

32 Kbytes of nonvolatile RAM are available on the AXPvme card. For full details of the layout and use of the onboard NVRAM, refer to Section 1.16. An onboard DIP switch allows the NVRAM to be supplied by the backplane 5 V standby (to remove dependence on the internal battery). The device will retain data for 10 years in the absence of Vcc, at 25°C.

1.1.3.13 Display

There is a single alphanumeric display on the front panel of the AXPvme, which is accessible in I/O space. The display is used for firmware status information. When not running resident firmware, the display is available for use under user software control.

1.1.3.14 PCI Mezzanine

A single PCI mezzanine connector is located on the system module to allow a PCI option daughtercard to be plugged in. PCI bus arbitration logic in AXPvme fully supports one optional PCI device with up to four PCI option interrupt request lines. The PCI clock is driven from AXPvme at a frequency of 32 MHz with processors with 256 Kbyte chache and 33 Mhz on processors with 512 Kbyte of cache.

1.1.3.15 DIP Switches

AXPvme modules have a set of four DIP switches used to enable/disable various functions. These switches are physically located behind the Network AUI connector and are labeled 1 through 4. The switches and their functions are listed in Table 1–2.

Table 1–2 Dip Switch Functions

Switch No.	Open	Closed
1	TOY clock has no connection to VME 5 V standby suppy.	TOY 5 V line assisted by VME standby power (extends battery life).
2	Flash ROM updates are disabled (Vpp = 0 V).	Flash ROM updates are enabled (Vpp = +12 V).
3	VME bus resets do not generate a module reset.	VME bus resets will generate a module reset.
4	Configures the VME corner to not be the VME system controller on powerup.	Configures the VME corner to be VME system controller on powerup. This has the same effect as connecting VMEP2 pin A23 (VME_MASTER_SWL) to ground for the AXPvme 66, AXPvme 100, and the AXPvme 231.

The default configuration, as delivered from the factory, assumes the module will be a VME system controller (will perform VME bus arbitration from slot 1).

Default Switch Settings:

1 -	Closed	(Allow VME 5 V standby power to assist TOY supply)
2 -	Open	(Flash updates disabled)
3 -	Open	(VMEbus resets do not reset module)
4 -	Closed	(Configured as VME system controller)

An alternate configuration assumes the module is not a VME system controller (is not in slot 1 and does not perform VME bus arbitration).

Nonsystem Controller Switch Settings:

1 -	Closed	(Allow VME 5 V standby power to assist TOY supply)
2 -	Open	(Flash updates disabled)
3 -	Closed	(VMEbus resets do reset module)
4 -	Open	(Not configured as VME system controller)

Note that switches 3 and 4 must always be configured opposite (one open, one closed) for normal system operation.

Note

The AXPvme 66, AXPvme 100, and AXPvme 231 may be configured as the VME system controller by closing switch 4 or by grounding VMEP2 pin A23. Switch 4 must be **open** and VMEP2 pin A23 must **not** be connected to ground for the modules to operate as a non-controller.

1.2 LCA Processor

The DECchip 21066A microprocessor is the second in a family of chips to implement the Alpha AXP architecture. The 21066A is a CMOS based superscalar, superpipelined processor using dual-instruction issue. The DECchip 21066A incorporates a high level of system integration to provide the best-in-class system performance for cost-focused applications. It integrates on-chip fully pipelined integer and floating-point processors, a high bandwidth memory controller, an industry standard I/O controller, an embedded graphics accelerator, internal instruction and data caches, and external cache control.

Summary of the DECchip 21066A features *:

- Fully pipelined, 64-bit RISC architecture
- Supported by multiple operating systems:
  - OSF
  - VxWorks
- Best-in-class performance
  - 231 MHz operation (AXPvme 230)
  - Superscalar, superpipelined (dual issue)
- Pipelined on-chip floating-point unit:
  - IEEE single- and double-precision, Digital F_floating and G_floating, longword and quadword data types. Limited support for D_floating data type.
- On-chip demand-paged memory management unit:
  - A 12-entry I-stream translation buffer with 8 entries for 8Kbyte pages and 4 entries for 4 MB pages
  - A 32-entry D-stream translation buffer with each entry able to map a single 8K, 64K, 512K, or 4 MB page.
  - Super page mapping
- On-chip high-bandwidth memory controller:
  - Full 64-bit memory datapath
  - Dynamic RAM (DRAM) controller
  - 64-bit error correction code (ECC)

* Refers to utilization in AXPvme dual-slot

  - Supports up to 4 banks of memory
  - RAS/CAS memory bus
- PCI I/O controller
  - 32-bit multiplexed address/data
  - Industry standard
  - Burst mode reads and writes
  - Asynchronous operation to CPU
  - Multimaster with peer-to-peer access
- On-chip 8-Kbyte direct mapped write-through data cache
- On-chip 8-Kbyte direct mapped instruction cache
- On-chip control for optional, external, write-back secondary cache:
  - Programmable cache size and speed
- Built-in phase-locked loop (PLL)
  - Frequency multiplier allows low-cost input clock
  - Programmable multiplier values
- Serial ROM interface:
  - Loads Icache after reset
  - Software controlled serial port after initialization
- 3.3 V supply voltage
  - Interfaces directly to 5 V logic

Please refer to the DECchip 21066-AA (Low Cost Alpha Microprocessor Engineering Specification) for details of the LCA chip.

1.3 Memory Subsystem

The memory subsystem of the AXPvme system can be divided into two sections:

- Main memory
- External Bcache

The memory controller, internal to the DECchip 21066, handles the operation and control of the external memories under the configuration control of various LCA internal registers.

An 8-bit ECC is used on the 64-bit memory word for single-bit error correction and double-bit/nibble error detection. The controller generates a CAS-before-RAS cycle to refresh all banks simultaneously.

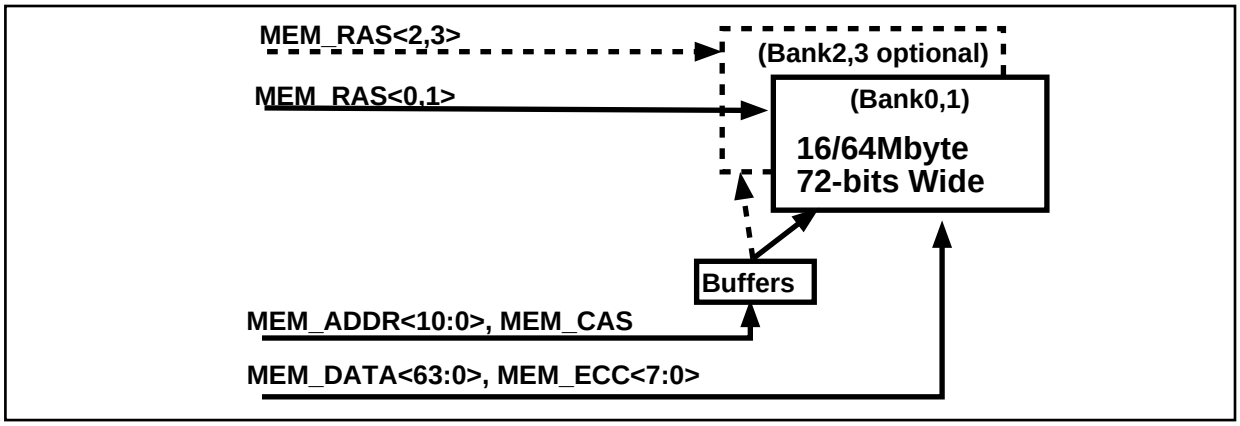
The AXPvme system will support 8/16/32/64/128 Mbytes of main memory.

The external backup cache of AXPvme is a fixed 256 Kbyte or 512 Kbyte SRAM write-back configuration, depending on the module version.

1.3.1 Main Memory

The memory interface is of the form shown in Figure 1–2. The memory controller attempts to maximize fast-page-mode operation. In this way the maximum memory bandwidth for AXPvme is approximately 75 Mbytes/s with a sustainable bandwidth of 64 Mbytes/s shared evenly between the CPU and the IOC.

Figure 1–2 Memory System



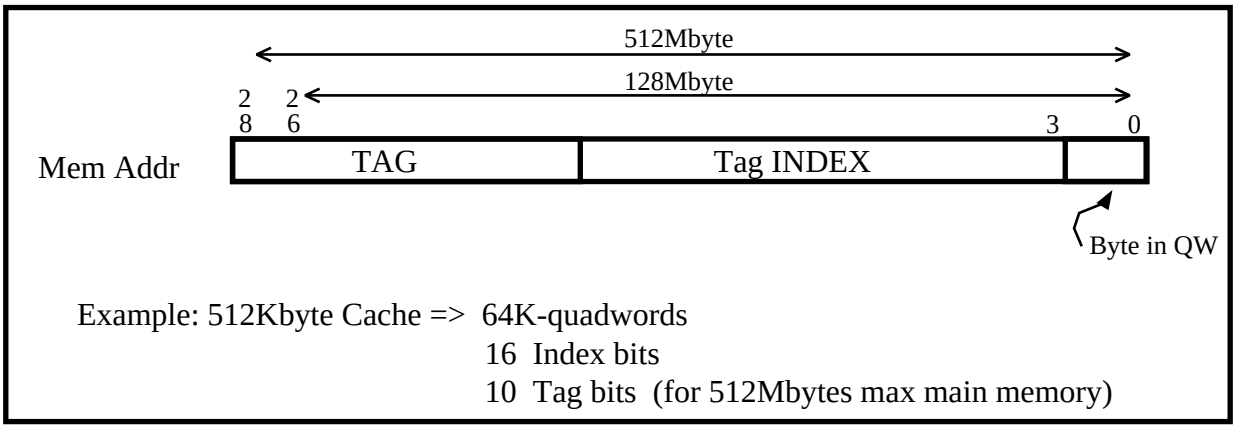
1.3.2 Backup Cache

The external Bcache is an optional system component that increases overall system performance by returning cached main-memory data faster than is possible from the DRAMs. The cache is implemented as 512 Kbytes SRAM. The cache tag size (see Figure 1–3) depends on the cache size. Table 1–3 shows the cache size of each AXPvme module.

Table 1–3 AXPvme Backup Cache Memory Size

AXPvme Module	Cache Size
64	256 Kbytes
64LC	256 Kbytes
160	256 Kbytes
100	512 Kbytes
166	512 Kbytes
230	512 Kbytes

Figure 1–3 Cache Tag



The cache word size is a quadword.

To allow caching of any memory address in the 512-Mbyte range for a 512-Kbyte Bcache requires 10 tag bits. In addition, the cache interface requires one parity bit and one "dirty" bit associated with each tag entry. This gives a total of 12 tag RAM storage bits required.

1.3.3 Memory Initialization

Each memory bank in the AXPvme system is controlled by the values in four associated registers. The correct configuration is set at module power on and should not be altered. There is a single Global Timing Register, which affects all system banks. Each bank then has 3 bank-specific registers to configure to individual bank needs.

The serial ROM configures memory as follows:

1. Global timing register (GTR)
2. Individual bank setup as required
   a. Bank timing registers (BTRs)
   b. Bank address mask register
   c. Bank configuration register

The two types of timing registers allow the memory controller interface of the LCA to be tailored to the type and speed of RAM chips used in the system.

The basic RAM configuration of the AXPvme system is banks of 4 Mbit DRAM chips with an access time of 70 ns. The number of banks, 2 or 4, is a function of the memory array installed.

For further details of these registers, see the LCA specification in Section 1.5.5.

1.3.4 Error Handling

There are two error registers associated with the memory interface of the LCA. These are the error status register and the error address register.

The error status register gives syndrome information associated with a memory interface error while the error address register freezes with the address of the access that caused an error.

For bit definitions and more information, see Section 5.6 in the LCA Specification Rev 2.1.

1.4 I/O Subsystem

The AXPvme I/O subsystem is based on the PCI bus. For full details of the PCI, refer to the PCI Specification V2.0. All of the module's I/O components are connected via the 32-bit, 5 V only, PCI implementation. The Low Cost Alpha has an integral PCI bridge interface to which each of the other devices connect.

The main elements of the I/O subsystem are:

- LCA<–>PCI bridge
- PCI central arbiter[1]
- VME interface
- SIO interface (giving connectivity to TOY, UART, ROM, and so forth.)
- Ethernet interface
- SCSI interface
- System interrupt controller[1]
- Mezzanine (PCI) connector

The maximum number of PCI devices supported for direct connection to AXPvme's PCI bus is six. They are the LCA, NI, SCSI, ISbus, VME adapters, and one other PCI option (which are supported via the mezzanine architecture). However, hierarchical configuration is supported to allow additional PCI devices connected over PCI-PCI bridges, and so forth.

1.4.1 PCI Addressing and Configuration

The base address of each PCI device, except the SIO, is programmable.

The individual base addresses are initialized by writing device configuration registers. The PCI defines a set of configuration registers for PCI compliant devices, which are accessible in PCI configuration space.

Within this special address space a fixed device select scheme is used to define the location of each device's configuration block.

The PCI defines a maximum size of 256 bytes for a configuration register block for any one device. The byte within the configuration area is defined by PCI_AD<7:0>, while PCI_AD<10:8> are reserved for configuration function codes. Thus, only the lowest 11 bits of the LCA's generated (and driven) 32-bit address are needed to select any location in a device's configuration region. In this way, address bits 11 to 15 can be used for a "no-logic" device select decode in AXPvme. In PCI configuration space, the seven supported devices can be accessed at the locations shown in Table 1–4.

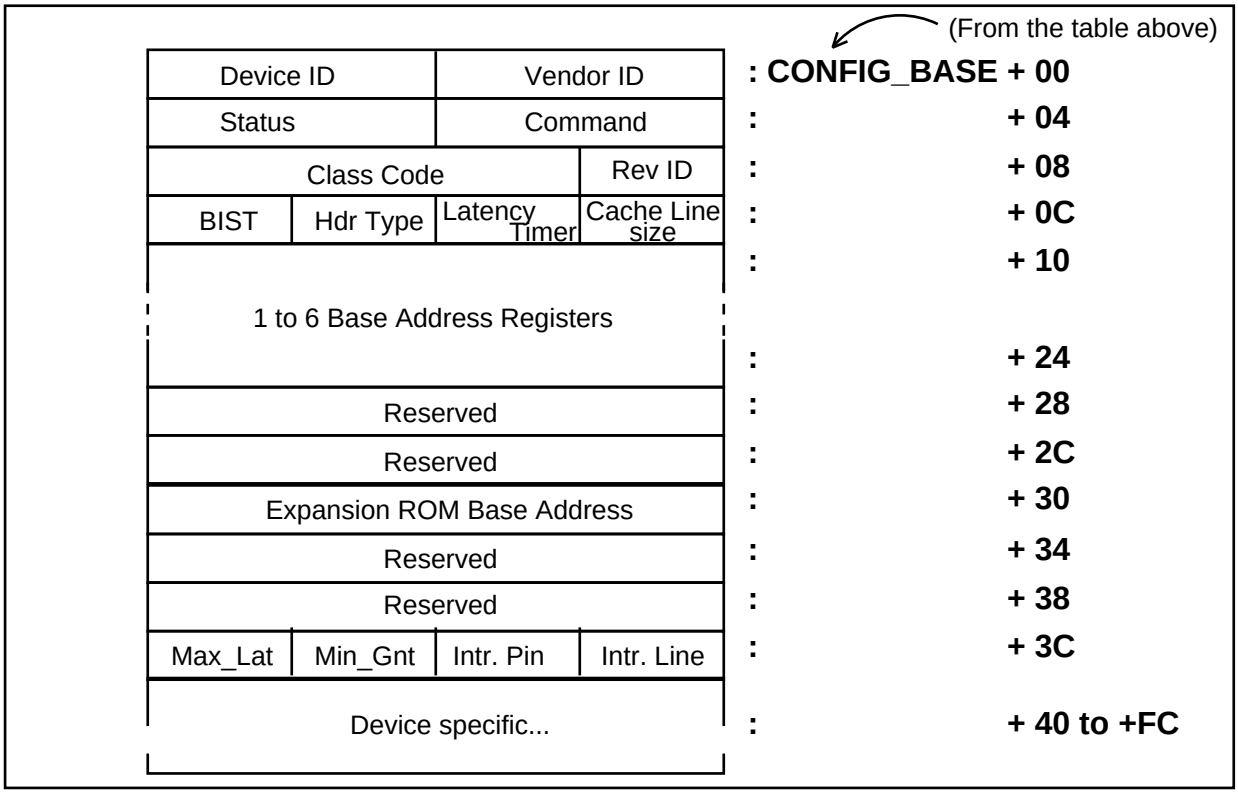
[1] The PCI arbitration and system interrupt controller are not PCI devices but do have registers that are accessed in PCI I/O space.

Table 1–4 Configuration Space Addressing

Device	Configuration Address	Notes
VME I/F	00000800 - 000008FF	4 base addresses
NI	00001000 - 000010FF	–
SCSI	00002000 - 000020FF	–
SIO I/F	00004000 - 000040FF	–
PCI Opt 1	00008000- ?	Option dependent
LCA	N/A	N/A

PCI configuration cycles can be driven from the LCA when the disable configuration bit in the IOC control status register is cleared. When this bit is cleared (for example, after reset), all accesses to normal PCI I/O addresses will map to configuration space cycles. Figure 1–4 shows the generic configuration space header.

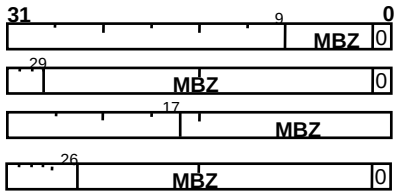
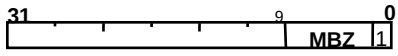
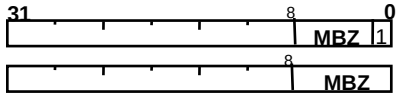
Figure 1–4 Generic Configuration Space Header



Configuration cycles with more than one of the address bits <11:15> set MUST NOT be made. There is no hardware mechanism to protect the user from the unpredictable results that this could cause.

Most of the devices that reside in PCI I/O in the AXPvme design have programmable base address registers that are set up in configuration space. Figure 1–5 shows each device and the layout of its base address registers.

Figure 1–5 PCI Devices, Size, and Base Registers

Device	Size	Mem	IO	Base Register
VME - Registers	512 bytes	✓		
- PCI->VME	512 Mbytes	✓		
- S/G RAM	128 Kbytes	✓		
- PCI->VME	64 Mbytes	✓		
NI - Registers	512 bytes		✓	
SCSI - Registers	128 bytes		✓	
- Registers	128 bytes	✓		

For further information about individual configuration registers, see the corresponding I/O device section.

1.4.2 PCI Arbitration

Arbitration for the PCI bus is centralized. A single bus arbiter monitors all "requests" for use of the bus and, based on some allocation scheme, decides on a bus winner to which it gives a "grant". The bus is owned by the winning device (once it takes up the grant by starting a cycle) until it finishes its cycle; that is, bus ownership is access based.

AXPvme's PCI arbitration logic is based on a fixed-priority allocation scheme. There is a small amount of programmability to the arbitration priority allocation, however, it is not intended as a general-purpose arbitration priority control mechanism. Currently, three priority schemes are supported based on the value written into the MOD_CNTRL_REG register bits <3:2>. These two bits will allow limited control over the PCI arbitration priority assignment based on the following scheme:

Priority	<3:2>=00	<3:2>=01	<3:2>=10	<3:2>=11
1	LCA	PCI	VME	Reserved
2	ENET	LCA	LCA	Reserved
3	SCSI	VME	PCI	Reserved
4	PCI	ENET	ENET	Reserved
5	VME	SCSI	SCSI	Reserved

Bus arbitration can also be tuned by modifying the way in which the various devices use their bus grants.

If the PCI device is programmed with a maximum data burst length, the duration for which the device will hold the bus is limited by a maximum data transfer. After the defined amount of data is transferred, the bus owner will give up the bus. Note that even with a programmed maximum bus burst length, there is no guarantee how long the bus owner will hold the bus. However, good device design should minimize holding of the bus when a master does not have data to transfer.

The second PCI mechanism for limiting the ability of one device to control the bus is the latency timer scheme.

Table 1–5 PCI Device Arbitration Control

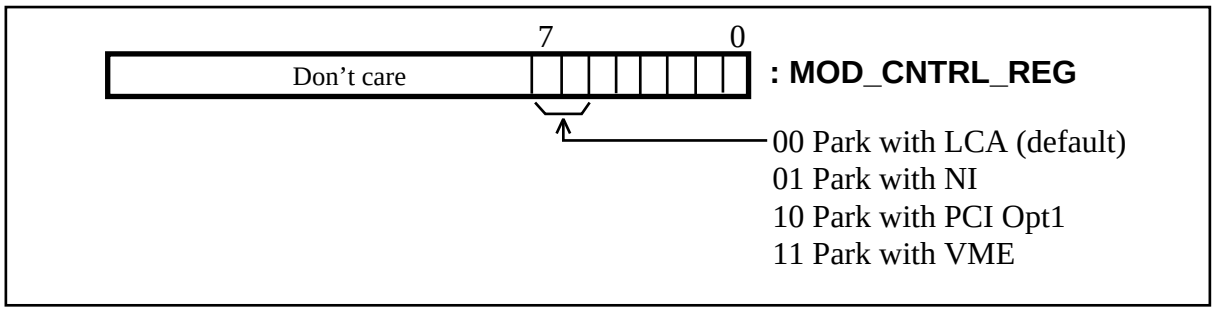
Priority	Device	Programmable Burst	Latency Timeout	Bandwidth
6 (highest)	LCA	Max=Quadword	No	?
5	NI	Yes	Yes	2 Mbyte/s
4	SCSI	Yes	Yes	5 Mbyte/s
3	PCI Opt 1	?	?	?
2	VME I/F	Max=2 Quadwords	Yes	>24 Mbyte/s

A latency timer value in the PCI configuration space of a device defines a minimum guaranteed bus tenure (measured in PCI cycles following the assertion of Frame#) beyond which the removal of the grant signal forces the device to end its access after the current data transfer completes.

AXPvme's fixed-priority arbiter understands the operation of the PCI latency timer mechanism. The bus is granted to the highest priority requester but once the winner starts its cycle, the arbiter will rearbitrate on the bus. The new winner cannot and will not (PCI specification) use the bus until the current owner concludes its cycle. This mechanism of reevaluating the bus winner during an active PCI cycle allows for full and efficient use of a device-based latency timeout scheme.

The arbiter will support "parking" the idle PCI bus with any one of the LCA, the VME interface, the NI, or PCI option 1. Selecting the device with which to park is programmed via two bits in the module control register (see Figure 1–6).

Figure 1–6 Module Control Register (PCI Park)

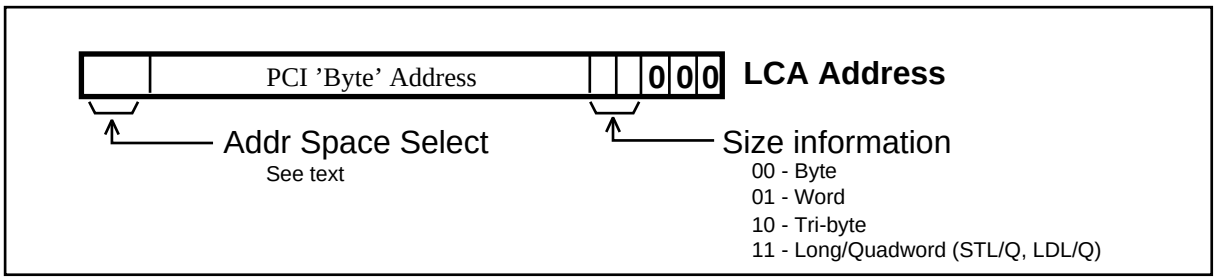


1.4.3 PCI Transfer

The Alpha AXP architecture does not currently support transfers smaller than longword quantities. The PCI devices often require transfers of bytes and words rather than full 32-bit longword data. To this end, the LCA employs a mechanism for reading and writing arbitrary bytes and words when accessing PCI space. See DECchip 21066-AA Rev. 2.1 Section 6.2.6.1 for a better description of sparse space addressing.

This mechanism allows I/O addresses to be formed with encoded information pertaining to the number of bytes and byte offset. In particular, the LCA I/O address has the form shown in Figure 1–7.

Figure 1–7 LCA I/O Addressing



The LCA operates as a PCI master when the CPU executes a load or store instruction that addresses a PCI peripheral space The LCA’s IOC provides address windows to the CPU that allow access to the memory, I/O, and configuration address spaces of PCI. The IOC also provides a register that, when read, will generate a PCI interrupt acknowledge cycle, or when written, will generate a PCI “special cycle”.

The address ranges for the various PCI address spaces supported are listed in Table 1–6.

Note

Throughout the text, I/O and memory addresses on the PCI are specified. The addresses described are the physical addresses that would appear on the PCI AD lines. If sparse space addressing is needed (less than longword data size), these address values should be inserted into the "PCI byte Address" field in the format above in order to form the CPU address.

Table 1–6 CPU Address Window to PCI

CPU Address <33:00>	Access	PCI Address Space
1 Axxx xxxx 1 Bxxx xxxx	Read Only	Interrupt Acknowledge
1 Axxx xxxx 1 Bxxx xxxx	Write Only	Special Cycle
1 Cxxx xxxx 1 Dxxx xxxx	R/W	I/O
1 Exxx xxxx 1 Fxxx xxxx	R/W	Configuration
2 xxxx xxxx	R/W	Sparse Memory
3 xxxx xxxx	R/W	Dense Memory

Due to the encoding of the CPU address used to generate PCI byte enables, only 27 physical address bits (CPU Addr<31:5>) can be used to generate the PCI address (AD<26:0>). This gives an effective PCI address space of 128 Mbytes. This 128 Mbytes is subdivided into a 16-Mbyte region and a 112-Mbyte region. The top 5 address bits of the PCI address (AD<31:27>) are generated differently, depending on which of these regions is referenced.

The bottom 16 Mbytes always map to the bottom 16 Mbytes of the PCI space, while the remaining 112 Mbytes can be mapped to the top 112 Mbytes in any naturally aligned 128-Mbyte window.

For this latter 112-Mbyte portion of the CPU to PCI window, the high-order 5 address bits are supplied from the Hardware Address Extension bits in the IOC_CNTRL register in the LCA (at CPU address 180000000h); that is, bits <31:27>. In other words, when addressing one of the PCI spaces described above,

```
if CPU Addr<31:29> = 000     then PCI AD<31:27> = 00000
                             else PCI AD<31:27> = HAE<31:27>.
```

1.4.3.1 Masked Transfers

Although the IOC supports arbitrary byte enables, only dual and tri-bytes that are contiguous and do not straddle a longword boundary are allowed.

To generate a masked PCI write transfer, always use an STL. This will become a single beat data burst on the PCI.

Either an LDQ or LDL instruction can be used to generate a masked PCI read. Again, the PCI burst length will be one.

1.4.3.2 Unmasked Transfers

Unmasked PCI write transfers can be generated by either an STL or STQ, with CPU address bits <4:3> = 11. The STL will produce a cycle of burst length 1 while an STQ will contain two data beats on the PCI.

Unmasked PCI read transfers can be generated by either an LDL or LDQ, with CPU address bits <4:3> = 11. The LDL will produce a PCI transfer of burst length 1 while an LDQ will produce a PCI transfer of burst length 2.

See Section 6.2, CPU Initiated PCI Cycles, in the LCA Specification for further details.

1.4.4 Main Memory as PCI Target

The I/O controller of the LCA allows a window from the 32-bit PCI memory address space to be mapped to a 34-bit address for system main memory.

PCI initiated masked reads are treated as unmasked reads by the LCA, while arbitrary byte enable combinations for PCI initiated writes are implemented by a read-modify-write operation in the 21066 memory controller.

Memory coherency is maintained for all PCI initiated transfers to cacheable memory.

Registers internal to the LCA are not accessible by PCI devices.

In order to allow the window from PCI to memory to exist, it is necessary to translate the PCI address to the CPU address. The IOC provides two programmable address windows that control access by PCI peripherals to system memory. These address windows are referred to as the PCI Target Windows. There is a set of three registers associated with each PCI Target Window–Window Base register, Window Mask register, and Translation Base register.

Bits <31:20> of the Window Base register specify a starting address in the PCI memory address space for the target window. Bits <31:20> of the Window Mask register provide a mask for base address matching between the PCI address and the Window Base register. A one in the Window Mask register means the corresponding address bit does not take part in the base address match. Thus, the size of the window is controlled by the mask register as shown in Table 1–7. Note also that a target window must always be naturally aligned.

Table 1–7 PCI Target Window Masking

WINDOW_MASK<31:20>	Size of Window
0000 0000 0000	1 Mbyte
0000 0000 0001	2 Mbytes
0000 0000 0011	4 Mbytes
0000 0000 0111	8 Mbytes
0000 0000 1111	16 Mbytes
0000 0001 1111	32 Mbytes
0000 0011 1111	64 Mbytes
0000 0111 1111	128 Mbytes
All others	Not supported by AXPvme

Note also that there are two control bits in <33:32> of the Window Base register. Bit<33> is a window enable bit. When cleared, the corresponding PCI target

window is disabled. Bit<32> is a S/G enable bit. When cleared, there is a direct mapping from the PCI memory address to the CPU system memory address. The LCA supports PCI—memory S/Ging when this bit is set. Refer to the LCA Specification for full details of this facility.

When direct translation is used, the high-order address bits of the CPU address formed are supplied from the window's Translation base register, while the lower order bits are passed from the PCI unaltered. The number of Translation register bits used depends on the window size.

For more information, refer to Section 6.3 of the LCA Specification.

1.4.5 PCI Address Space Layout

Using the PCI base address registers (see Section 1.4.1) in each of the PCI devices, the layout of PCI address space can be defined. Table 1–8 shows the PCI to CPU address translation.

Note that within a given PCI space (memory or I/O), devices must not be configured to overlap.

Table 1–8 PCI to CPU Address Translation

Window Size	CPU Address	Translation_Base Register Unused Bits MBZ
1 Mbyte	Trans_Base<33:20> : PCI_ADDR<19:0>	Trans_Base<19:9>
2 Mbyte	Trans_Base<33:21> : PCI_ADDR<20:0>	Trans_Base<20:9>
4 Mbyte	Trans_Base<33:22> : PCI_ADDR<21:0>	Trans_Base<21:9>
8 Mbyte	Trans_Base<33:23> : PCI_ADDR<22:0>	Trans_Base<22:9>
16 Mbyte	Trans_Base<33:24> : PCI_ADDR<23:0>	Trans_Base<23:9>
32 Mbyte	Trans_Base<33:25> : PCI_ADDR<24:0>	Trans_Base<24:9>
64 Mbyte	Trans_Base<33:26> : PCI_ADDR<25:0>	Trans_Base<25:9>
128 Mbyte	Trans_Base<33:27> : PCI_ADDR<26:0>	Trans_Base<26:9>
All others	Not supported by AXPvme	

1.4.5.1 PCI Memory Space

The flash ROM is fixed in the PCI Memory space. It takes up the first 1 Mbyte of the address map, from 00000000h to 000FFFFFh.

DC7407 Control and Status registers (including VIC64 registers) will be located starting at address 00100000h.

VME interface S/G RAM is defined to reside in PCI memory. This should be programmed to some convenient, naturally aligned 128-Kbyte block. This base address will be initialized by firmware and should not be altered thereafter. The recommended base address is 00200000h.

The 53C810 allows its register space to be accessed with PCI Memory space cycles. This register region is 128 bytes in extent. This base address will be initialized by firmware and should not be altered thereafter. The recommended base address is 00300000h.

The DECchip 21040-AA Control and Status registers are accessible through both PCI Memory and PCI I/O address space. The recommended base address in PCI Memory space is 00400000h.

The address space between 00500000h and 03FFFFFFh is defined as Sparse space available for PCI Mezzanine devices (approximately 59 Mbytes).

The remainder of LCA Sparse space, 04000000h through 07FFFFFFh, is defined as the DC7407 Sparse space VME window (64 Mbytes).

Address 08000000h defines the start of 384 Mbytes of Dense Memory space available for general use.

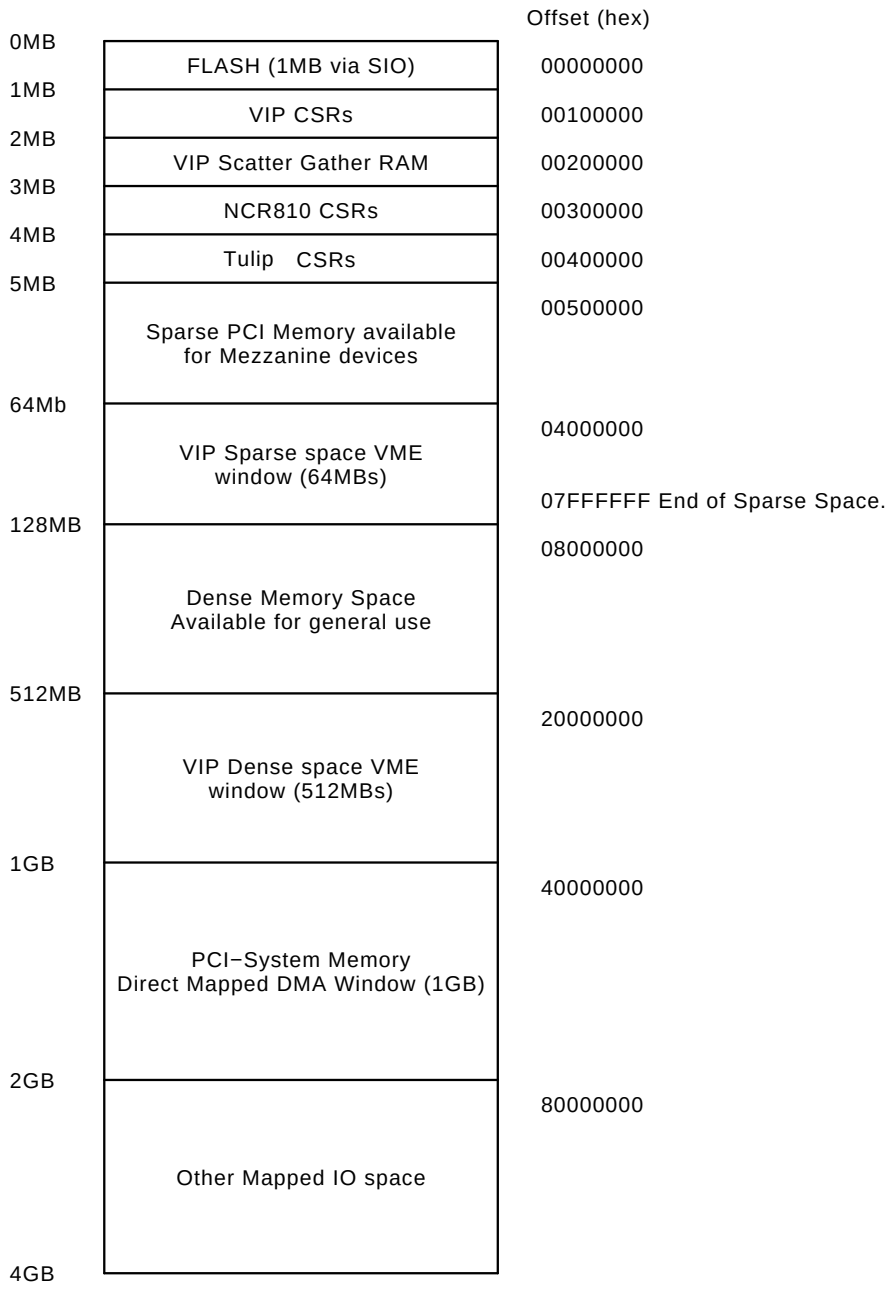
Address 20000000h defines the start of a 512 Mbyte DC7407 Dense Space VME window.

Address 40000000h defines the start of the 1 Gbyte PCI-System Memory Direct Mapped DMA window.

Address 80000000 defines the start of the 2 Gbytes of address space allocated to "other Mapped I/O space".

Figure 1–8 shows the system memory map for the PCI Memory space.

Figure 1–8 PCI Memory Space Address Map



1.4.5.2 PCI I/O Space

In the standard AXPvme system, some of the PCI devices have their register region accessible in the PCI I/O address space. These register blocks are small (less than 1 Kbyte each). It is recommended that they all be grouped locally within the lowest 16 Mbytes of PCI I/O space.

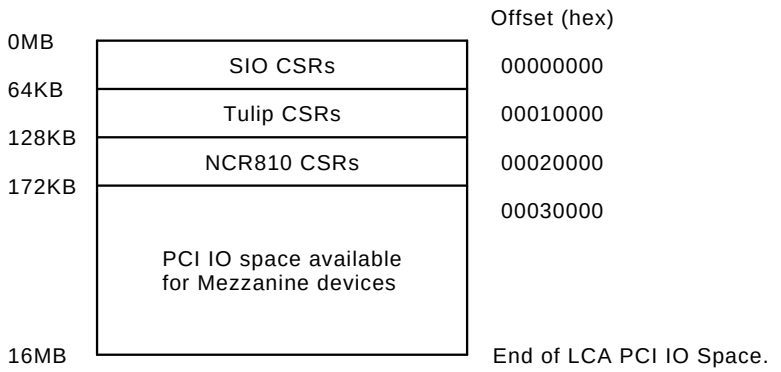
The SIO (82378) interface always occupies the lowest 64K of PCI I/O space from 00000000h to 0000FFFFh.

Firmware will initialize the base addresses of the various other PCI devices as follows:

DECchip 21040-AA	00010000h
NCR53C810	00020000h
PCI Mezzanine	00030000h

Figure 1–9 shows the system memory map for the PCI I/O space.

Figure 1–9 PCI I/O Space Address Map



1.5 VME Interface

The VME interface for AXPvme is designed to conform to the IEC 821, IEEE1014-1987, and D64 sections of IEEE1014 Rev.D (draft) standards. A full master/slave interface with bus arbitration and interrupt control is included. Addressing modes A16, A24, and A32 are supported for master transactions, while as a VMEbus slave, the interface responds to A24 and A32 modes and A16 for access to a number of interprocessor communications registers. Transfers are supported with D08, D16, D32, and block mode D64 data sizes. Flexible inbound and outbound S/G mapping functions are supported.

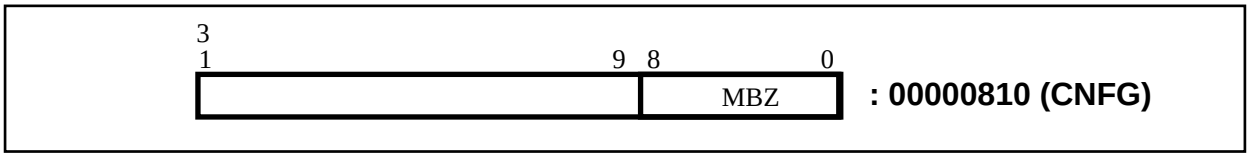
1.5.1 PCI Access to VME Interface

The PCI interface to VME must be configured, at startup, by writing three base address registers within the DC7407. A fourth register can be used to read the hardware setting for the second VME window if required. These registers are accessible only through PCI Configuration address space. Once these registers are initialized, PCI Memory space can be used to set up the remainder of the VME subsystem for access to VME devices. It is important to note that the windows defined by these registers must not overlap with each other. A brief description of these registers (and the region of address space they define) follows.

1.5.1.1 VME_CSR_BASE Register

PCI Configuration address space is used to write the VME_CSR_BASE register with the base address (see Figure 1–10) of a 512 byte window, in PCI Memory space, through which the DC7407, VIC64, and CY7C964 registers are accessed. This register is located at 00000810 in PCI Configuration Space (see Section 1.4).

Figure 1–10 VME_CSR_BASE Register



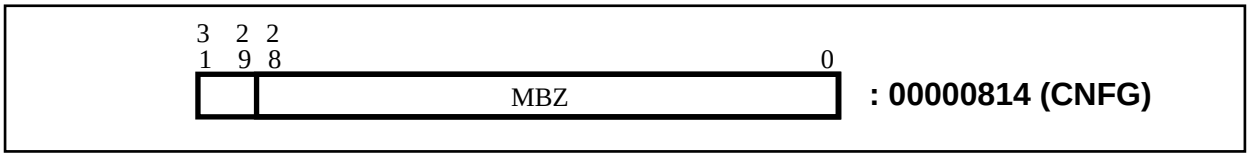
Only bits <31:9> of this register are writable as the register block aligns on a 512-byte boundary. Bits <3:0> are always read as 0, indicating that it is a PCI Memory region base address.

The VME interface registers themselves are described throughout the text, and in all cases, their locations will be specified as VME_CSR_BASE + xxxx. This represents their address in PCI Memory space.

1.5.1.2 VME_WINDOW_1_BASE Register

PCI Configuration address space is used to write the VME_WINDOW_1_BASE register with the base address (see Figure 1–11) of a 512-Mbyte window, in PCI Memory space, through which VME address space can be accessed. This register is located at 00000814 in PCI Configuration Space (see Section 1.4).

Figure 1–11 VME_WINDOW_1_BASE Register



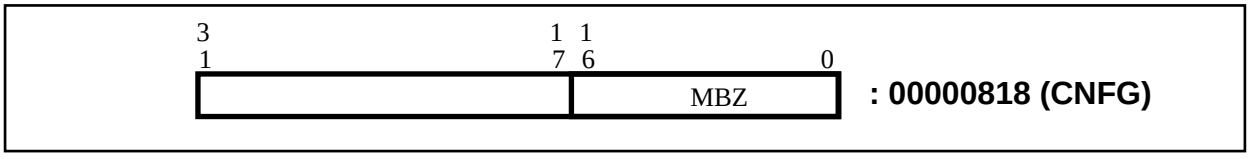
Only bits <31:29> are writable because the 512-Mbyte window must be aligned on a natural boundary.

The size of the PCI - VME window is selected by the state of the DC7407 “SPARE_5” pin. Driving this pin high selects a naturally aligned 512-Mbyte window while driving this pin low selects a naturally aligned 256-Mbyte window. AXPvme systems have this pin hardwired high.

1.5.1.3 VME_SG_BASE Register

PCI Configuration address space is used to write the VME_SG_BASE register with the base address (see Figure 1–12) of a 128-Kbyte window, in PCI MEM space, through which the S/G RAM can be accessed. This register is located at 00000818 in PCI Configuration Space (see Section 1.4).

Figure 1–12 VME_SG_BASE Register

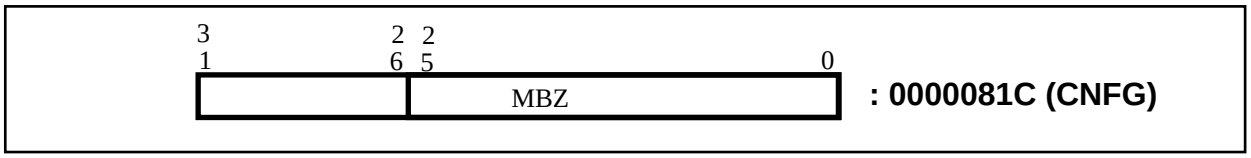


1.5.1.4 VME_WINDOW_2_BASE Register

PCI Configuration address space is used to write the VME_WINDOW_2_BASE register with the base address (see Figure 1–13) of a second window, in PCI MEM space, through which VME address space can be accessed. This register is located at 0000081C in PCI Configuration Space (see Section 1.4).

The size of this window is determined by the state of the two DC7407 pins (SPARE<4:3>). AXPvme systems have these pins hardwired to <11>, fixing this window size at 64 Mbytes (naturally aligned).

Figure 1–13 VME_WINDOW_2_BASE Register

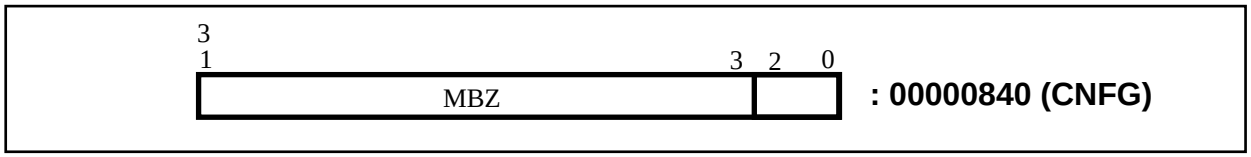


Only bits <31:26> are writable because the 64-Mbyte window must be aligned on a natural boundary.

1.5.1.5 VME_WINDOW_2_SIZE Register

PCI Configuration address space can be used to read the VME_WINDOW_2_SIZE register (see Figure 1–14) to size the region mapped by the VME_WINDOW_2_ BASE register. Bits <2:0> indicate the hardwired size of the VME_WINDOW_2. AXPvme systems will return <111> for bits <2:0>, indicating a 64-Mbyte window size. This register is located at 00000840 in PCI Configuration Space and is READ ONLY (see Section 1.4).

Figure 1–14 VME_WINDOW_2_SIZE Register



Only bits <2:0> are readable. The bits will encode the size of the VME_ WINDOW_2 region as follows.

VME_WINDOW_2_SIZE <2:0>	VME_WINDOW_2 Region Size
000	8 Mbytes
001	16 Mbytes
011	32 Mbytes
111	64 Mbytes

1.5.2 Master Operation

As a master, AXPvme has two windows onto the VME. The largest is a fixed size (512-Mbyte) window positioned in PCI Memory space by the VME_WINDOW_2_ BASE register. This address window is divided into 2048 x 256 Kbyte blocks each with its own programmable S/G entry.

The other is a 64-Mbyte window positioned in PCI Memory space by the VME_ WINDOW_2_BASE register. This corresponds to 256 x 256 Kbyte blocks each with its own programmable S/G entry.

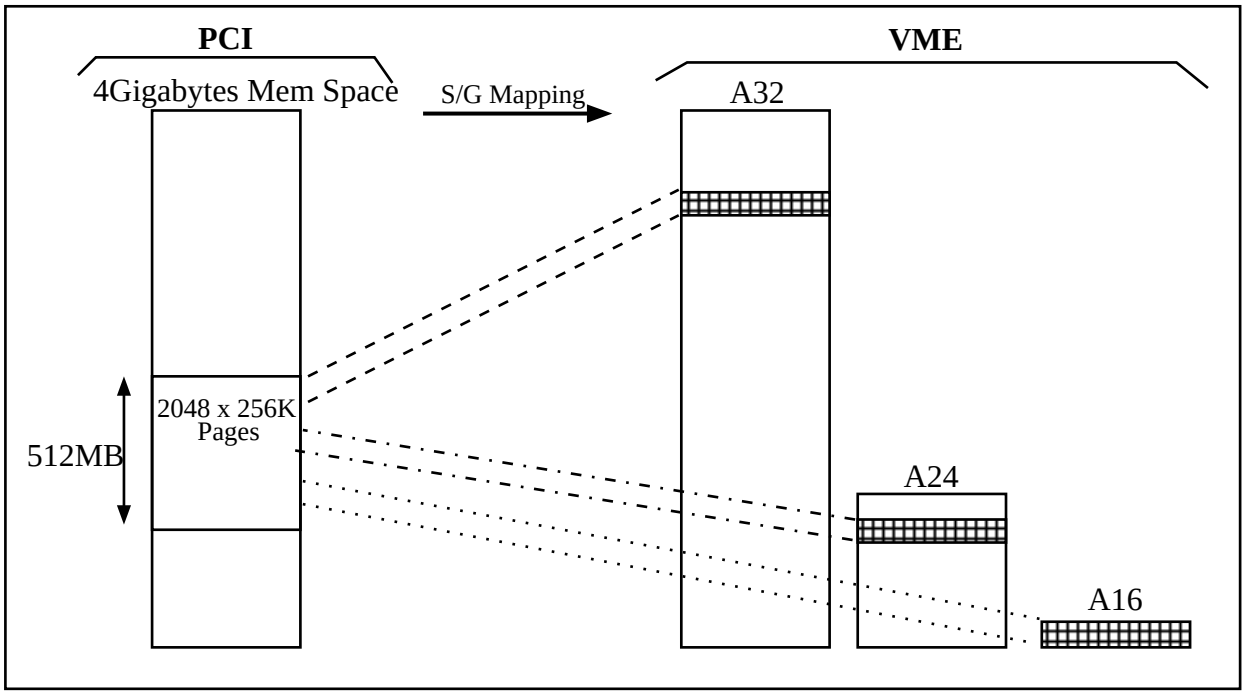
While each S/G entry maps a unique page within the VME_WINDOW_2 region, it also maps an overlapped page from the VME_WINDOW_1 region. For example, entry 5 of the outbound S/G RAM maps both page 5 of the VME_WINDOW_1 region and page 5 of the VME_WINDOW_2 region to exactly the same address on VME. Depending on the size of the VME_WINDOW_2 region, up to 256 outbound S/G entries will map "overlapped" PCI Memory address space. In the Alpha AXP architecture, this allows both "dense" and "sparse" space access to the same region of VME address space. The VME_WINDOW_2 region always maps to the bottom entries of the outbound S/G mapping registers.

In addition, each 256K page can be mapped to any one of the three VMEbus address spaces (A32/A24/A16). Numerous pages can be mapped to the same VME address to allow access to the same location with different modes (see Figure 1–15).

Access to A16, A24, and A32 spaces is supported in user and supervisor mode (the AM code used is fully programmable for each page).

The VME master interface operates in two ways - single and block transfers.

Figure 1–15 Example PCI to VME Mapping



1.5.2.1 Outbound S/G Mapping

The PCI to VME S/G entries, for both inbound and outbound accesses, must be read/write accessible in order to configure the VME interface for operation. The details on programming these S/G entries are described elsewhere in this chapter. The S/G RAM is an 32K x longword block in memory space (although only the top 27 bits are read/writable, the remaining 5 bits are MBZ).

The outbound S/G entries control and map all master accesses from AXPvme onto the VMEbus. Each entry defines the mapping for a 256-Kbyte page.

A PCI Memory access in either of the VME_WINDOW_1 or VME_WINDOW_2 regions (that is, PCI Mem address bits <31:29> match the VME_WINDOW_BASE address register), will cause a S/G lookup.

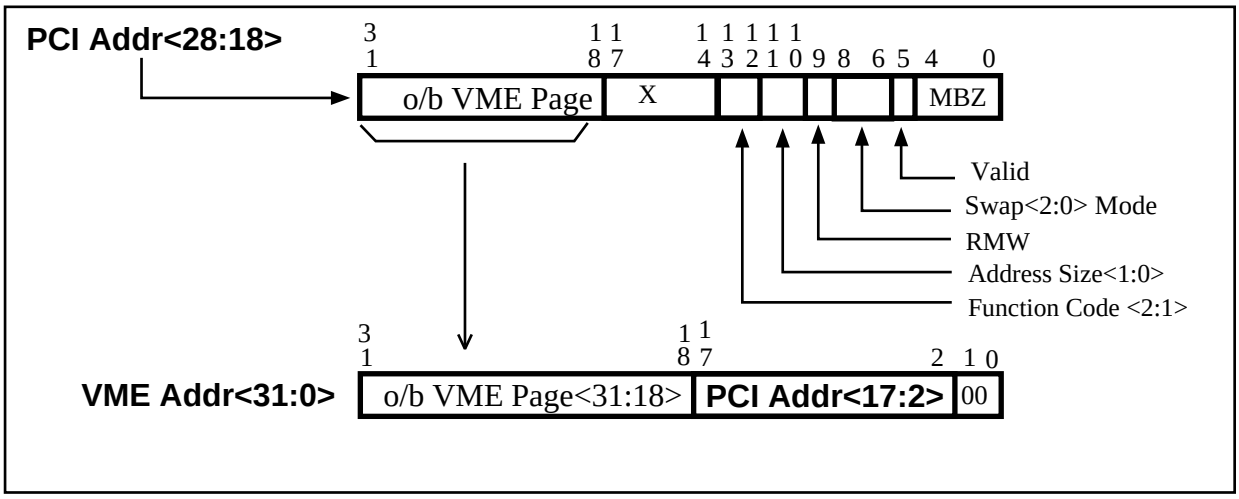
Bits <28:18> of the PCI address will specify the 256-Kbyte page involved and so identify the correct S/G entry, while PCI address <17:2>, along with PCI byte enables, specifies the byte address within that page.

If the VALID bit (bit <5>) in the S/G entry is not set, indicating an invalid entry, then no VMEbus transaction will take place. Instead, an error bit in the VIP_ BESR will be set. If a corresponding bit is set in the VIP_ICR register, this event will cause a DC7407 interrupt assertion.

If the entry is Valid, the interface will proceed to make the VMEbus access.

Figure 1–16 shows an Outbound S/G entry and how the VMEbus address is formed from the page information and the PCI address.

Figure 1–16 Outbound S/G Entry



Address Modifier S/G Field

The address modifier used in the master VME transfer is derived from the address size (ASIZ) and function code (FC) fields in the S/G entry. These ASIZ and FC fields map directly to the VICs "ASIZ" and "FC" inputs. Table 1–9 shows the use of these fields.

Table 1–9 Formation of AM Codes from S/G Entry

ASIZ1/0	FC2/1	Blk Mode	Operation	AM<5:0>
01 (A32)	00	No	User Data	09h
	01	No	User Program	0Ah
	10	No	Supervisory Data	0Dh
	11	No	Supervisory Prog	0Eh
	0x	Yes	User Block	0Bh (D64 08h)
	1x	Yes	Supervisory Block	0Fh (D64 0Ch)
11 (A24)	00	No	User Data	39h
	01	No	User Program	3Ah
	10	No	Supervisory Data	3Dh
	11	No	Supervisory Prog	3Eh
	0x	Yes	User Block	3Bh (D64 38h)
	1x	Yes	Supervisory Block	3Fh (D64 3Ch)
10 (A16)	0x	No	User Access	29h
	1x	No	Supervisory Access	2Dh
00	–User–	–Defined–	–AM codes–	VIC_AMSR

Byte Swapping

See Section 1.5.6 for a full description of the Swap field.

VMEbus Read-Modify-Write

When the RMW bit (read-modify-write bit) is set in a S/G entry, any master access to that page will set up the VME interface to do the next two accesses as a single indivisible sequence of VMEbus cycles (the access that involved the S/G entry with RMW set and the next access). The VMEbus ownership will be acquired for the current access and will be held until another master operation is done by the processor. This is particularly aimed at doing atomic VMEbus read-modify-write cycles.

For this RMW function to operate correctly, the VIC Interface Configuration Register must be programmed with VIC_ICR<7:5> = 001. A value of VIC_ ICR<7:5> = 000 disables the RMW mode regardless of the setting in S/G, while any other VIC_ICR<7:5> value will give UNPREDICTABLE results.

The VIC64 "Bus Capture and Hold" mechanism can be used to form indivisible sequences of VMEbus cycles.

1.5.2.2 Programmed I/O - Single VME Accesses

Single D08, D16, and D32 VME transfers are executed by individual accesses to either of the two VME window regions in PCI MEM space. The data size for the VME transfers are derived from the byte enabling of the corresponding PCI cycle.

1.5.2.3 Master Block Mode

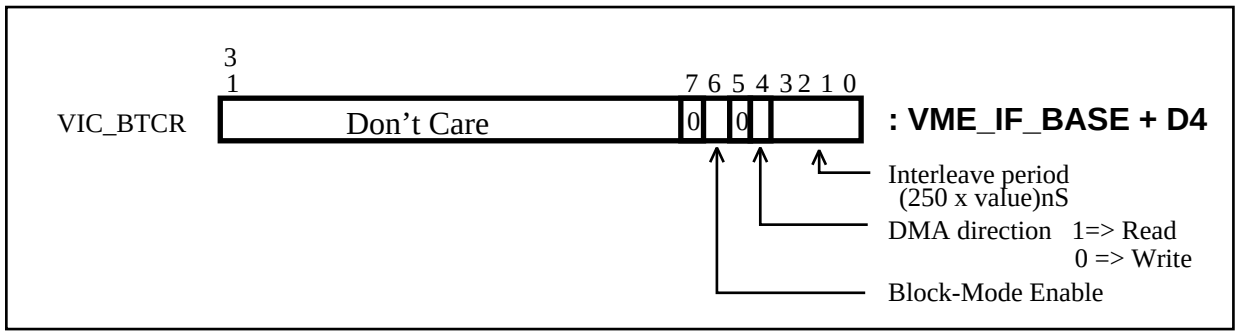
A block-mode DMA engine in the VME interface can be programmed to transfer up to 64 Kbytes without processor intervention in D16, D32, or D64. The interface fully handles the segmentation of the transfer so as not to violate the VMEbus specification in relation to crossing VME 256-byte boundaries for D16 and D32 or 2-Kbyte VME boundaries for D64.

Some minor restrictions will apply to master block-mode transfers.

- MBLT D64 transfers that do not start on naturally aligned 2K boundaries on the VME require some special care. Essentially, if 2 Kbyte boundary crossing is enabled (VIC_BTDR <7> = 1), the VME starting address MUST be aligned to a 2 Kbyte boundary.
- The PCI address must not cross a 64 Kbyte aligned boundary. This must be taken into consideration when working out the size of a block transfer (but is handled by the O/S DMA interface, if you are using an O/S).

Because the VMEbus specification prohibits crossing any 256/2K byte boundaries, any DMA must split into a number of bus transfers. At the interval between these transfers, the VME interface can be programmed to wait a period before rearbitrating for the VMEbus and proceeding. This delay gives slave access to the AXPvme the opportunity to complete during a block-mode transfer. This Interleave Period is programmable in the VIC Block Transfer Control Register register (see Figure 1–17).

Figure 1–17 VIC Block Transfer Control Register (VIC_BTCR)



The transfer burst length on the VMEbus can be programmed to be less than the maximum 256/2K burst via the VIC_RCR register (see Figure 1–19). The burst length field is the number of data transfers done in any given burst on the VMEbus, that is, a value of 4 in this field means in D16 that 4 words are transferred, while in D32 it would mean 4 longwords. For D64 block-mode operation, the burst-length figure is multiplied by four to give the maximum number of data transfers before giving up the bus. This means a maximum burst length value of 64 allows 256 (64x4) transfers of D64 data, which is 2048 bytes.

The operation on the PCI bus during an MBLT transaction is much the same as for a block mode slave operation, which is described in Section 1.5.5.

An MBLT transfer setup involves defining the data size, the transfer direction, transfer length in bytes (must be even as D08 block mode is not supported), and the source and destination addresses. PCI Memory to VMEbus address mapping is handled as usual through the S/G, however, the address modifiers in the mapping entry are automatically transformed to generate the block-mode version of the AM code specified (except for user-defined address modifier codes).

The setup sequence for a master DMA is given in the list below.

1. Write DMA transfer length to the VME Byte Length Registers, VIC_BTLR0, VIC_BTLR1[1]. D64 BLT operations will be distinguished by a write to the VIC64's BTDR register bit 4.
2. Write the DMA direction bit (read/write) and DMA enable bit to the VIC_ BTCR.
3. Write to the desired PCI MEM address (that will map to the target VMEbus address) with PCI start address as the write data.
4. Clear the DMA enable bit in the VIC_BTCR to allow normal master operation.
5. Wait for completion notification (no other VME master operations allowed during DMA).

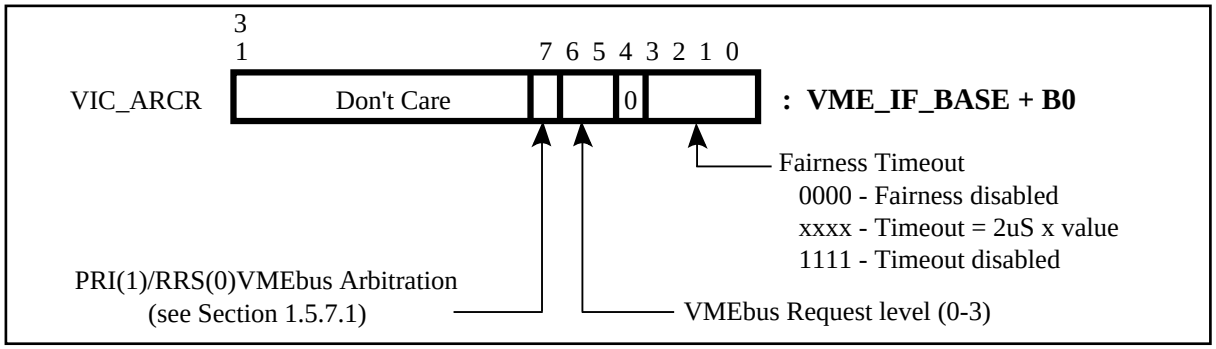
As mentioned in step 5 above, notification of DMA completion is via a DMA completion interrupt, which is enabled in the VIC_DMAICR and whose vector is generated by the VIC_EGIVBR.

[1] PCI Deferred Writes can be enabled to decouple the CPU from the holdups on the "local-bus" when setting up DMAs.

1.5.3 VMEbus Requester

When AXPvme wishes to act as the VMEbus master, the VME interface must request ownership of the bus. Controlling the manner and level of the bus request is achieved using the VIC Arbiter/Requester Configuration Register, shown in Figure 1–18.

Figure 1–18 VIC Arbiter/Requester Configuration Register (VIC_ARCR)



MR-6383-AI

1.5.3.1 VMEbus Request

Level

The level at which AXPvme requests the bus is set by writing the VMEbus Request Level field. See Section 1.5.7.1 on VMEbus arbitration for details of how bus request levels are used by a system controller.

Fair Request Timeout

Master grants at any given level are passed down the VME along a daisy-chain. If there are multiple requesters on the bus, masters further down this grant daisy-chain can experience bus starvation. To minimize this problem a Fair Request scheme can be implemented by bus masters (note, if any one master does not obey the fairness scheme, then it can starve the others).

When operating in fair request mode, AXPvme does not request the bus if anyone else is requesting the bus. This fair approach to asserting the request line is adhered to for the Fairness Timeout period, after which the request is asserted regardless. In this way there is a standoff period, which allows everyone along the grant chain the opportunity to win the VME.

This fairness request scheme is disabled in AXPvme by writing a value of 0 to the Fairness Timeout field. Nonzero values will set the number of 2 µs intervals to be used for the fairness timeout, while a value of F will disable the fairness timeout, that is, AXPvme will only request the bus if nobody else wants the bus.

1.5.3.2 VMEbus Release

Once AXPvme has acquired bus mastership, it is important to control the manner in which this ownership will be relinquished. Four bus release modes are supported by AXPvme. The release mode is configured in the VIC Release Control Register (VIC_RCR).

Release-On-Request (ROR)

In this release mode, AXPvme will retain bus ownership after completion of the cycles for which it requested ownership, until another device requests the bus. Then AXPvme will release the VMEbus.

Release-When-Done (RWD)

When set up for RWD, AXPvme will release the bus immediately after completion of the cycles for which it requested ownership.

Release-On-Clear (ROC)

In this mode, AXPvme will retain the ownership of the bus after completion of the cycles for which it requested ownership, until the system controller asserts the Bus Clear signal.

VMEbus Capture and Hold (BCAP)

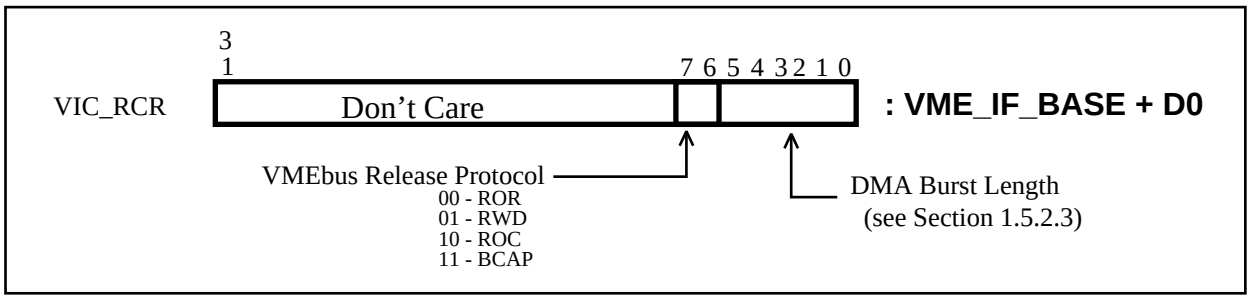
Putting AXPvme into this mode claims the VMEbus for itself for as long as the BCAP mode is selected. The VMEbus is only released when the AXPvme release mode field is reprogrammed to ROR, RWD, or ROC.

RMC

In addition to these four bus release modes, there is the use of the S/G Read-Modify-Write bit, which forces AXPvme to hold ownership of the VMEbus for two accesses before releasing in the programmed ROR, RWD, or ROC fashion. See Section 1.5.2.1 for details.

Control over the release modes of AXPvme is via the VIC Release Control Register (VIC_RCR, offset D0).

Figure 1–19 VIC Release Control Register (VIC_RCR)



1.5.4 Slave Operation

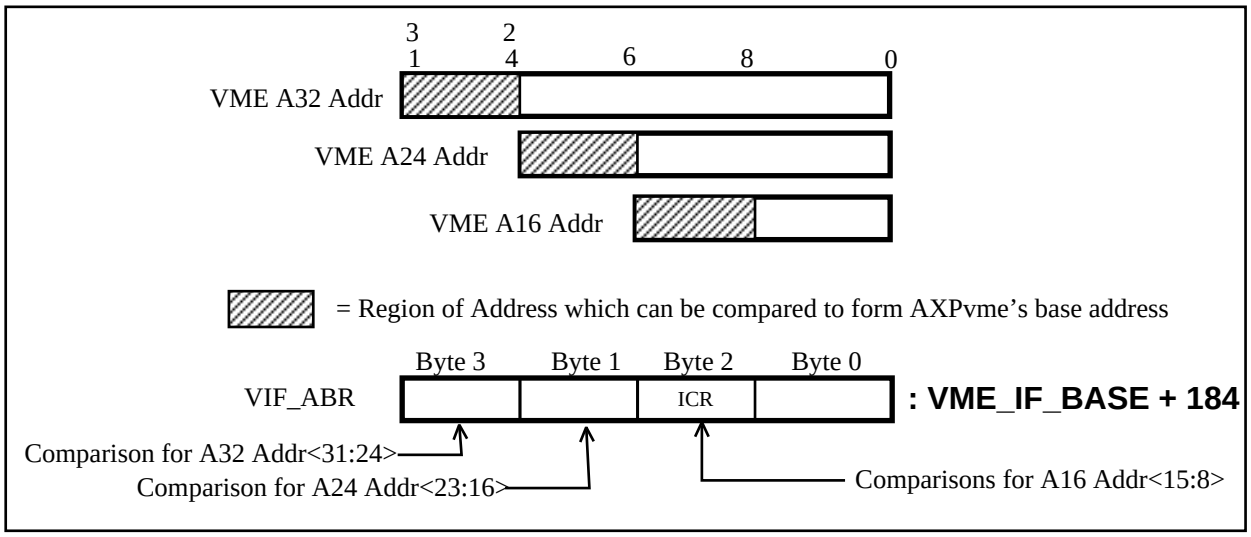
As a VME slave, the AXPvme system will respond to A24 and A32 access. In addition, a small number of bytewide interprocessor communications registers are accessible in A16 space.

Incoming slave accesses are mapped and controlled by two incoming S/G maps. In A32, AXPvme occupies up to 128 Mbytes mapped by 16384 S/G entries, each mapping an 8-Kbyte page. In A24, AXPvme occupies up to 16 Mbytes mapped by 2048 S/G entries, each mapping an 8-Kbyte page.

1.5.4.1 A32, A24 VMEbus Address Decode

The VME address decode (see Figure 1–20) is implemented using the CY7C964 elements in the interface. Three CY7C964 elements are accessed together in the longword of VIF_ABR. This register must be accessed as a longword even though the individual bytes represent address match data for separate VME address spaces.

Figure 1–20 Address Decode



Associated with each of the top three comparison bytes is a bit mask to control the number of bits that will be checked during a VMEbus address match. These mask bits are contained in the VIF_MASK register, at VME_IF_BASE + 180. If a bit is set, then the corresponding address to base register bit is not used in the address comparison.

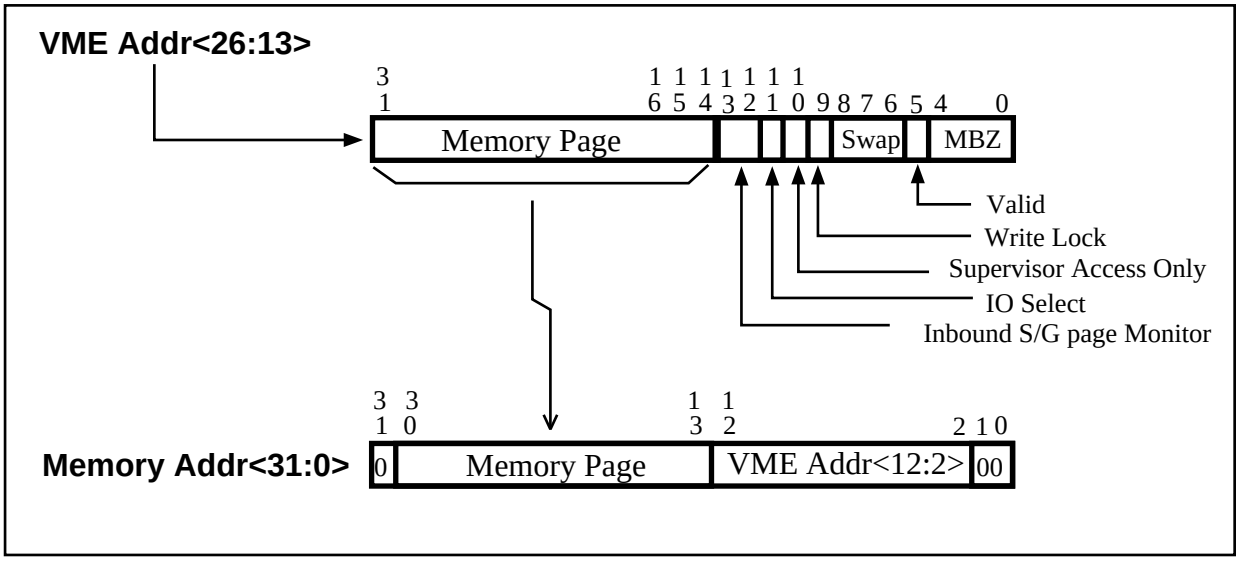
Note that for normal AXPvme operation, at least the top 5 bits of the A32 address match byte must be used for matching.

Bytes 1–3 of the base register and mask register will be contained in CY7C964 elements. All 3 bytes of these registers must be written simultaneously. Note that byte 0 is not used and will not affect address recognition. See CY7C964 specification for more detail on the comparison and mask registers.

1.5.4.2 Inbound S/G Mapping

The inbound S/G RAM format is shown in Figure 1–21.

Figure 1–21 Inbound S/G Entry (With A32 Addressing Example)



Byte Swapping

As for outgoing data, a comprehensive set of hardware byte-swapping can be programmed via the S/G RAM.

Page Protection

S/G control can limit slave accesses to read-only (page can be write-locked) and can further restrict access to supervisory cycles only.

Memory or I/O Target

More often than not, VME slave accesses will be directed at main memory. To facilitate this, the IOC in the LCA chip will be configured to map PCI memory cycles to main memory (see Section 1.4). In these cases, when the VME interface requests ownership of the PCI and gets a bus grant, it will transfer the VME data to the mapped main memory address using a PCI memory cycle. It may be desirable in some instances to allow a VME device access to the AXPvme as a slave in order to use one of its I/O resources. To facilitate this function, setting the I/O select bit in the incoming S/G map forces a PCI I/O cycle rather than a Memory cycle.

Note that bit <31> of the incoming PCI address is always filled with "0" (forcing access to the lower 2 Gbytes of PCI memory space) and bits <1:0> are padded with "00" (PCI uses C/BE to specify which bytes are being accessed).

Configuration cycles are never initiated by the VME interface.

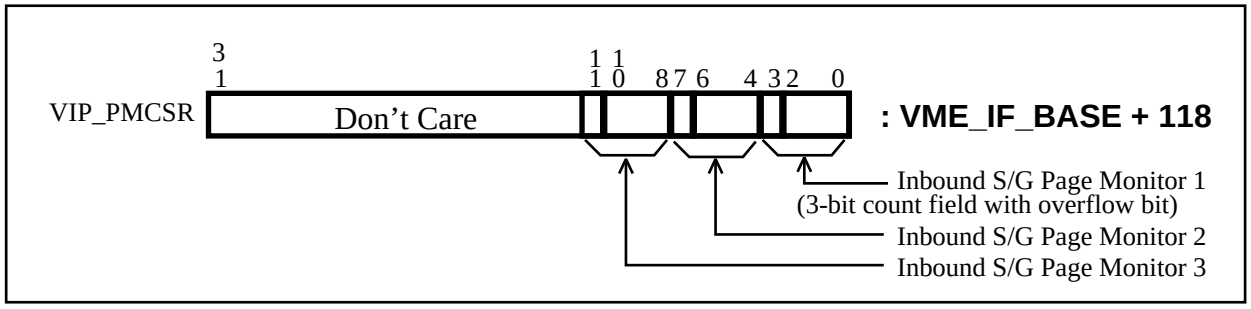
1.5.4.3 Inbound S/G Page Monitor

The Inbound S/G page monitor facility allows AXPvme to watch accesses to selected Inbound S/G pages. When the Inbound S/G page Monitor field in the incoming S/G is zero, no monitoring function is performed on that page.

There are three separate Inbound S/G page monitor counters implemented in the AXPvme VME interface.

By setting the two-bit Inbound S/G page monitor field in a given S/G entry to one, two, or three, an access to that page will cause the corresponding Inbound S/G page monitor counter to increment. The Inbound S/G page monitor counters are readable in a VME interface register (VIP_PMCSR), shown in Figure 1–22.

Figure 1–22 Inbound S/G Page Monitor Control/Status Register



An overflow of any Inbound S/G Page Monitor Counter will cause a corresponding bit in the VIP_BESR to be set. If enabled (via the VIP_ICR), this condition will cause the VIP_LIRQ <0> interrupt to be asserted at VIC LIRQ <2> (and ultimately, the LCA) .

1.5.5 Programming S/G RAM

S/G RAM is not initialized by hardware and will come up in a random state. Firmware should initialize this area to a reasonable default state before using the VME subsystem. Operating system drivers should also initialize this area on startup.

The 8K S/G longword entries are in three regions:

- 2048 A24 inbound entries at VME_SG_BASE + 10000h, each mapping an 8K page of A24 VME address space to PCI address space
- 16384 A32 inbound entries at VME_SG_BASE, each mapping an 8K page of A32 VME address space to PCI address space
- 2048 outbound entries at VME_SG_BASE + 1E000h, each mapping a 256K page in PCI Memory space onto the VME

The index to the A24 inbound S/G entry for A24 accesses is formed by VME A24 address bits <23:13>.

The index to the A32 inbound S/G entry for A32 accesses is formed by VME A32 address bits <26:13>.

The index to the outbound S/G entry for master accesses is dependent upon which region was used for the access (and the size of the VME_SUB_WINDOW region).

- VME_WINDOW region accesses generate an outbound S/G entry from PCI address bits <28:18>.
- An access through a 64MB VME_SUB_WINDOW region uses PCI address bits <25:18>.
- An access through a 32MB VME_SUB_WINDOW region uses PCI address bits <24:18>.

- An access through a 16MB VME_SUB_WINDOW region uses PCI address bits <23:18>.
- An access through an 8MB VME_SUB_WINDOW region uses PCI address bits <22:18>.

The S/G RAM is fully programmable over the PCI. The mapping of the S/G RAM takes up 128 Kbytes of PCI memory space,and has its own base address.

The S/G RAM can be programmed independently of master or slave VME activity.

1.5.6 Byte Swapping

The byte swapper function is programmed via the swap bits [SWP] in the S/G entries. SWP[1:0] define the swap mode 0 through 3 and SWP[2] enables D64 swapping, which is only used in D64 BLT transfers.

Mode 0: No Swap. There is no byte swap, and in transferring bytes from the little endian PCI to the big endian VMEbus, the address of any byte as seen on the two buses will remain the same.

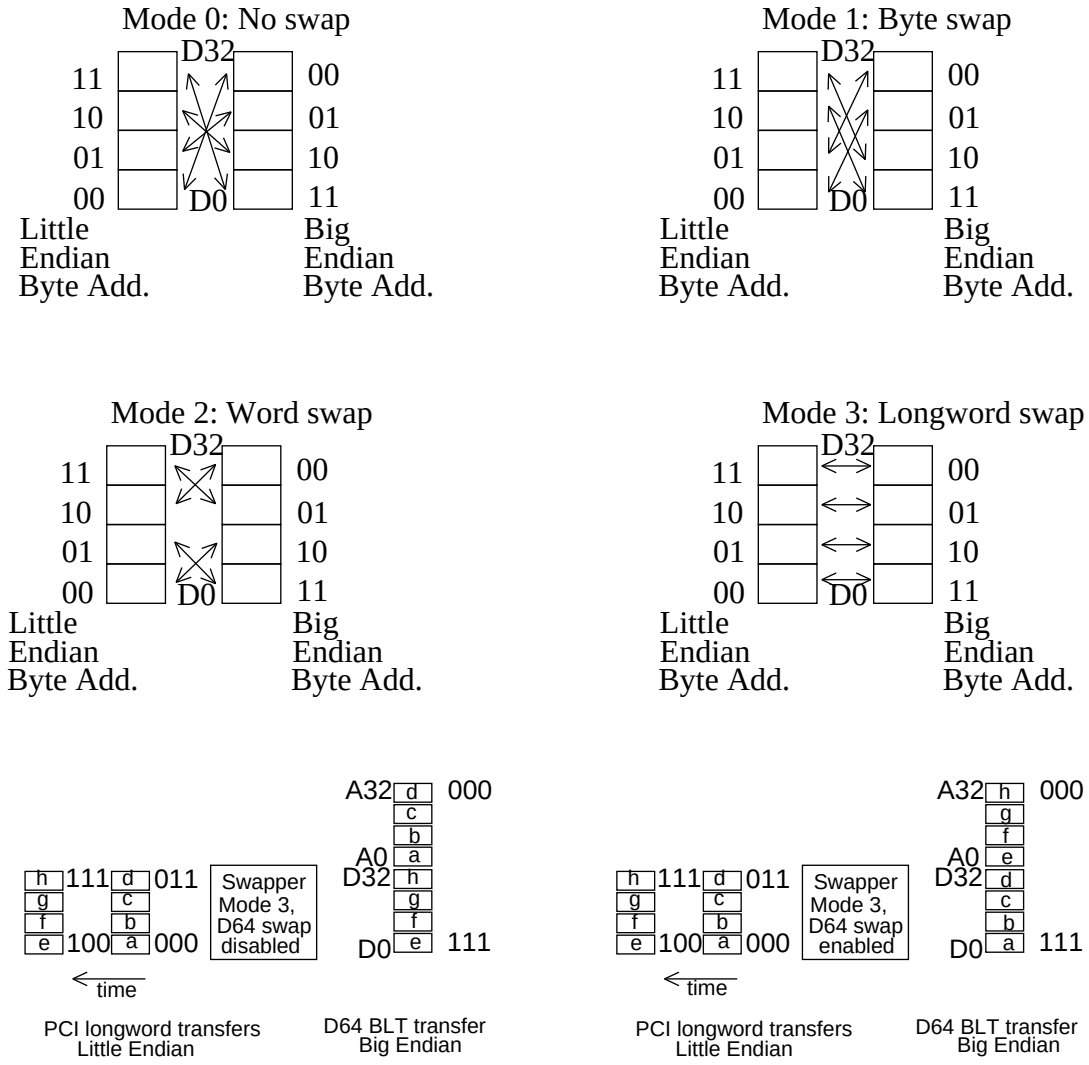
Mode 1: Byte Swap. The bytes within words are swapped.

Mode 2: Word Swap. The words within longwords are swapped.

Mode 3: Longword Swap. Basically, a combination of modes 1 and 2. The result is that byte 11 in a longword becomes byte 00 when swapped, 10 becomes 01, 01 becomes 10, and 00 becomes 11.

The D64 swap enable bit is used only in D64 slave or master BLT transfers and swaps the order that the longwords are taken from or put into memory over the PCI. When enabled with mode 3 swap, this means that byte 000 in a quadword becomes byte 111 (that is, the binary byte address is inverted). The 4 swap modes are described in Figure 1–23 with the D64 swap cases illustrated for the mode 3 case.

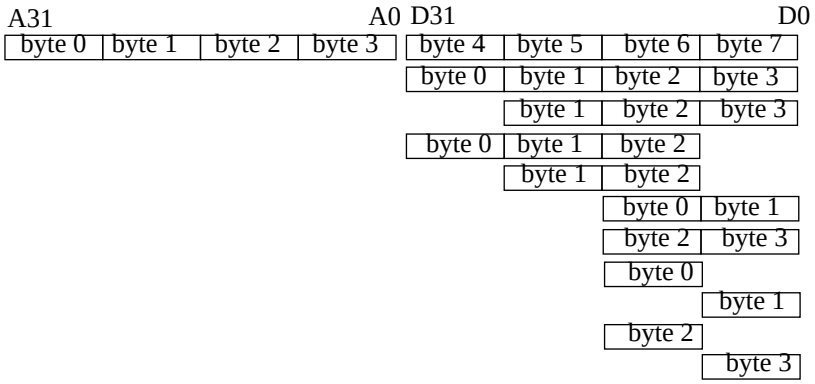
Figure 1–23 DC7407 Swap Modes



D64 swap illustrated in combination with mode 3 longword swap.

There is still another aspect of byte or lane swapping that is implemented external to the DC7407 by the VIC64. When transfers less than complete longwords are done onto or from the VMEbus, the data must be driven onto certain VMEbus lanes depending on the data width. This is described in Figure 1–24.

Figure 1–24 Big Endian VME Byte Lane Formats



D64 transfers are a particular case here and are handled by the VIC64 and the CY7C964 components. The DC7407 will transfer data only onto the VIC64 local data bus and, under the control of the VIC64, these are packed to form quadwords in the CY7C964s and onto the VMEbus as shown. Only full quadword transfers as BLTs are allowed in D64 mode.

The longword transfers, tri-byte transfers, and unaligned word transfers all use the byte lanes in the same way. However, when the low word in a longword is transferred, the data is switched to/from its usual lanes D[31:16] to/from D[15:0]. Byte transfers in the low word of a longword are swapped in a similar way. The DC7407 internally implements this word swapping logic.

Table 1–10 shows the local bus address and size signals used for the various swap modes when the DC7407 is master of the local bus. Those cycles where the DC7407, on the basis of the A1,0 and SIZ1,0 lines, moves the data to or from the D0-16 lane are marked with "L" in the adjacent column.

Those cycles which, because of the byte swapping, would result in a noncontiguous arrangement of bytes on the VMEbus, are not allowed and are target aborted on the PCI. PCI transfers with noncontiguous PCI byte enables are included in Table 1–10, although at this point the DECchip 21066 cannot generate cycles with noncontiguous PCI bytes enabled.

Table 1–10 PCI BE# to Local A1,0 and SIZ1,0 Translation for Various Swap Modes

PCI BE# <3:0>	Mode 0 No Swap A1,0	SIZ1,0		Mode 1 Byte Swap A1,0	SIZ1,0		Mode 2 Word Swap A1,0	SIZ1,0	Mode 3 Longword Swap A1,0	SIZ1,0
1111	No cycle			No cycle			No cycle		No cycle	
1110	00	01	L	01	01	L	10	01	11	01
1101	01	01	L	00	01	L	11	01	10	01

(continued on next page)

Table 1–10 (Cont.) PCI BE# to Local A1,0 and SIZ1,0 Translation for Various Swap Modes

PCI BE# <3:0>	Mode 0 No Swap A1,0	SIZ1,0		Mode 1 Byte Swap A1,0	SIZ1,0		Mode 2 Word Swap A1,0	SIZ1,0		Mode 3 Longword Swap A1,0	SIZ1,0	
1011	10	01		11	01		00	01	L	01	01	L
0111	11	01		10	01		01	01	L	00	01	L
1100	00	10	L	00	10	L	10	10		10	10	
1001	01	10		Noncontig			Noncontig			01	10	
0011	10	10		10	10		00	10	L	00	10	L
1000	00	11		Noncontig			Noncontig			01	11	
0001	01	11		Noncontig			Noncontig			00	11	
0000	00	00		00	00		00	00		00	00	
0101	Noncontig			Noncontig			Noncontig			Noncontig		
1010	Noncontig			Noncontig			Noncontig			Noncontig		
0110	Noncontig			01	10		01	10		Noncontig		
0010	Noncontig			01	11		00	11		Noncontig		
0100	Noncontig			00	11		01	11		Noncontig		

As a VME slave or during DMA driven BLTs, the VIC64 will drive the local bus address lines and the DC7407 will generate the byte-enable combinations to drive onto the PCI. As in the master case, some of these result in noncontiguous byte arrangements enables on the PCI side of the bus. These will be passed to PCI with the corresponding byte enables asserted. As can be seen in Table 1–11, the VME data for byte and aligned words will always be received on the data lines D[15:0]. The DC7407 will only support being set up as a D32 slave (VIC_ SS0CR0).

Table 1–11 Local Bus A1,0 and SIZ1,0 to PCI BE# Translation

Local Bus A1,0	SIZ1,0	VME Data		Mode 0 BE#	Mode 1 BE#	Mode 2 BE#	Mode 3 BE#
00	00	D[31:0]		0000	0000	0000	0000
00	11	D[31:8]		1000	0100	0010	0001
01	11	D[23:0]		0001	0010	0100	1000
00	10	D[15:0]	L	1100	1100	0011	0011
01	10	D[23:8]		1001	0110	0110	1001
10	10	D[15:0]		0011	0011	1100	1100
00	01	D[15:8]	L	1110	1101	1011	0111
01	01	D[7:0]	L	1101	1110	0111	1011
10	01	D[15:8]		1011	0111	1110	1101
11	01	D[7:0]		0111	1011	1101	1110

1.5.7 System Controller Operation

AXPvme can operate as a full VME system controller (in slot 1). As a system controller, the AXPvme provides the following functions:

- VMEbus arbitration control (driving BGIOUT*)
  - Priority (PRI)
  - Round-Robin (RRS)
  - Single-level (SGL)
- VMEbus interrupt control (driving IACK*)
- Driving SYSCLK
- Timeout timers for data transfers and arbitration

The system controller functions are controlled via a number of registers in the VME interface. Each of these byte registers is mapped into the lowest byte of an aligned longword in PCI Memory space.

AXPvme is selected as a system controller at powerup (global reset to the VME interface logic) by the state of a module DIP switch (E41 position 4 closed).

1.5.7.1 VMEbus Arbitration

The operation of AXPvme as a VMEbus arbiter is controlled by the VMEbus Arbiter/Requester Configuration Register (VIC_ARCR, offset B0). Figure 1–18 shows the layout of the register. The three arbitration schemes supported by AXPvme are achieved by setting the VME interface to arbitrate with a priority or round–robin scheme via bit 7 of the VIC_ARCR, in conjunction with the setting of request levels of the various VMEbus devices.

In PRI mode, BR3 requests have the highest priority while BR0 is the lowest.

In the RRS arbitration is assigned on a rotating basis—when the bus is granted to a requester on bus request line BR[n]*, then the highest priority for the next arbitration is assigned to bus request line BR[n-1]* (or BR3 if the previous level was BR0).

SGL arbitration is obtained by programming the VME interface for PRI bus arbitration and setting all requesters to the same bus request level.

1.5.7.2 VMEbus Interrupt Handling

For a complete description of the operation and handling of VMEbus interrupts, refer to Section 1.17.

As system controller, AXPvme drives the IACK daisy-chain. In response to a VMEbus IRQx* assertion, system software will normally initiate an IACK cycle on the VMEbus by reading the VIP_IRR register. Exceptions to this flow are made for the use of auto-vectored interrupts.

1.5.7.3 SYSCLK Output

AXPvme will drive SYSCLK from slot 1 as a system controller. The clock driven will be fixed 16 MHz with a nominal 50% (+/- 10%) duty cycle. Note this 16 MHz will have no fixed phase relationship with other bus timings.

1.5.7.4 Timeout Timers for Arbitration and Transfers

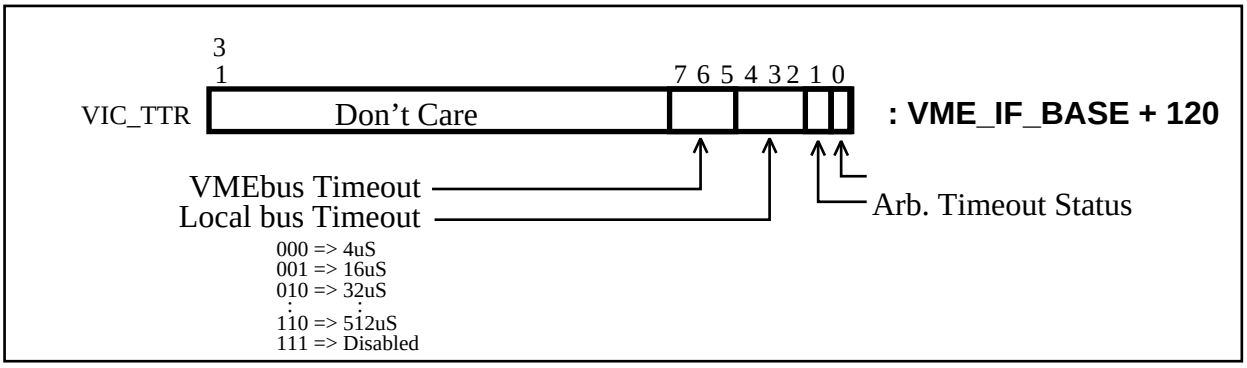
By default, as the VMEbus system controller, AXPvme will operate as an arbitration watchdog. After issuing a bus grant to the winning requester, the VME interface will monitor the bus and if it does not detect activity (BBSY* asserting) within 8 µs, it will terminate the bus ownership (by itself asserting BBSY*) to allow rearbitration on the VME.

This arbitration timeout on the VMEbus cannot be disabled, however, the condition can be used to generate a local interrupt to the LCA processor. Control of this interrupt is via the VIC_EGICR (see Section 1.17). Bit <1> in the VIC_ TTR, Figure 1–25, is set if the Arbitration timer expires.

When AXPvme is configured as the VMEbus system controller and the transfer timeout timer is enabled, the VME interface starts this timer whenever the data phase of a cycle is signaled (DSx* asserting). If the timer expires before the data cycle is acknowledged or completed in error, the system controller will flag a bus error (asserting BERR*). This condition sets a status bit in the VMEbus Error Status Register (VIC_BESR).

The transfer timeout is configured in the VMEbus Transfer Timeout Register (VIC_TTR, offset A0), shown in Figure 1–25.

Figure 1–25 VMEbus Transfer Timeout Register (VIC_TTR)



1.5.8 Interprocessor Communication

The VIC64 features five bytewide general-purpose interprocess communication registers (ICRs) and six access "switches", which are accessible over the VMEbus. These registers are also accessible in the normal VME interface register space mapped in PCI MEM space. When accessed over the VMEbus, they are located in A16 space by byte 1 of the VIF_ABR register.

1.5.8.1 Interprocessor Communication Registers (ICRs)

Five of these general-purpose registers are simply 8-bit read/write registers accessible both over the VMEbus and in local PCI Memory space. Two others allow VIC64 status and hardware revision information to be read over the VMEbus.

Bits <4:0> in the final register are set when there is a write access to the corresponding ICR. See the VIC64 specification for more complete details.

1.5.8.2 Interprocessor Communication Global Switches (ICGS)

The ICGSs are software switches that may be set over the VMEbus (not locally accessible over the PCI) to interrupt a group of VMEbus modules that share an A16 base address.

Because the global switches are meant to be issued to several modules, the slave targets of a global switch access do not acknowledge the cycle, but rather the master driving the write DTACKs the cycle itself (the VIF_ABR register should be set to generate a self-access by the global-switch write).

A write to an even address clears the selected switch and a write to an odd address sets the switch.

If Global-switch interrupts are enabled in the VIC64 ICGSICR, an interrupt is generated to the local processor via the system interrupt controller. The vector for the interrupt is generated from the VIC64 ICGSVBR.

Bits <4:0> in the final register are set when there is a write access to the corresponding ICGPR. See the VIC64 specification for more complete details.

1.5.8.3 Interprocessor Communication Module Switches (ICMS)

The ICMSs are software writable switches that may be set over the VMEbus to interrupt a processor. The module switches, however, are meant to be issued to a specific module.

Because the module switches are meant for a specific module, the cycle is just like a normal write on the bus (nothing special like the ICGS case).

If Interprocessor Communication Module-switch interrupts are enabled in the VIC64 ICMSICR, an interrupt is generated. The vector for the interrupt is generated from the VIC64 ICMSVBR.

Note

There are restrictions on the usage of the above registers sets. Only one set can be used, as described, at a time.

The interprocessor communication register map is shown in Figure 1–26.

Figure 1–26 Interprocessor Communication Register Map

VIF_ABR<byte 1> + 01	8-bit General-purpose Register 0
VIF_ABR<byte 1> + 03	8-bit General-purpose Register 1
VIF_ABR<byte 1> + 05	8-bit General-purpose Register 2
VIF_ABR<byte 1> + 07	8-bit General-purpose Register 3
VIF_ABR<byte 1> + 09	8-bit General-purpose Register 4
VIF_ABR<byte 1> + 0B	VIC Revision register (read only)
VIF_ABR<byte 1> + 0D	VIC Status register (read only)
VIF_ABR<byte 1> + 0F	Inter-communication register status
VIF_ABR<byte 1> + 10	Clear Global Switch 0 (write only)
VIF_ABR<byte 1> + 11	Set Global Switch 0 (write only)
VIF_ABR<byte 1> + 12	Clear Global Switch 1 (write only)
VIF_ABR<byte 1> + 13	Set Global Switch 1 (write only)
VIF_ABR<byte 1> + 14	Clear Global Switch 2 (write only)
VIF_ABR<byte 1> + 15	Set Global Switch 2 (write only)
VIF_ABR<byte 1> + 16	Clear Global Switch 3 (write only)
VIF_ABR<byte 1> + 17	Set Global Switch 3 (write only)
VIF_ABR<byte 1> + 20	Clear Module Switch 0 (write only)
VIF_ABR<byte 1> + 21	Set Module Switch 0 (write only)
VIF_ABR<byte 1> + 22	Clear Module Switch 1 (write only)
VIF_ABR<byte 1> + 23	Set Module Switch 1 (write only)
VIF_ABR<byte 1> + 24	Clear Module Switch 2 (write only)
VIF_ABR<byte 1> + 25	Set Module Switch 3 (write only)
VIF_ABR<byte 1> + 26	Clear Module Switch 3 (write only)
VIF_ABR<byte 1> + 27	Set Module Switch 3 (write only)

Note

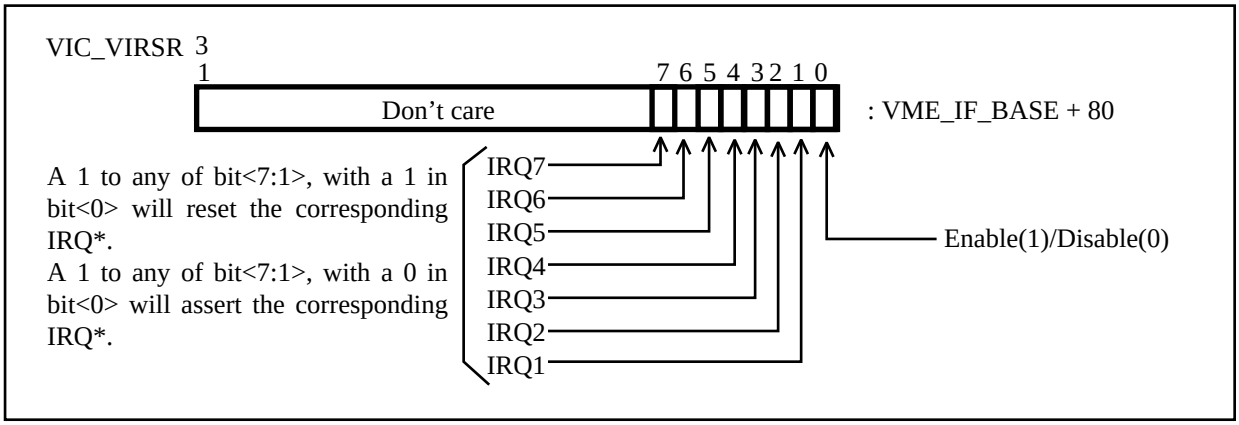
General-purpose Registers 0–7 are also accessible from PCI Memory space starting at address VME_IF_BASE + 60h.

1.5.9 AXPvme Generated VMEbus Interrupts

AXPvme can act as a VMEbus interrupter (as well as a VMEbus interrupt servicing agent, see Section 1.17).

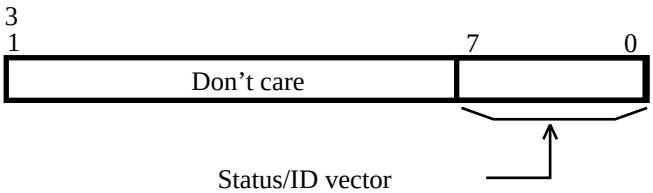
The VIC Interrupt Request/Status register (VME_IF_BASE + 80h) is used to control the state of AXPvme's IRQ1*-7* lines driven out onto the VME. Figure 1–27 shows the form of this register. When read, this register shows the current state of the IRQ lines driven out from AXPvme.

Figure 1–27 VMEbus Interrupt Request/Status Register



AXPvme employs the Release-On-Acknowledge mechanism for removal of its interrupt requests. Alternatively, the interrupt requests can be deasserted by writing to the same VMEbus Interrupt Request/Status register that is used to assert the IRQ* lines. When AXPvme sees an IACK on the VMEbus to one of its interrupt requests, it responds with a vector that is programmable in the VMEbus Interrupt Vector Base Registers (see Figure 1–28) starting at PCI Memory address VIF_ABR+84h.

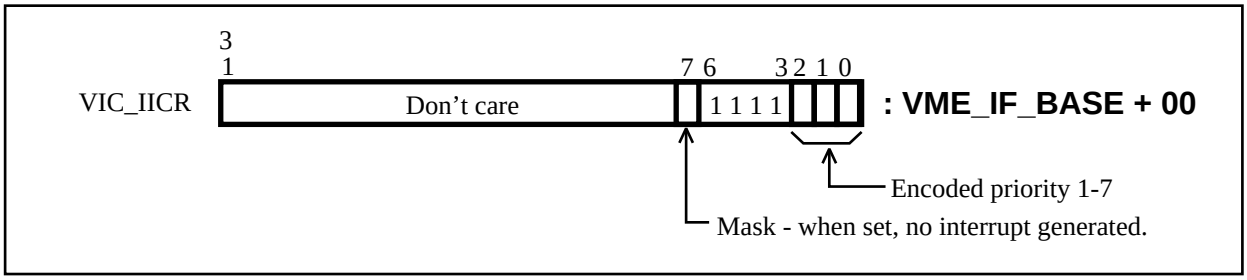
Figure 1–28 VMEbus Interrupt Vector Base Registers



Address	Register	Line
:VME_IF_BASE +84	VIVBR1	IRQ1*
:VME_IF_BASE + 88	VIVBR2	IRQ2*
:VME_IF_BASE + 8C	VIVBR3	IRQ3*
:VME_IF_BASE + 90	VIVBR4	IRQ4*
:VME_IF_BASE + 94	VIVBR5	IRQ5*
:VME_IF_BASE + 98	VIVBR6	IRQ6*
:VME_IF_BASE + 9c	VIVBR7	IRQ7*

A local interrupt can be generated to the CPU, by the VME interface, when it sees an interrupt acknowledge cycle to itself on the VMEbus. This interrupt (which can be enabled or disabled) can be used to inform system software that the VMEbus interrupt request has been serviced, for instance. The VIC Interrupter Interrupt Control register (VME_IF_BASE + 00) is shown in Figure 1–29.

Figure 1–29 VMEbus Interrupter Interrupt Control Register



The vector that is returned when the processor locally IACKs this interrupt comes from the VIC Error Group Interrupt Vector Register (VIC_EGIVBR) described in Section 1.17.

1.6 VME Interface

1.6.1 How AXPvme Uses the VIC64 Registers

There are some registers in the VME interface (most notably those that live in the VIC64) which, for the AXPvme implementation, must be set up to fixed configuration values by software before normal, predicable operation can be guaranteed. This section describes these registers and other recommended VME interface initialization.

Note that with power-up defaults for the DC7407 registers, the only steps required to operate the VME interface are:

1. Set up the three VME interface PCI base registers
2. Program S/G RAM as required
3. Configure the VIC64 as per VIC64 initialization (some timing control register values will be defined)
4. Operate the VME interface as a normal VIC64 based system

The use of DC7407 registers with values other than defaults will enable a user to take advantage of the various added features of the DC7407.

This should be a guide to required VIC64 register values in the AXPvme module. This is not intended as a description of the register set in the VIC64 but to recommend or mandate settings for operation in the AXPvme VME interface.

The address map in the VIC64 databook places the VIC registers in byte 3 of a particular longword address; in AXPvme, the VIC registers accessed over the PCI are seen at byte zero in each longword.

VIICR

Bits 2-0	Local IPL setting for VMEbus interrupter acknowledge received interrupt
Bits 6-3	Reserved should read as 1s
Bit 7	Interrupt mask bit

VICR1-7

Bits 2-0	Local IPL setting for VMEbus interrupt
Bits 6-3	Reserved should read as 1s
Bit 7	Interrupt mask bit
DMASICR	
Bits 2-0	Local IPL setting for end of DMA interrupt
Bits 6-3	Reserved should read as 1s
Bit 7	End of DMA interrupt mask bit
LICR1-7	
Bits 2-0	Local IPL setting for LIRQ interrupt line
Bit 3	Indicates voltage level at LIRQ pin
Bit 4	Autovector enable (this bit must be set in AXPvme)
Bit 5	Edge/level enable (for LICR2 and LICR7, this bit must be clear in AXPvme)
Bit 6	Polarity set (for LICR2 and LICR7, this bit must be clear in AXPvme)
Bit 7	Local interrupt mask bit
ICGSICR	
Bits 2-0	Local IPL for global switch interrupts
Bit 3	Reserved should read as 1
Bits 7-4	Interrupt mask bit for ICGS [3:0]
ICMSICR	
Bits 2-0	Local IPL for module switch interrupts
Bit 3	Reserved should read as 1
Bits 7-4	Interrupt mask bit for ICMS [3:0]
EGICR	
Bits 2-0	Local IPL for error group interrupts
Bit 3	SYSFAIL asserted (read only)
Bit 4	SYSFAIL interrupt mask
Bit 5	Arbitration timeout interrupt mask
Bit 6	VIC/CY write post fail interrupt mask
Bit 7	AC fail interrupt mask
ICGSIVBR	
Bits 1-0	Read only
Bits 7-2	User defined, combine with ICGS switch number to provide vector
ICMSIVBR	
Bits 1-0	Read only
Bits 7-2	User defined, combine with ICMS switch number to provide vector
LIVBR	
Bits 1-0	Read only
Bits 7-2	User defined, combine with LIRQ number to provide vector
EGIVBR	
Bits 1-0	Read only
Bits 7-2	User defined, combine with fixed codes to provide vector

ICFSR	
Bits 3-0	Module switches
Bits 7-4	Global switches
ICR0-4	General-purpose registers accessible over the VME or local bus
ICR5	Read-only register containing the VIC64 revision accessible over VME or local bus
ICR6	
Bits 1-0	Read only from the VMEbus. These bits should be cleared down by the processor after reset.
Bits 5-2	Reserved should read as 1s
Bit 6	Should be cleared by the processor as soon as possible after reset. If enabled via LICR7, this bit being set asserts SYSFAIL* on the VMEbus.
Bit 7	Read only
ICR7	
Bits 4-0	Read and write from the VME or locally. These bits are set if the corresponding ICR is written.
Bit 5	Read only
Bit 6	HALT and RESET control
Bit 7	VME SYSFAIL* mask (must be set after reset if resets are not to be translated into SYSFAIL* assertion)
VIRSR	
Bit 0	Enable for VME interrupter
Bits 7-1	If bit 0 is set during the write that sets any given bit, the corresponding VMEbus interrupt is asserted. These bits are cleared if bit 0 is cleared during the write that sets a bit.
VIVBR1-7	Each register sets the vector returned on VME interrupt acknowledge cycles at that interrupt level.
TTR	
Bit 0	Set to include VMEbus acquisition time in local bus timeout
Bit 1	When system controller set to indicate arbitration timeout
Bits 4-2	Recommended Local bus timeout period is 64 µs (011)
Bits 7-5	Recommended VMEbus timeout period is 128 µs (100)
	Note: The use of this depends largely on the VME environment. However, if the AXPvme is a system controller and a cycle times out on the Local bus after timing out on the VMEbus, then the Local bus cycle will hang. Avoid this scenario by making the Local bus time out first or not at all.
LBTR	
Bits 3-0	Minimum PAS assertion time. Leave at zero.
Bit 4	Minimum DS deasserted time. Must be set in AXPvme.
Bits 7-5	Minimum PAS deasserted time. Must be binary 110 in AXPvme.
BTDR	
Bit 0	Dual Path enable. Set this bit.
Bit 1	AMSR register. Used to set up user-defined AM codes for BLTs.
Bit 2	Local bus 256 bus byte boundary. Recommend this be set.

Bit 3	VME 256 bus crossing enabled. Recommend this be set.
Bit 4	Enables D64 Master Operation.
Bit 5	Enable enhanced Turbo mode. Must be clear.
Bit 6	Enables D64 Slave operation. Recommend that this be set.
Bit 7	Enable 2K byte boundary crossing for D64. If set, software must check that the D64 BLT start address is 2 Kbyte aligned and that the transfer does not cross a 64 Kbyte boundary.
ICR	
Bit 0	Read-only system controller pin.
Bit 1	Turbo enable. Must be clear.
Bit 2	Metastability delay. Recommend this be clear.
Bits 4,3	Deadlock signaling. Must be clear.
Bits 5-7	RMC Control bits 1 to 3.
ARCR	
Bits 3-0	VMEbus fairness timer enable.
Bit 4	DRAM refresh enable. Must be clear.
Bits 6,5	VMEbus request level.
Bit 7	Arbitration mode.
AMSR	Used to define response top and generation of user-defined address modifier codes.
BESR	All 8 bits are flags set by the VIC after status conditions that must be cleared by the processor.
DMASR	
Bit 0	Block transfer in progress. Once set, must be cleared by processor.
Bit 1	LBERR during DMA transfer
Bit 2	BERR during DMA transfer
Bit 3	Local bus error
Bit 4	VMEbus BERR
Bits 5,6	Reserved read as 1s
Bit 7	Master write post information stored in CYs
SS0CR0	
Bits 1-0	Accelerated transfer mode. Must be set to binary 10.
Bits 3,2	Must be binary 01 for A24 slave selection.
Bit 4	D32 enable. Must be set in AXPvme.
Bit 5	Supervisor access.
Bits 7,6	Periodic timer enable. Must be binary 00 in AXPvme.
SS0CR1	Local bus timing values. Must be “00” hex in AXPvme.
SS1CR0	
Bits 1-0	Must be set to binary 10 (accelerated transfer mode).
Bits 3,2	Must be binary 00 for A32 slave selection.
Bit 4	D32 enable. Must be set in AXPvme.
Bit 5	Supervisor access.
Bit 6	VIC/CY master write posting enable. Recommend this be clear.

Bit 7	Slave write post enable. Must be clear in AXPvme.
SS1CR1	Local bus timing values. Must be “00” hex in AXPvme.
RCR	
Bits 5-0	Block transfer burst length.
Bits 7,6	VMEbus release mode.
BTCR	
Bits 3-0	Interleave period. A value of F hex is recommended.
Bit 4	Data direction bit (0=write, 1=read).
Bit 5	MOVEM enable. Recommend this be clear.
Bit 6	BLT with local DMA enable.
Bit 7	Module based DMA transfer enable.
BTLR1-0	These are the block transfer length registers for local DMA BLTs.
SRR	System reset register

1.6.2 VME Register Summary

VME_IF_BASE +

00	VIC_IICR	VMEbus Interrupter Interrupt Control Register
04-1C	VIC_ICPR1-7	VMEbus Interrupt Control Registers 1-7
20	VIC_DMASICR	DMA Status Register
24-3C	VIC_LICR1-7	Local Interrupt Status Register
40	VIC_ICGISR	ICGS Interrupt Control Register
44	VIC_ICMSICR	ICMS Interrupt Control Register
48	VIC_EGICR	Error Group Interrupt Control Register
4C	VIC_ICGSIVBR	ICGS Vector Base Register
50	VIC_ICMSVBR	ICMS Vector Base Register
54	VIC_LIVBR	Local Interrupt Vector Base Register
58	VIC_EGIVBR	Error group Interrupt Vector Base Register
5C	VIC_ICSR	Interprocessor Communications Switch Register
60-70	VIC_ICR0-4	Interprocessor Communications Registers 0-4
74	VIC_ICR5	Interprocessor Communications Register 5
78	VIC_ICR6	Interprocessor Communications Register 6
7C	VIC_ICR7	Interprocessor Communications Register 7
80	VIC_VIRSR	VMEbus Interrupt Request/Status Register
84-9C	VIC_VIVBR1-7	VMEbus Interrupt Vector Base Registers 1-7
A0	VIC_TTR	Transfer Timeout Register
A4	VIC_LBTR	Local Bus Timing Register
A8	VIC_BTDR	Block Transfer Definition Register
AC	VIC_ICR	Interface Configuration Register
B0	VIC_ARCR	Arbiter/Requester Configuration Register
B4	VIC_AMSR	Address Modifier Source Register
B8	VIC_BESR	Bus Error Status Register

BC	VIC_DMASR	DMA Status Register
C0	VIC_SS0CR0	Slave Select 0/Control Register 0
C4	VIC_SS0CR1	Slave Select 0/Control Register 1
C8	VIC_SS1CR0	Slave Select 1/Control Register 0
CC	VIC_SS1CR1	Slave Select 1/Control Register 1
D0	VIC_RCR	Release Control Register
D4	VIC_BTCR	Block Transfer Control Register
D8	VIC_BTLR1	Block Transfer Length Register 1
DC	VIC_BTLR0	Block Transfer Length Register 0
E0	VIC_SRR	System Reset Register
E4	BTLR2	Block Transfer Length Register 2
E8-FC		Reserved Locations
100	VIP_CR	VME Interface Processor Control Register
104	VIP_BESR	VME Interface Processor Bus Error/Status Register
108	VIP_ICR	VME Interface Processor Interrupt Control Register
10C	VIP_IRR	VME Interface Processor Interrupt Reason Register
110	VIP_HWIPL	VME Interface Processor Hardware IPL Mask Register
114	VIP_DIAG CSR	VME Interface Processor Diagnostic Register
118	VIP_PMCSR	VME Interface Processor Page Monitor CSR
11C	VIP_OBISGABR	VME Interface Processor Outbound Internal S/G Entry ABR
120	VIP_OBISGMSK	VME Interface Processor Outbound Internal S/G Entry Mask
124	VIP_OBISGWORD	VME Interface Processor Outbound Internal S/G Entry Control Word
128	VIP_IBISGMSK	VME Interface Processor Inbound Internal S/G Entry Mask
12C	VIP_IBISGWORD	VME Interface Processor Inbound Internal S/G Entry Control Word
130	VIP_SGCCHIX	VME Interface Processor SG Cached Index
134	VIP_SGCWRD	VME Interface Processor SG Cached Control Word
138	VIP_PCIERTADR	VME Interface Processor PCI Error Target Address Register
13C	VIP_PCIERTCBE	VME Interface Processor PCI Error Target Command /Byte Enables Register
140	VIP_PCIERIADR	VME Interface Processor PCI Error Initiator Address Register
144	VIP_LERADR	VME Interface Processor VME/Local Bus Error Address Register
148-17C		Reserved Locations
180	VIFMASK	VMEbus i/f Address Base Mask Register
184	VIFABR	VMEbus i/f Address Base Register
188-3FC		Reserved

1.6.3 VME Subsystem Restrictions (as of 22-Nov-94)

This section describes limitations on the use of the VME subsystem due to outstanding hardware constraints. The intention is that these will be eliminated as new revision hardware components become available. This section will be updated as restrictions change. Please contact your Field Application Engineer for the latest status on these constraints.

1.6.3.1 D64 Writes to Invalid Pages

When developers perform illegal memory access operations using D64 transfers, the module fails to recover according to specification causing the system to crash. Developers and customers should not be using memory in this fashion. If they do, the module crashes and a re-boot is required.

The illegal memory access operations are:

- D64 write to a page mapped to a nonexistent memory location
- User mode D64 write to a Supervisor page
- D64 write to a page with an invalid S/G map
- D64 write to a write protected page

1.6.3.2 D64 Write/IACK Cycle Collisions

An AXPvme can be used to generate interrupts on the VME, responding to IACK cycles as the interrupts are serviced. The AXPvme can also be accessed as a VME Slave using D64 data type. When an AXPvme has just finished receiving an inbound D64 write, there is a small period of time when an IACK cycle will not be acknowledged with the proper vector. There are several workarounds that will prevent this situation from occuring. Which workaround depends upon your application.

If an AXPvme is not posting interrupts to the VME, IACK cycles will not be run against the AXPvme.

If the AXPvme is not being used as the target of D64 writes, the collision will not occur.

If interrupts are posted to the VME only under conditions where there is no VME Master targeting the AXPvme with D64 writes, the collision will not occur.

1.6.3.3 Collision of VIC64 Master Write Posting with Master Block Transfers

Write Posting at the VIC64 is not recommended. A collision of the following three cycles will cause a bus timeout error:

- Posted Master Write in the VIC64/CY964
- AXPvme is being accessed as a VME Slave
- Master Block Transfer is being initiated by a “Pseudo write” to the VIC64 over the Local bus

1.6.3.4 VIC64 Errata: A16 Master Cycles During Interleave

The Cypress VIC64-00 Design Considerations document (dated 22 February 1994) lists the following errata.

“ A16 master cycles during an interleave period with dual path enabled will cause BERR* and LBERR* to be asserted. ”

Followup conversations with Cypress (and testing) has determined that a further statement should be added. Apparently, the problem only occurs if the DMA enable bit is set (BTCR<6>). The DMA drivers used with the AXPvme systems always clear this bit immediately after the "pseudo-write" to avoid any PIO being taken as another "pseudo-write". Therefore, BTCR<6> is always clear by the time an A16 access could get through in an interleave gap.

While this is not a problem for customers using driver software supplied by Digital, anyone writing their own interface to the VIC64s DMA engine should use the same sequence to ensure this problem is not encountered.

1.6.3.5 Module Reset Temporarily Disables IACK Chain

If an AXPvme module is reset, the IACK path through the module being reset is disabled for the duration of the reset (approximately 500 ms). If a VME module further down the IACK chain has a VME interrupt posted and the IACK cycle is attempted during the AXPvme reset, the IACK will not be received and the VME IACK cycle will time out.

1.7 Network Interface

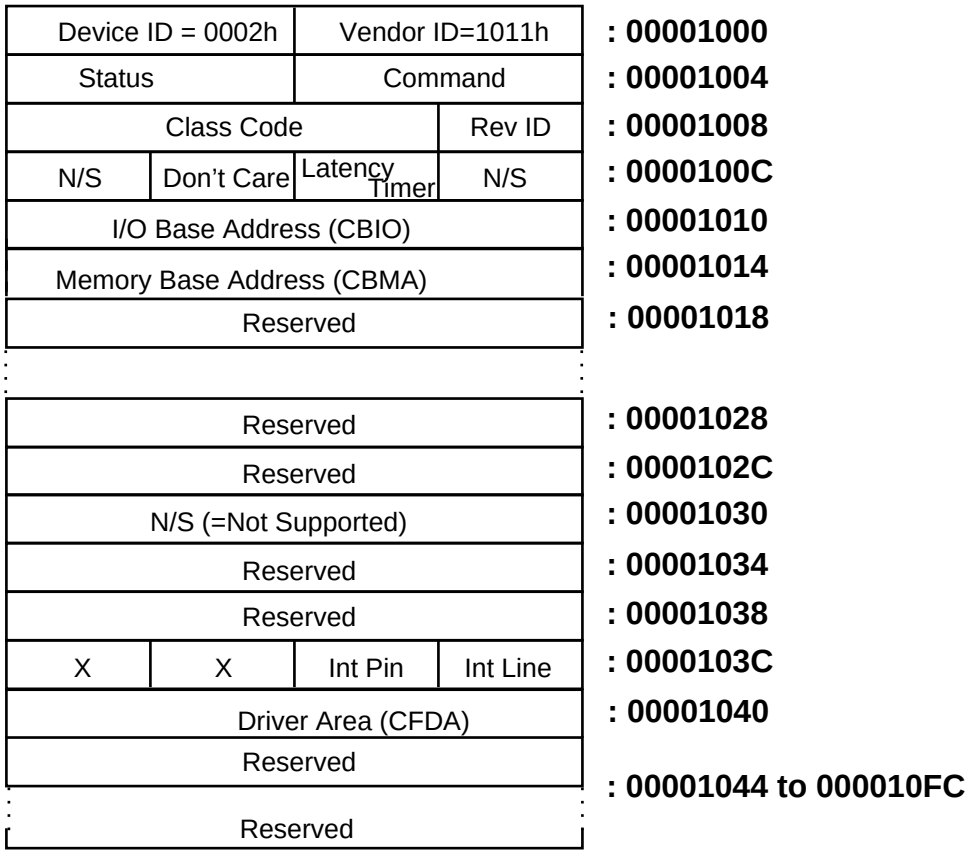
The AXPvme attaches to the network via an Ethernet AUI connector on the front panel. This interface is based on the DECchip 21040-AA. This is a PCI based Ethernet solution with which processor intervention in LAN control is kept to a minimum. It behaves as a bus slave when communicating with the host (that is, configuration register and CSR accesses) and as a bus master when communicating with memory.

Refer to the DECchip 21040-AA specification for full details of programming and use.

1.7.1 DECchip 21040-AA PCI Configuration Registers

The DECchip 21040-AA will respond to PCI configuration reads and writes to its configuration registers (see Figure 1–30). For full bit definitions of these registers, refer to the DECchip 21040-AA specification.

Figure 1–30 DECchip 21040-AA PCI Configuration Registers



1.7.2 DECchip 21040-AA CSRs

The DECchip 21040-AA has 16 Command and Status Registers that may be accessed by the host. The address field in Table 1–12 reflects the offset from the CSR Base Address (CBIO,CBMA). The CSRs are located in the host I/O or memory space. The CSRs are quadword aligned and can only be accessed

using longword instructions. See the DECchip 21040-AA specifications for more details.

Table 1–12 DECchip 21040-AA CSRs

Register	Meaning	Address
CSR0	Bus Mode Register	xxxx xx00H
CSR1	Transmit Poll Demand	xxxx xx08H
CSR2	Receive Poll Demand	xxxx xx10H
CSR3	Rx List Base Address	xxxx xx18H
CSR4	Tx List Base Address	xxxx xx20H
CSR5	Status Register	xxxx xx28H
CSR6	Serial Command Register	xxxx xx30H
CSR7	Interrupt Mask Register	xxxx xx38H
CSR8	Missed Frame Register	xxxx xx40H
CSR9	ENET ROM Register	xxxx xx48H
CSR10	Reserved	xxxx xx50H
CSR11	Full-Duplex Register	xxxx xx58H
CSR12	SIA Status Register	xxxx xx60H
CSR13	SIA Connectivity Register	xxxx xx68H
CSR14	SIA Tx Rx Register	xxxx xx70H
CSR15	SIA General Register	xxxx xx78H

1.7.3 DECchip 21040-AA PCI Cycles

The DECchip 21040-AA responds as a slave to single longword accesses in I/O space and configuration space. The DECchip 21040-AA acts as a master over the PCI for memory reads and memory writes.

The DECchip 21040-AA's tenure on the bus for DMA accesses to memory can be controlled via the programmable burst length in the Bus Mode Register CSR0 and the PCI latency timer value in the Configuration Latency Timer Register.

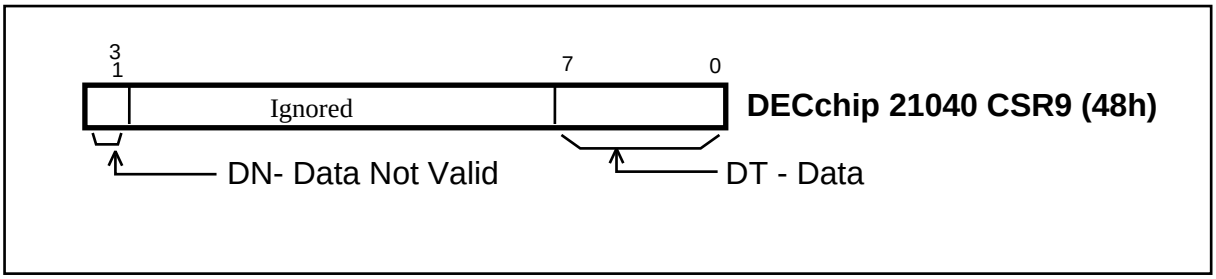
The DECchip 21040-AA will handle target initiated retry, abort, and DEVSEL abort cycle termination when it is a bus master. Target aborted terminations will cause an interrupt.

As a slave, burst writes to its I/O space will cause target retry termination of the cycle after the first longword access.

1.7.4 Ethernet Address ROM

AXPvme's Ethernet ID address is stored in an on-board SROM. The SROM is readable from the DECchip 21040-AA register CSR9 (see Figure 1–31). Each read access causes 8-bit serial read cycles from the ENET ROM. Writing to the register resets the pointer of the ENET ROM to its first location. The SROM is a 20-pin PLCC and is socketed.

Figure 1–31 DECchip 21040-AA CSR9 (ENET ROM Register)



1.8 SCSI

The SCSI interface used in the AXPvme is based on the NCR 53C810 chip. (The PCI version of the NCR 53C720 SCSI controller chip, though only single-ended 8-bit plus parity operation, is supported).

For full operational programming details, see the 53C720 programming guide and the most up-to-date 53C810 chip specification.

1.8.1 SCSI ID

The SCSI ID is set by writing the SCID register (offset 04h) in the 53C810. In AXPvme this will be set up, from NVRAM, by firmware. The ID can be updated from the console. The default SCSI ID is 7.

1.8.2 SCSI Connection and Termination

The SCSI bus is routed to the VMEbus P2 connector. The pinning for the user-defined pins of the P2 connector is given in Section 1.21. An interface to a standard SCSI cable is handled via a special P2 breakout module. This module brings the SCSI bus to a standard 50-pin SCSI connector pinning for direct connection to an unshielded SCSI A-cable. Note that AXPvme single-slot and AXPvme dual-slot have different power needs and therefore require different breakout module designs. Both, AXPvme single-slot and AXPvme dual-slot breakout modules support the same 50-pin SCSI connector pinning.

SCSI termination is implemented on these special P2 breakout boards as well. Active SCSI terminators are enabled/disabled through the use of a 6-pin jumper block on the breakout module. Pins 1 and 3 should be jumpered to enable SCSI termination at the AXPvme. Pins 3 and 5 should be jumpered to disable SCSI termination at the AXPvme.

1.8.3 53C810 Configuration Block

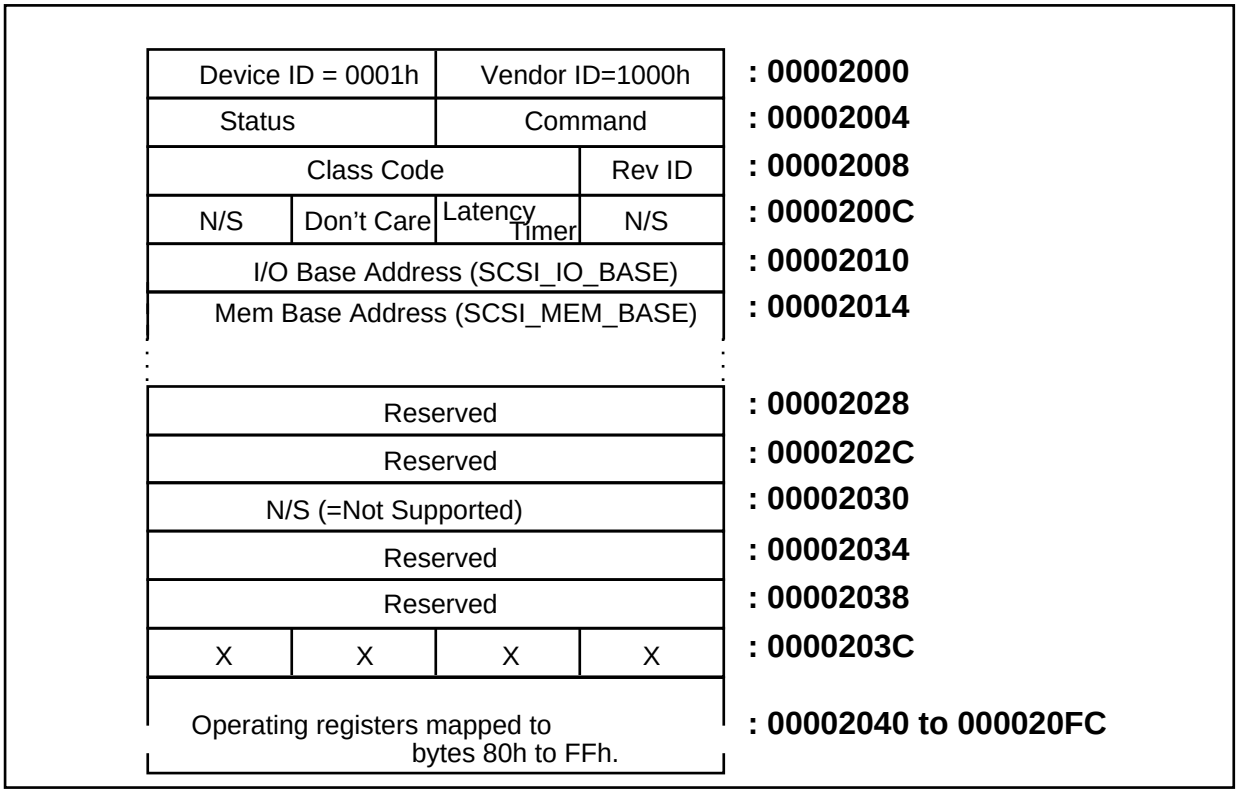
The SCSI controller resides in PCI Configuration space between the addresses 00002000 to 000020FF (see Section 1.4.1).

The 53C810 has two base address registers, one for I/O and the other for Memory space. This allows the 128 bytes of registers to be accessible in both PCI Memory and I/O regions.

Figure 1–32 shows the supported fields in the 53C810's PCI Configuration block.

Note that the 53C810 supports a Latency Timer, and via normal (not PCI configuration) registers, has a programmable data burst size on the PCI (see DMODE register. Offset 38h.).

Figure 1–32 53C810 Configuration Block



1.8.4 SCSI Programming

The NCR 53C810 device allows a low-level register interface or programming via NCR's SCSI Scripts, which allow the controller to effect high-level SCSI operations with very little intervention from the processor.

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

1.8.5 SCSI Control Status Registers

The 53C810 has 128 accessible bytewide control and status registers (see Table 1–13). These registers are accessible in PCI I/O space starting at address SCSI_IO_BASE, in PCI Memory space starting at address SCSI_MEM_BASE, and in Configuration space starting at 00002080h.

Table 1–13 53C810 Register List

Label	R/W	Description	Offset	Label	R/W	Description	Offset
SCNTL0	R/W	SCSI Control 0	00	DBC	R/W	DMA Byte Counter	24-26
SCNTL1	R/W	SCSI Control 1	01	DCMD	R/W	DMA Command	27
SCNTL2	R/W	SCSI Control 2	02	DNAD	R/W	DMA Next Add for Data	28-2B
SCNTL3	R/W	SCSI Control 3	03	DSP	R/W	DMA SCRIPTS Pointer	2C-2F
SCID	R/W	SCSI Chip ID	04				30-33
SXFER	R/W	SCSI Transfer	05	ScratchA	R/W	Gen Purpose Scratch Pad	34-37
SDID	R/W	SCSI Destination ID	06	DMODE	R/W	DMA Mode	38
GPREG	R/W	General Purpose	07	DIEN	R/W	DMA Interrupt Enable	39
SFBR	R/W	1st Byte Rx'ed	08	DWT	R/W	DMA Watchdog Timer	3A
SOCL	R/W	Output Cntrl Latch	09	DCNTL	R/W	DMA Control	3B
SSID	R	Selector ID	0A	ADDER	R	Sum o/p of internal adder	3C-3F
SBCL	R/W	Bus Control Lines	0B	SIEN0	R/W	SCSI Interrupt Enable 0	40
DSTST	R	DMA Status	0C	SIEN1	R/W	SCSI Interrupt Enable 1	41
SSTAT0	R	SCSI Status 0	0D	SIST0	R	SCSI Interrupt Status 0	42
SSTAT1	R	SCSI Status 1	0E	SIST1	R	SCSI Interrupt Status 1	43
SSTAT2	R	SCSI Status 2	0F	SLPAR	R/W	SCSI Longitudinal Parity	44
DSA	R/W	Data Structure Addr	10-13	SWIDE	R	SCSI Wide Residue Data	45
ISTAT	R/W	Interrupt Status	14				46-47
		RESERVED	15-17	STIME0	R/W	SCSI Timer 0	48
CTEST0	R/W	Chip Test 0	18	STIME1	R/W	SCSI Timer 1	49
CTEST1	R	Chip Test 1	19	STEST0	R	SCSI Test 0	4C
CTEST2	R	Chip Test 2	1A	STEST1	R	SCSI Test 1	4D
CTEST3	R	Chip Test 3	1B	STEST2	R/W	SCSI Test 2	4E
TEMP	R/W	Temporary Stack	1C-1F	STEST3	R/W	SCSI Test 3	4F
			20	SIDL	R	SCSI Input Data Latch	50-51
CTEST4	R/W	Chip Test 4	21	SODL	R/W	SCSI Output Data Latch	54-55
			22	SBDL	R	SCSI Bus Data Lines	58-59

(continued on next page)

Table 1–13 (Cont.) 53C810 Register List

Label	R/W	Description	Offset	Label	R/W	Description	Offset
CTEST6	R/W	Chip Test 5	23	ScratchB	R/W	Gen Purpose Scratch Pad	5C-5F

1.8.6 Clocking Information

The AXPvme implementation of the SCSI interface ties the SCLK pin to 32 MHz. This has ramifications on the configuration of the SCNTL3 (03h) and SXFER (05h) registers. The following settings yield peak SCSI-2/SCSI-1 transfer rates of 8/5.33 Mbytes/s, respectively.

SCNTL3 (03h) <7:0> = 12h
SXFER (05h) <7:4> = 0h

See NCR's 53C810 documentation for further details.

1.9 ISbus

The ISbus is a simple 8-bit data, 16-bit address, non-multiplexed resource bus. It is interfaced to the PCI via the SIO chip from Intel (the 82378IB). The interface translates PCI I/O references to the ISbus into simple read and write cycles to the resources hanging off the ISbus lines. In addition, PCI Memory read cycles to system ROM locations are supported.

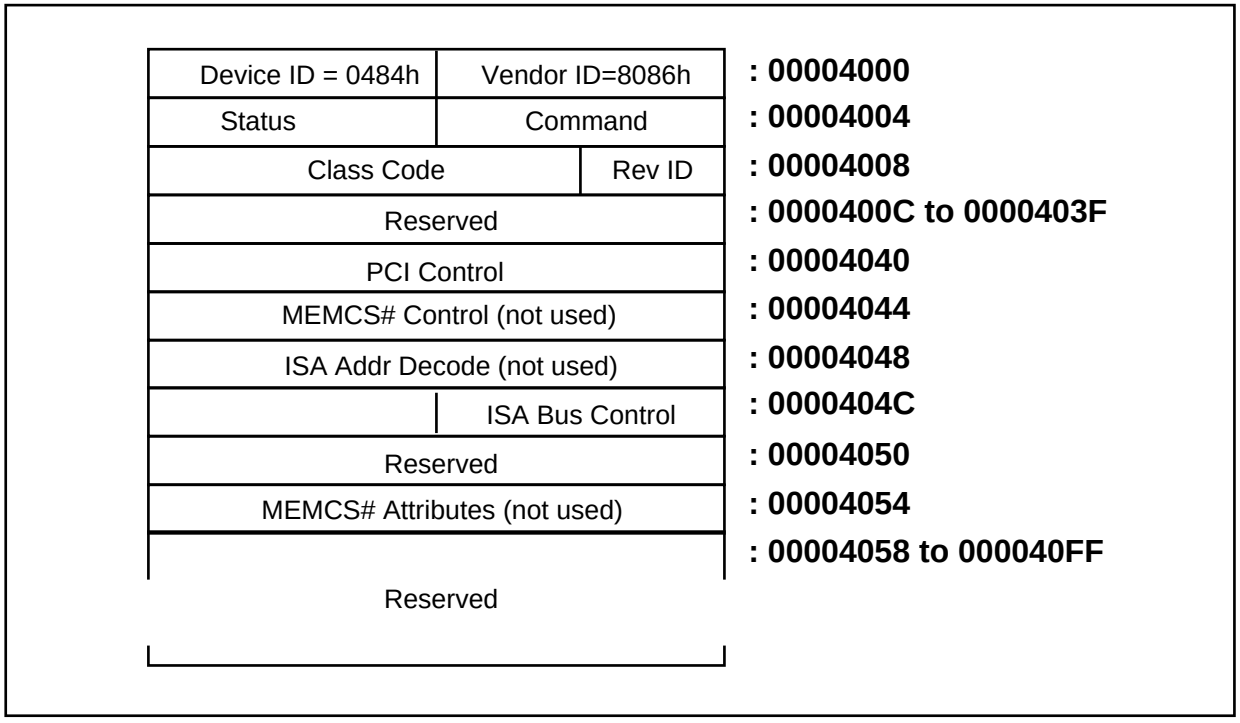
Many of the system functions and registers are interfaced over the ISbus. Figure 1–34 shows the layout of the ISbus for AXPvme. The addresses shown are both PCI I/O addresses and ISbus addresses.

1.9.1 ISbus Adapter (SIO) Configuration Space

The SIO does not have any base addresses registers (rather, it negatively decodes fixed regions in both PCI I/O and memory space).

However, there are a number of registers of interest for the AXPvme system. Figure 1–33 shows the layout of the SIO configuration space with the registers of interest to AXPvme shown.

Figure 1–33 SIO Configuration Block



1.9.1.1 AXPvme Required Setup

Many of the reset defaults of the SIO for configuration registers suit the required setup for AXPvme. Only registers that need to be changed from the defaults will be discussed here.

PCI Control Register

This register is used to enable the SIO to respond to PCI IACK cycles and to set the expected assertion speed of the DEVSEL# signal so that the subtractive decode sample point can be set. PCI Posted Write Buffer should also be enabled.

Bit <5> must be set to a 1 (default), while bits <4:3> must be set to <00> to allow slow sample point timing for negative decode. Bit <2> (PCI Posted Write Buffer Enable) should be set to <1>. All other bits should be zero.

ISbus Control

The ISbus Control word is dealt with as two separate bytewide registers: the ISA Controller Recovery Timer Register at offset +4Ch and the ISA Clock Divisor Register at offset +4Dh.

The I/O recovery mechanism in the SIO is used to add additional recovery delay between PCI originated I/O cycles to the ISbus. As only 8-bit cycles are supported by AXPvme, only bits <6:3> of the register are significant.

Bits <6:3> define the number of system-clock ticks which are inserted between back-to-back cycles. Note that there is a difference between version "IB" and version "ZB" of the SIO. The table below applies "IB" of the SIO only. The "*"ed row shows the required AXPvme value, bits <7,2:0> MBZ for AXPvme.

Bits <6:3>	I/O Recovery Cycles
0xxx	+0
* 1001	+1
1010	+2
1011	+3
1100	+4
1101	Reserved
1110	Reserved
1111	Reserved
1000	Reserved

ISA Clock Divisor

The ISA Clock Divisor register controls the enabling of positive decode for BIOS ROM and the PCI to ISA clock divisor.

For AXPvme, the BIOS ROM region must not be positively decoded so bit <6> of this register must be cleared, while bits <2:0> should be 000 for a 32 MHz PCI system. All other bits must be zero.

1.9.2 ISbus Address Space

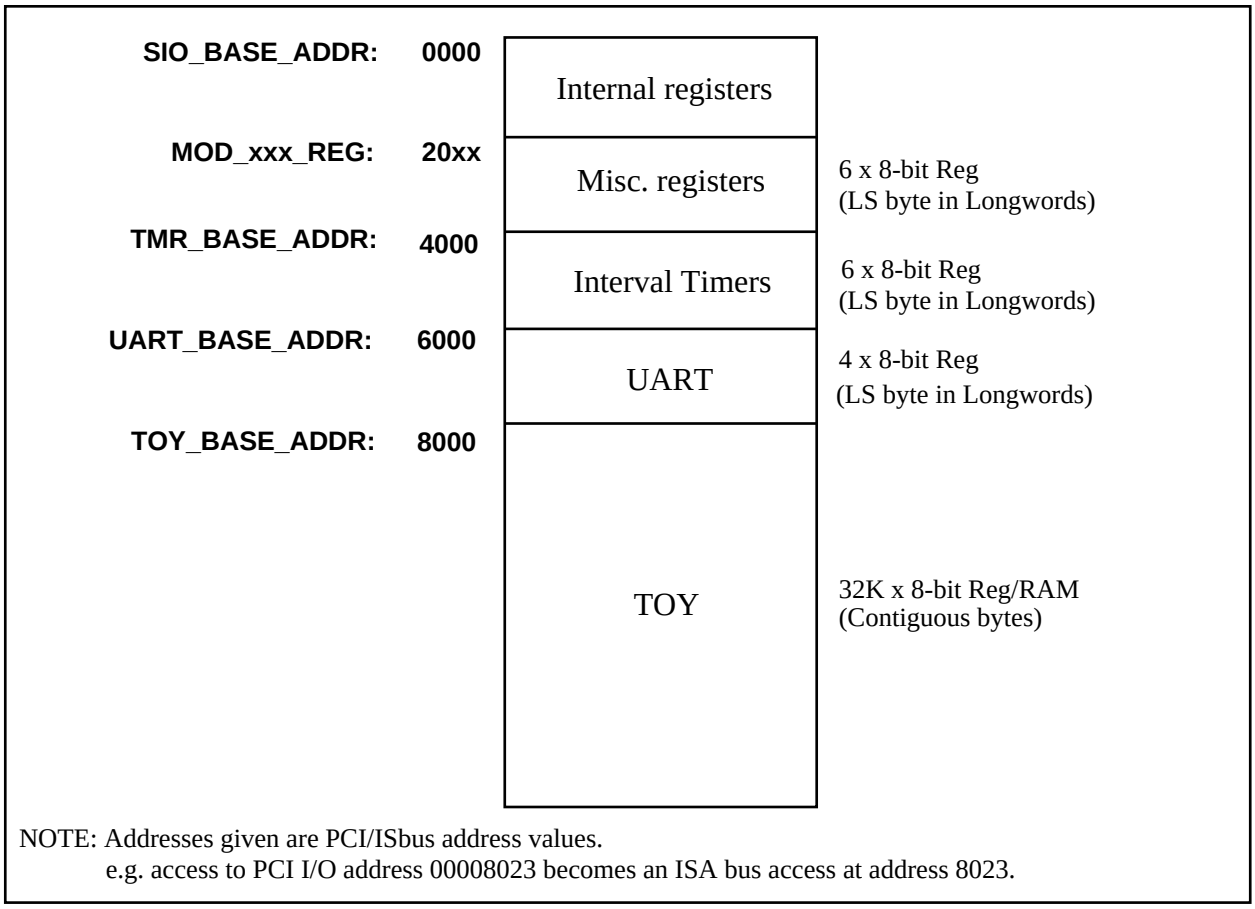
In I/O space, the bottom 64K of PCI address space is picked up by the SIO and mapped onto internal registers and out onto the ISbus.

In Memory space, the bottom 1 Mbyte (PCI memory address 00000 to 100000h) is mapped straight-through to the ISbus.

It should be noted that other regions of I/O and Memory space could be passed on to the ISbus by the SIO but none of these are of interest to the AXPvme system. In particular, these address regions are negatively decoded, so will not upset any other PCI device that is programmed to positively decode PCI addresses.

Figure 1–34 shows the ISbus I/O address layout.

Figure 1–34 ISbus I/O Address Layout



NOTE: Addresses given are PCI/ISbus address values.
e.g. access to PCI I/O address 00008023 becomes an ISA bus access at address 8023.

Each of these regions relate to separate resources on the ISbus and are dealt with in individual sections in this specification.

The Internal Register section (0000 to 1FFF) contains registers internal to the SIO. Most of these registers are concerned with SIO DMA functions that are not supported by AXPvme.

The only internal registers of interest in this system are:

Address	Function
0020	Programmable Interrupt Controller 1 - Control
0021	Programmable Interrupt Controller 1 - Mask
0061	NMI Status and Control
00A0	Programmable Interrupt Controller 2 - Control
00A1	Programmable Interrupt Controller 2 - Mask

These registers are described in Section 1.17.

1.9.3 ISbus Operation

The DECchip 21066 can access the ISbus devices in I/O space on a byte-by-byte basis. AXPvme only supports single-byte accesses to all ISbus locations.

Most resources about the ISbus are accessed as the least-significant byte of aligned longwords. The exception to this is the TOY and the ROM. Both of these regions are contiguous bytes, but it must be remembered that accesses to the ISbus must only have one PCI byte enable asserted.

For more information on the SIO and its operation, refer to the most recent 82378IB Chip Specification document.

1.10 Module Registers

There are 5 miscellaneous registers in the AXPvme system, implemented in module logic for a variety of read/write functions.

- Module Display Control Register - 2400h
- Module Configuration Register - 2800h
- Module Control Register Number 1 - 2C00h
- Module Control Register Number 2 - 3400h
- Reset Reason Register - 3000h
- Clear Heartbeat Register - 2000h

1.10.1 Module Display Control Register

Base address = MOD_DISP_REG = 2400h.

The display is a 5x7 dot-matrix intelligent display device, with 96 characters. The unit is read/writable via the Display Control Register (MOD_DISP_REG).

The display character is stored in bits <6:0>. The most significant bit (bit <7>) can be set to increase the brightness of the display.

Figure 1–35 shows the character set of the display. The numbers along the left-hand edge are the most-significant hex digit of the character number, while the least-significant is along the top. For example, the character “W” is displayed by writing a value of 57h to the display register. A value of D7h would display “W” full brightness.

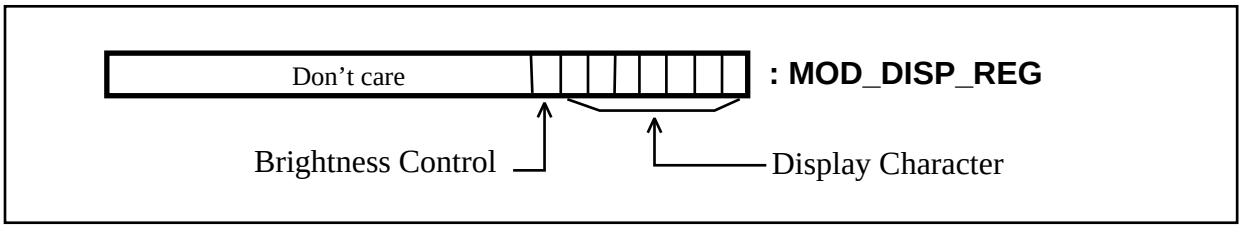
Figure 1–35 Display Character Set

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0,1					B	L	A	N	K							
2		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	:::

After system reset, the display will default to character 7Fh (“:::”) at full brightness. During system reset, all dots in the matrix will be lit.

Figure 1–36 shows the Module Display Control Register.

Figure 1–36 Module Display Control Register

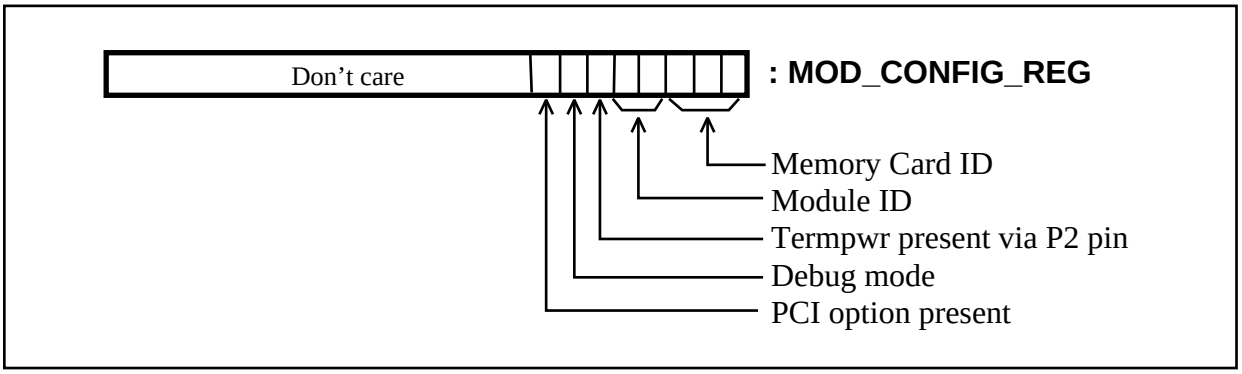


1.10.2 Module Configuration Register

Base address = MOD_CONFIG_REG = 2800h.

This read-only register contains information relating to module revision, memory module installed, cache/SCSI options, and so forth. The information read from this register is hardwired on the module, so it is unaffected by resets. Figure 1–37 shows the Module Configuration Register.

Figure 1–37 Module Configuration Register



1.10.2.1 Memory Card ID <2:0>

This field gives a 3-bit signature associated with the memory card installed.

Memory Card ID <2:0>	Memory Type
000	No Memory
010	8 Mbytes
110	16 Mbytes
111	32 Mbytes
100	64 Mbytes
101	128 Mbytes

1.10.2.2 Module ID <4:3>

These two bits reflect the hardware type of the module.

Module ID <4:3>	Type
00	Reserved
01	AXPvme Single-slot
10	AXPvme Dual-slot
11	AXPvme with 512 Kbyte Cache

1.10.2.3 Present Bits <7:5>

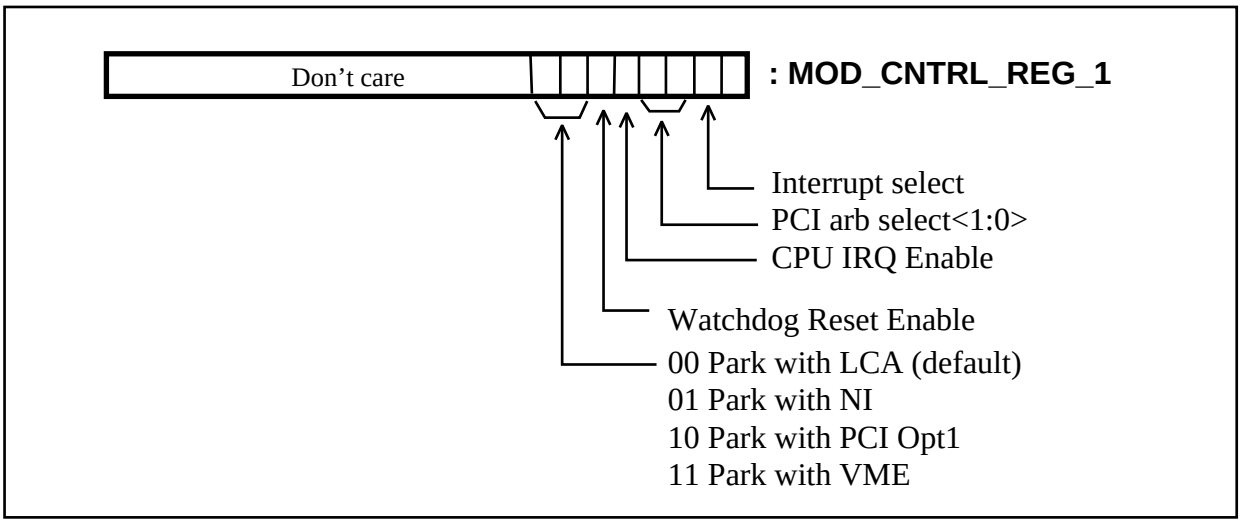
A one in any of these bits indicates that the corresponding function is present. Bit <5> set implies that Termpower is supplied to P2 from the SCSI breakout card. Bit <6> set indicates that optional Bcache and SCSI interface are installed on the module. Bit <7> set indicates that a PCI mezzanine option card is installed on the module.

1.10.3 Module Control Register 1

Base address = MOD_CNTRL_REG = 2C00h

The module control register 1 is a read/write register for controlling miscellaneous module functions. This register is reset to zero on ANY system reset. Figure 1–38 shows the Module Control Register.

Figure 1–38 Module Control Register 1



1.10.3.1 Park Device Select Bits <7:6>

These two bits select the device with which the idle PCI bus will be parked. Reset default is the LCA. See Section 1.4.2 for more information.

1.10.3.2 Watchdog Reset Enable Bit <5>

When this bit is zero, watchdog expiration has no effect. If bit <5> is set and the Diagnostic In Progress (DIP) bit of the Reset Reason register is cleared, a watchdog expiration will generate a hardware reset of the module. Reset default is disabled.

1.10.3.3 CPU IRQ Enable Bit <4>

This bit must be set to enable interrupts (from SIO, VIC, and Halts) to the processor. Reset default is disabled.

1.10.3.4 PCI Arbitration Selection Bits <3:2>

These two bits will allow limited control over the PCI arbitration priority assignment based on the following scheme:

Priority	<3:2>=00	<3:2>=01	<3:2>=10	<3:2>>=11
1	LCA	PCI	VME	Reserved
2	ENET	LCA	LCA	Reserved
3	SCSI	VME	PCI	Reserved
4	PCI	ENET	ENET	Reserved
5	VME	SCSI	SCSI	Reserved

1.10.3.5 Interrupt Select Bit <1>

This bit selects the source of SIO IRQ <11:9>. If clear (the default), VMEIRQ <3:1> are selected. If set, PCI_OPT <D,C,B> are selected. See Figure 1–64 for details.

1.10.4 Module Control Register 2

Base address = MOD_CNTRL_REG_2 = 3400h

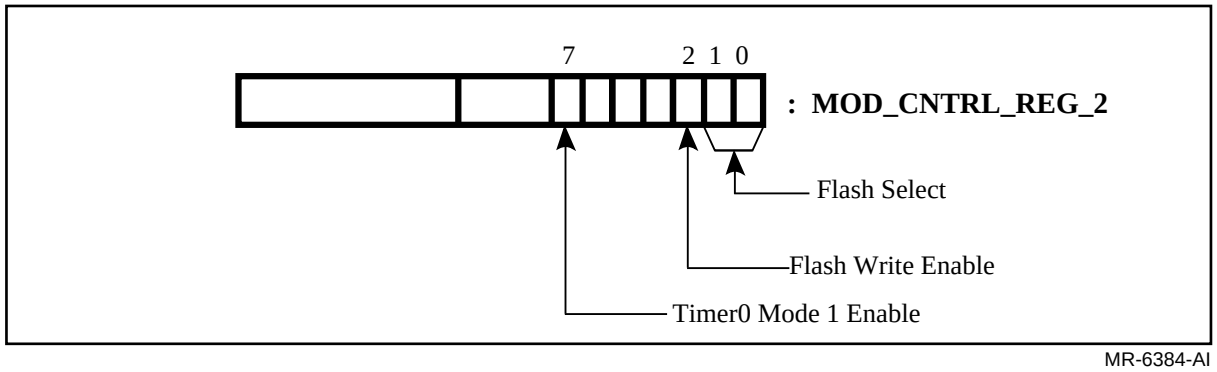
The module control register 2 is a read/write register for controlling miscellaneous module functions. This register is reset to zero on ANY system reset. Bits <6:5:4:3> of this register are reserved.

Note

The module control register 2 is not implemented on the AXPvme 64, AXPvme 64LC, and AXPvme 160 modules.

Figure 1–39 shows the Module Control Register 2.

Figure 1–39 Module Control Register 2



1.10.4.1 Flash Select <1:0>

These two bits default to <00> at power up, selecting the device containing the console image in the bottom 512 Kbyte. The upper 512 Kbyte of this device is available as user flash. <01>, <10>, and <11> select the other three user flash devices, for a total of 3. 5 Mbytes of user flash.

1.10.4.2 Flash Write Enable <2>

This bit defaults to 0 at power up. Setting this bit to 1 asserts "write enable" to the four flash ROMs, allowing them to be written. This bit should be left cleared when not updating the flash ROMs to avoid corrupting the flash ROMs accidentally.

1.10.4.3 Timer 0 Mode 1 Enable

This bit is cleared on power up. When cleared, timer 0 in the 82C54 can only operate in modes 0 and 3. Setting this bit to a 1 inverts the polarity of the TIMER0 gate input of the 8254 timer chip, allowing proper operation in modes 1 and 5. The default behavior of timer 0 is exactly like the AXPvme 160.

1.10.5 Reset Reason Register

Base address = 3000h.

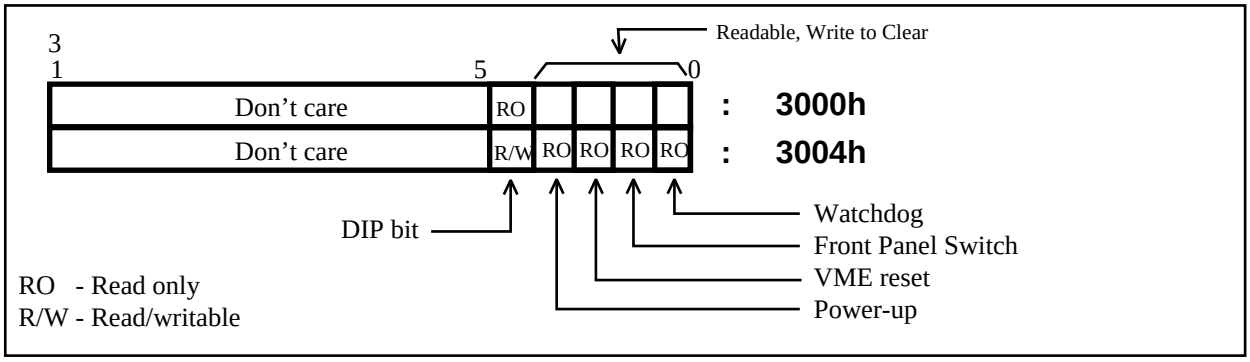
The Reset Reason register is a read/pseudowritable register located at a fixed address on ISbus in PCI I/O address space. It records the cause of a module reset. The reason for a module reset can be any of Power-up, Front panel switch, Watchdog, or VMEbus reset.

The register contains the four reset status bits and one read/writable bit, designated DIP (Diagnostics In Progress).

The register is aliased in two adjacent longwords. At PCI I/O address 3000h, all five bits are readable, but ANY WRITE will clear bits <3:0>. At this location, bit <4> is read only.

At PCI I/O address 3004h, bits <3:0> are read only and DIP is fully read/writable. Figure 1–40 shows the Reset Reason Register.

Figure 1–40 Reset Reason Register



It should be noted that if the "Power-up" bit is set, all other bits should be ignored.

When a watchdog timeout occurs, the Watchdog bit is set immediately so it is available to indicate the HALT reason before the system is actually reset. In this case, the register forms part of the halt reason information in the system (see Section 1.17). If the DIP bit is set, the system will not be reset. At PCI I/O address 3024h, any write will set all bits <4:0>. This is for test ONLY.

1.10.6 Heartbeat "Clear-Interrupt" Register

Base address = HBEAT_CLR_REG = 2000h.

When the heartbeat clock is enabled in the TOY chip, each active (low to high, at a frequency of 1024 Hz) transition causes the heartbeat status bit to be set. The bit is not directly readable but directly drives the heartbeat interrupt line into the SIOs IRQ1 pin.

Writing (data independent) to the Heartbeat "Clear-Interrupt" register clears the heartbeat status bit and dismisses the interrupt request. See Section 1.17.

1.10.7 Front Panel Status LEDs

The front panel of the AXPvme has two discrete LEDs showing module status.

1.10.7.1 The AMBER LED

The left LED shows the status of the Watchdog bit in the reset reason register. When the bit is set, by the expiration of the watchdog, the LED is lit. When the bit is cleared by firmware or software writing to the reason register, the LED will be extinguished.

This LED can also be used as a quick check on the slave selection of AXPvme. It is lit during a valid slave select from the VMEbus. The LED will flash on for 350 ms each time AXPvme is accessed as a slave. Moderate to heavy VME slave activity will result in the LED appearing continuously lit.

1.10.7.2 The GREEN LED

The second LED shows that onboard 5 V is within specification. If the sensed voltage drops below 4.5 V, the LED will go out and the module will be reset. The module will be held in reset until the 5 V supply returns to greater than 4.5 V.

1.11 ROM

The AXPvme has two ROM structures.

The first is the processor startup Serial ROM (SROM), which contains 8K of code serially loaded into the LCA's internal Icache on powerup.

The second is system ROM, which is accessible over the ISbus. This ROM is implemented using Flash devices allowing in-place updating (module switch 2 allows updates to be disabled).

1.11.1 Serial ROM

This 8K of SROM is copied into the processor's instruction cache on reset. Execution control is passed to this code in PAL mode. The function of the SROM code can be broadly described as:

- Verify the processor's operation
- Identify the reset type
- Find 2 Mbytes of good memory
- Identify the ability to read system ROM (checksum)
- Decompress 512K of ROM (init code) into memory
- Transfer control to init code

This SROM is socketed to allow future firmware upgrades. See Firmware documentation for more details.

1.11.2 System ROM

Base address (PCI memory space) = ROM_BASE_ADDR = 00000000h.

The flash ROM is accessible as a contiguous 1 Mbyte in PCI memory space. Only BYTE accesses to the ROM are supported. For AXPvme systems with 4 Mbytes of flash ROM, the ROM is bank switched in 1 Mbyte banks using bits <1:0> of the module control register 2.

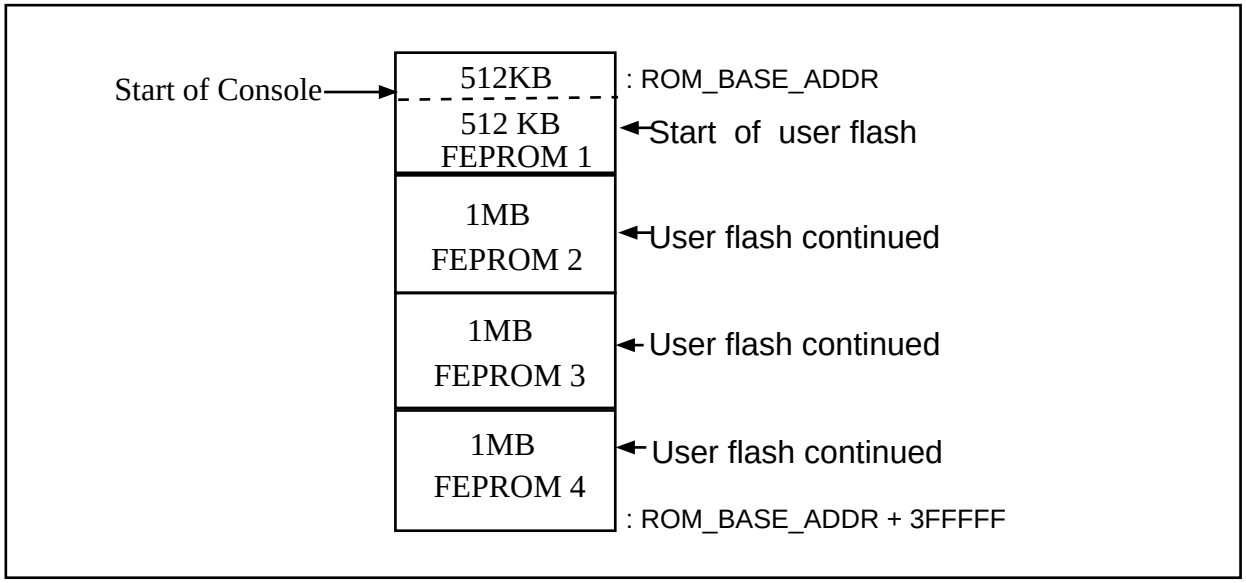
1.11.3 Flash ROM Updating

The flash ROM space can be rewritten. Switch 2 on the module enables flash devices for update. The flows outlined below assume Vpp programming voltage is applied (switch 2 closed). The first 512 Kbytes of flash ROM are reserved for console use Figure 1–41). The remaining space in the flash ROM is reserved for onboard User code. Refer to the *update* command in Chapter 3 for programming these devices.

Note

If the processor has 1 Mbyte of flash ROM, it is divided as shown by FEPROM 1 in Figure 1–41 and uses only that block. If the processor has 4 Mbyte of flash ROM, it uses all four blocks of user flash ROM.

Figure 1–41 Flash ROM Layout/Addressing



1.11.4 Write Protect

Flash ROM is protected from unauthorized/accidental updates by a hardware switch on the AXPvme module. E44 switch 2 must be closed to connect the proper programming voltage to the flash ROM devices. Switch 2 should ALWAYS be open when Flash is not being updated.

Flash ROM updates are also protected via an onboard software enable. The UART channel A DTR bit must be asserted to allow writes to the flash ROM devices.

1.12 Console UART

AXPvme provides two general-purpose asynchronous serial communication lines. These lines are data-lead only DEC423 ports. One of the two ports is dedicated to the system console.

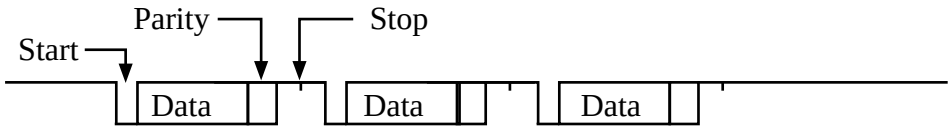
The serial interface is based on the 85C30 serial communications controller (SCC) chip. AXPvme has a single 85C30 with the required line drivers and receivers, with onboard ESD/EOS protection.

The physical connection to the serial ports is via two front panel 6-pin MMJ connectors. The MMJ pinning is the DEC STD 52 standard pin-out.

1.12.1 UART Operation

The 85C30 SCC provides two independent full-duplex channels, programmable for use in any common asynchronous (or synchronous) data-communication protocol. AXPvme will only support asynchronous use of the SCC, therefore only asynchronous operation (see Figure 1–42) of the 85C30 will be presented in this specification. For information on the other modes of SCC operation, refer to the 85C30 chip specification and application notes. But remember, only data leads are directly accessible from the AXPvme front panel.

Figure 1–42 Typical Asynchronous Protocol



The operation of each of the two SCC channels (referred to as channel A and channel B) is completely independent. In effect, the SCC is two separate UART controllers in a single package. For this reason, the operation of one channel is described, from which the operation of the other can be inferred.

For each channel, transmit and receive blocks are independent (full-duplex) with independently programmable character size (5 to 8 bits), parity (off, odd or even). However, transmit baud rate must be the same as the receive baud rate.

The transmit block can supply one, one-and-a-half, or two stop bits per character and can provide a “break” output at any time.

The receive channel has a three-character FIFO buffer. Framing and overrun errors are detected and buffered together with the partial character on which they occurred. The receiver can also be enabled to detect “break” characters. Any of these conditions can be enabled to generate a CPU interrupt request.

1.12.2 Data/Register Access

Base address = UART_BASE_ADDR = 6000h.

A single channel of the SCC is accessed via the least significant byte of two contiguous longwords in PCI I/O space. These two bytes allow access to the 15-plus internal control and status registers of the device as well as to the transmit and receiver buffers. The base of the UART is fixed in the SIO region of the PCI I/O space. For the purposes of this discussion, everything will be referenced to the UART_BASE_ADDR.

Accesses to the SCC can be divided into two broad types.

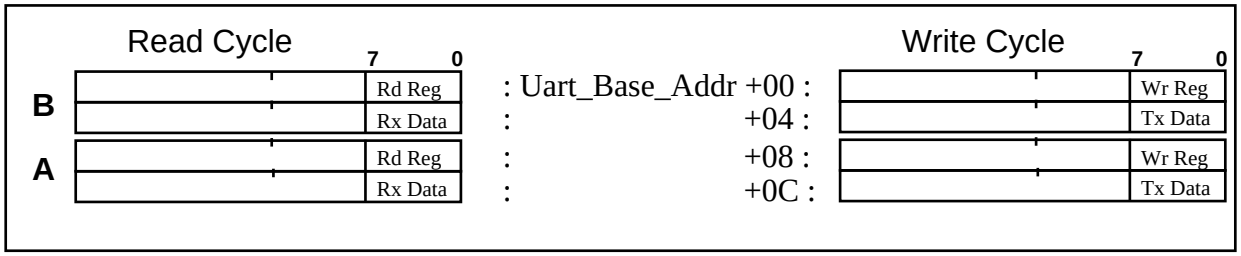
1.12.2.1 Serial Line Data

The first type of access is a serial line data operation. To write a character out, the data is simply written to the TX data register. A character received from the serial can be retrieved from the UART by reading the RX data register. These two data registers are colocated in the first byte of the SCC interface and are selected by the read or write operation. For example, to send a character, simply doing a write to the channels data register while reading from the same address will return a character from the receive buffer.

1.12.2.2 Internal Registers

The second type of access is used to get at SCC internal CSRs. Though there are 8 readable and 15 writable registers for a single channel, they are accessed via a single byte in the SCC address space. Read registers will be selected for bus read cycles and write registers are accessed when the cycle to the SCC are writes. Figure 1–43 shows the SSC memory map.

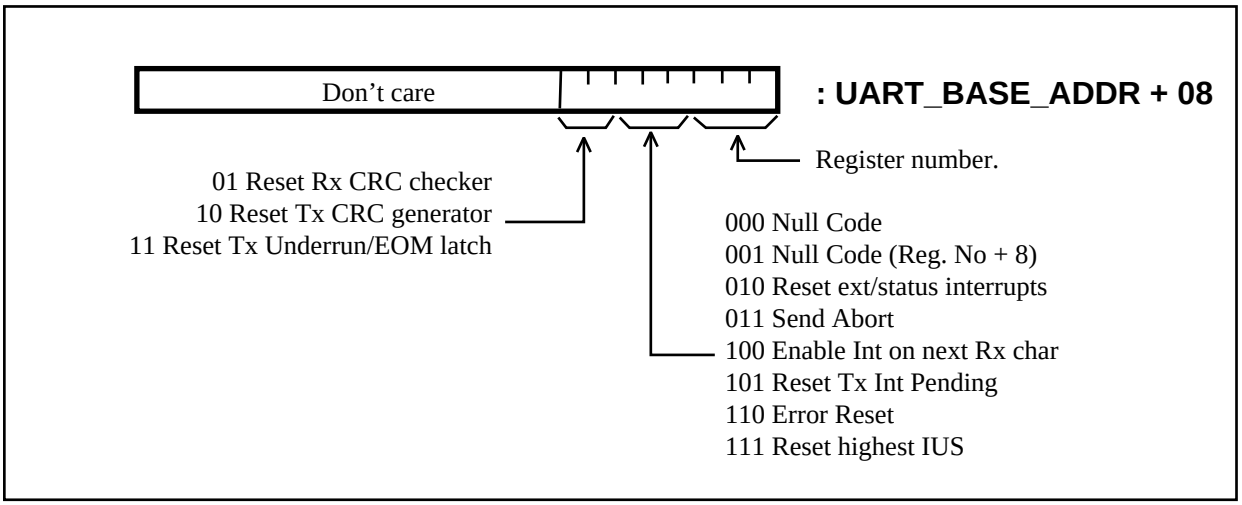
Figure 1–43 SCC Memory Map



Only Read Register 0 (RR0) and Write Register 0 (WR0) are directly accessible via this single bytewide address location. In order to get at any of the other registers, first a write to WR0 selects the target register, which can then be accessed by a subsequent read or write.

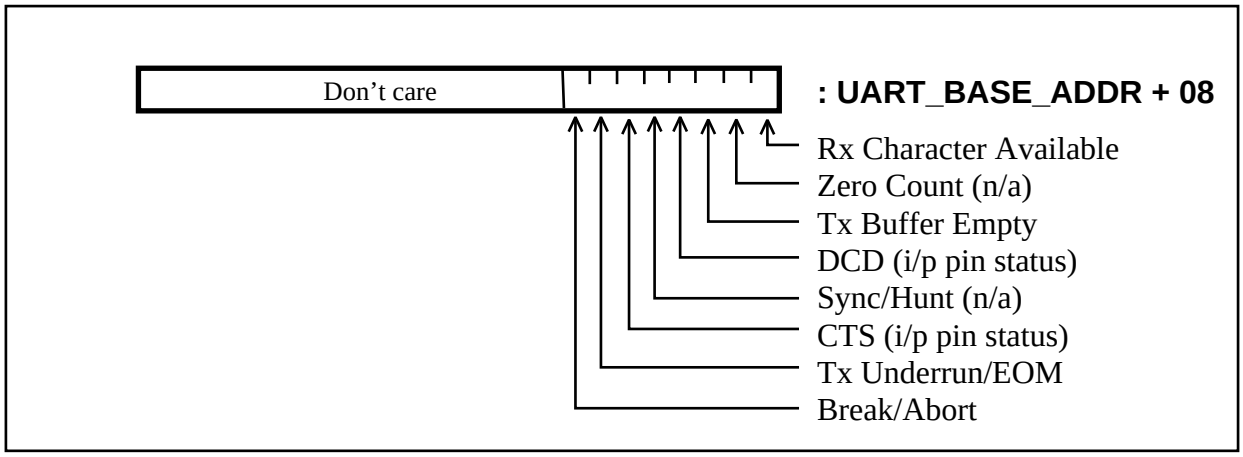
Write Register 0 (WR0) is used to configure some global control state of the SCC and/or to set the register number for the next read or write access. Figure 1–44 shows the format of WR0. Note that if bits <7:4> are zero, the lower 4 bits can be used to set up for a subsequent access to read or write registers other than WR0 or RR0.

Figure 1–44 Write Register 0 (Channel A)



In the same way that WR0 is directly accessible, normal SCC status can be read by a simple read from the same address. Figure 1–45 shows the format of RR0.

Figure 1–45 Read Register 0 (Channel A)



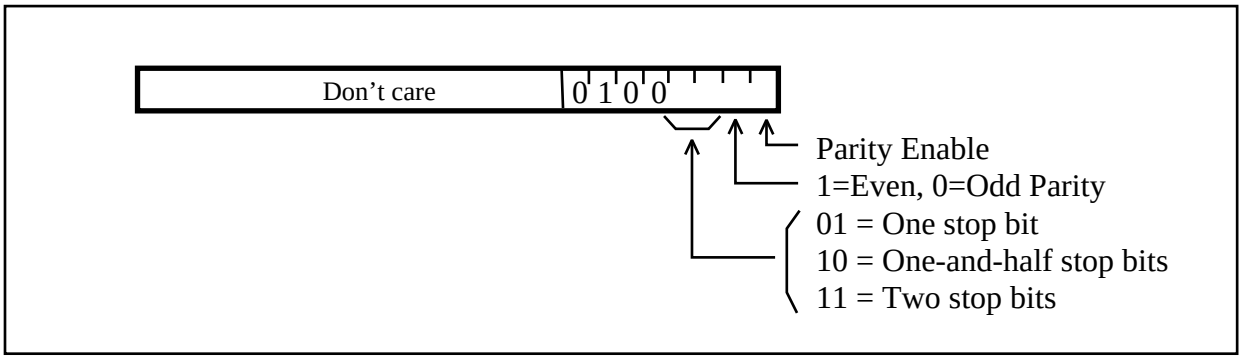
The sequence to access any other register is a simple two-stage operation.

1. Write WR0 with the desired register number.
2. The next (one) access, read or write, will operate on the specified internal register.

1.12.3 SCC Operation in Asynchronous Mode

The operation mode of the SCC is selected by writing the WR4 (see Figure 1–46). As synchronous modes are not to be enabled for the AXPvme usage of the 85C30, bits <7:4> should be 0100 to select the x16 clock mode.

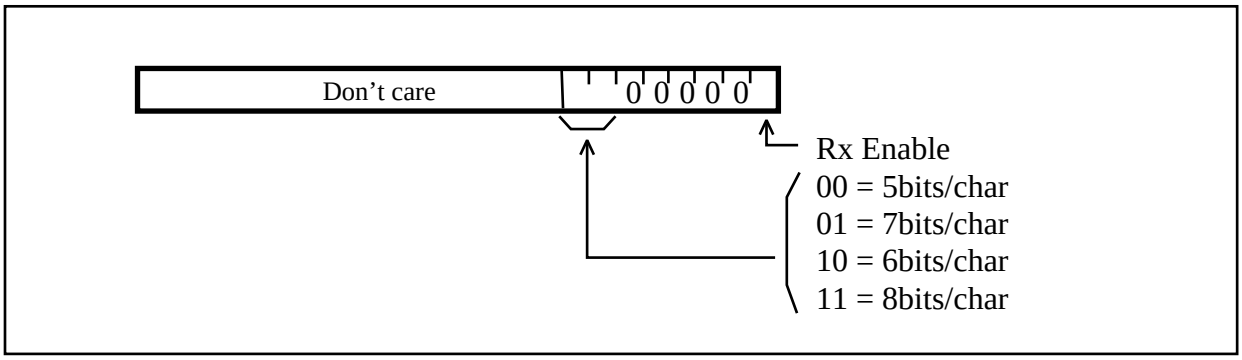
Figure 1–46 Write Register 4



1.12.3.1 RX Operation

The SCC receiver block is configured by Write Register 3 (see Figure 1–47). In particular, for asynchronous operation, bits <5:1> must be zero, while bit <0> enables the RX circuit and bits <7:6> select the character size (bits per character).

Figure 1–47 Write Register 3

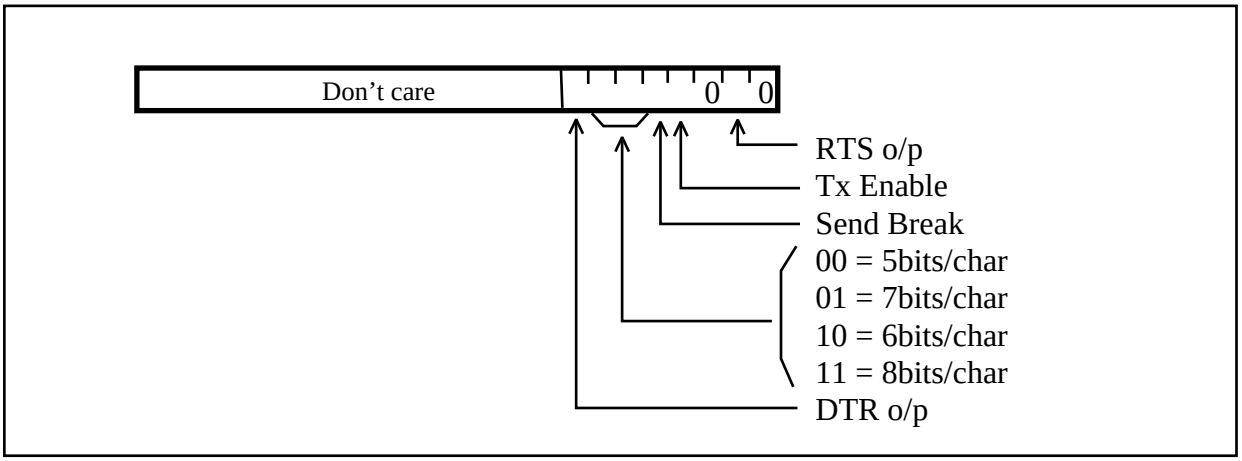


1.12.3.2 TX Operation

Write Register 5 allows the transmit data size to be set up. In addition, if a "break" character is to be transmitted by the SCC, it is accomplished by writing bit <4> of WR5 (see Figure 1–48).

Finally, this register allows the RTS and DTR signal pins to be driven by software, but these bits are write only (there is no way to read the state of drive for RTS or DTR). AXPvme uses channel A DTR as a software controlled flash ROM write enable. RTS is unused in either channel and channel B DTR is unused.

Figure 1–48 Write Register 5



1.12.3.3 Baud Rate Control

The AXPvme 85C30 has a single external clock source to drive the internal Baud Rate generator. Write Register 11 is used to select the clock source for the TX and RX sections of the device. AXPvme's only connected clock source is the 16 MHz oscillator connected to the RTxC inputs for both channel A and B. Thus for correct operation of the SCC, the Baud Rate generator must be enabled in WR14 and selected as the clock source for both the RX and TX channels in WR11.

Simply write WR11 = 56H and WR14 = 03H to select and enable the BR generator as the data clock for both the receiver and transmitter. Note that the bits controlling the Internal Loopback and Auto-echo diagnostic features of the SCC are also contained in WR14.

The output of the baud rate generator can be programmed to scale down the input clock to attain the clock frequencies for desired data transfer rates. The divide is set up via a 16-bit time constant register. This 16-bit register is programmed in 8-bit chunks. WR12 is the low 8 bits of the time-constant while WR13 holds the top byte.

The formula relating baud rate to the programmed TC is given by:

$$\text{Baud Rate} = \frac{1}{2(\text{Time Constant} + 2)\ (\text{BR Clock period})}$$

Where "BR Clock period" equals Clock mode/Clock Frequency. The fixed input clock for AXPvme is 16 MHz. Table 1–14 gives the time constant value for standard baud rates with a "x16" Clock mode selected in WR4 (bit <7:6> = 01).

Table 1–14 SCC Baud Rates

Baud Rate	TC (dec)	TC (hex)
19200	24	0018
9600	50	0032
4800	102	0066
2400	206	00CE
1200	415	019F
600	831	033F

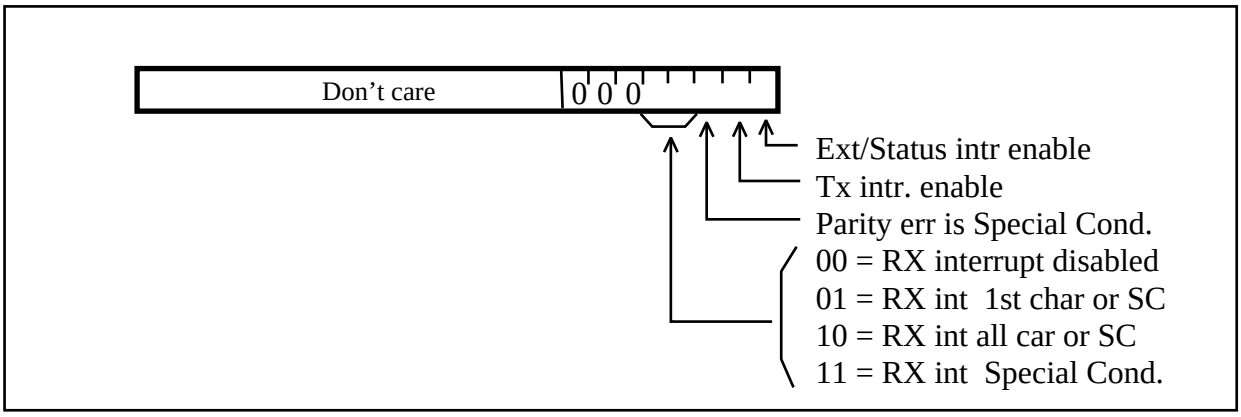
1.12.3.4 Interrupt Generation

There are three types of events that can be enabled to cause an interrupt request.

- Transmit event
  The only transmit event is that the TX buffer has become empty.
- Receive event
  The receiver can be configured to interrupt on:
  - 1st RX character or special condition
  - All RX characters or special condition
  - Special condition only (RX overrun, framing error, optionally parity error)
- External/Status events
  - Transitions on CTS, DCD, and SYNC (all n/c in AXPvme)
  - Break detect
  - TX underrun
  - Zero count in the baud rate generator

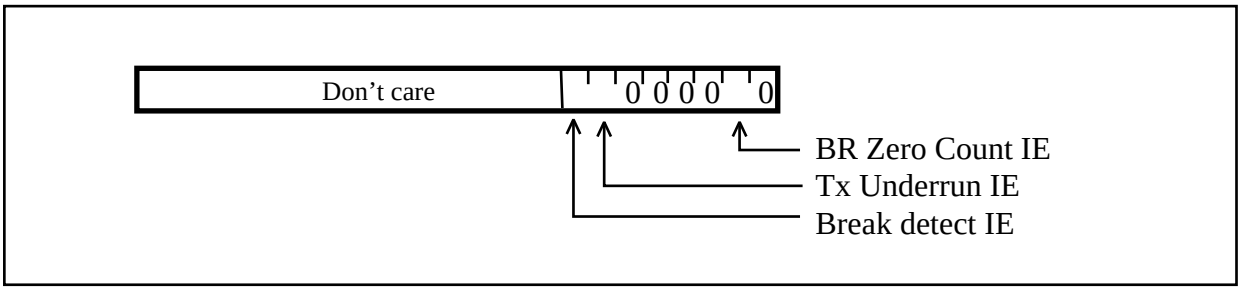
Each interrupt source has an interrupt enable bit associated with it. These bits are write-only and are accessible in WR1 and WR15. Figure 1–49 shows the Interrupt Control Register.

Figure 1–49 Interrupt Control Register (Write Register 1)



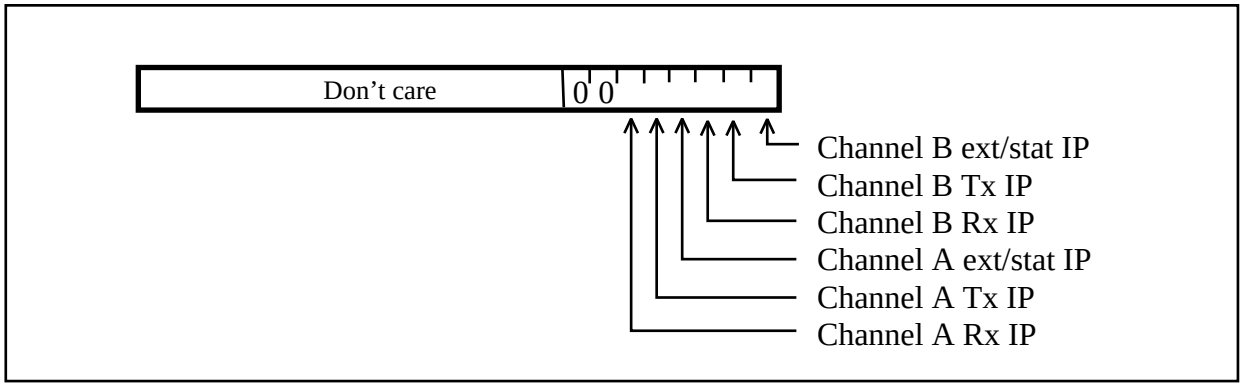
Write register 15 (see Figure 1–50) controls the enabling of the external/status interrupt sources.

Figure 1–50 External/Status Interrupt Control (Write Register 15)



Read Register 3 (see Figure 1–51) in Channel A ONLY, gives a summary of pending interrupts. This information, along with the status from RR0, allows system software to determine the full state of the SCC.

Figure 1–51 Read Register 3 (Interrupt Pending Register)



The programmable vector in the 85C30 device is not used in the AXPvme system. The interrupt request asserted by the SCC in the AXPvme system is asserted to, and handled by, the defined system interrupt controller. See Section 1.17 for more details.

In order for interrupt operation to be used in the SCC, the Master Interrupt Enable bit <3> in WW9 must be set. This register also allows software to reset either Channel of the SCC individually or to drive the entire device into reset.

1.12.3.5 Read Registers

In addition to RR0 and RR3 described above, there are six other readable registers implemented in the SCC. RR10, RR12, RR13, RR15, and RR2 are read-back registers for WR10, WR12, WR13, WR15, and WR2 (WR/RR2 is the programmable interrupt vector not used in AXPvme), while RR1 contains the RX overrun error status bit (<5>) and the parity error bit (<4>).

1.12.4 System Use and Firmware

Channel A will be used for the AXPvme user console. As such, it will be configured by firmware as an asynchronous line with baud rate, parity, data bits, and stop bits configuration definable by the user and stored in NVRAM.

In the absence of valid data in NVRAM on powerup, channel A will be programmed with a default of 9600 baud, 8-bits, no parity, one stop bit.

Channel B is uncommitted and uninitialized by system firmware.

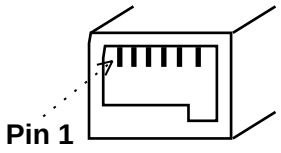
1.12.5 Physical Details

The physical connection to the two serial lines is by way of the standard DEC423 MMJ connector (see Figure 1–52) on the AXPvme front panel (see Section 1.20 for the layout of the front panel).

The MMJ pinout is per DEC STD 52.

Pin No:	Function	Notes
1	Ready out	Tied High
2	TX +	–
3	TX -	Tied Low
4	RX -	–
5	RX +	–
6	Ready in	Not used

Figure 1–52 DEC423 MMJ Connector



1.13 TOY Clock

A Dallas Semiconductor DS1386 chip is used to implement the time-of-year (TOY) clock function in AXPvme.

Timekeeping information includes 1/100ths of seconds, seconds, minutes, hours, days, date, month, and year. The date at the end of the month is automatically adjusted for months with less than 31 days, including corrections for leap years. The stored time can be selected for 24 hour or 12 hour with AM/PM indication formats.

Timekeeping functionality is maintained in the absence of Vcc by an internal lithium energy cell, which has an active life of at least 10 years. In addition, the device internally protects against spurious accesses during power transitions.

Timekeeping accuracy is better than +/-1 minute/month at 25°C.

Access to the TOY, to examine or set current time, is via 9 bytewide registers in ISbus space. Note that the alarm features of the DS1386 are not supported in AXPvme. The watchdog and SRAM functionality are supported and are described in Section 1.15 and Section 1.16. The square wave output is used to generate a fixed 1024 Hz interval interrupt.

This part will be socketed to allow:

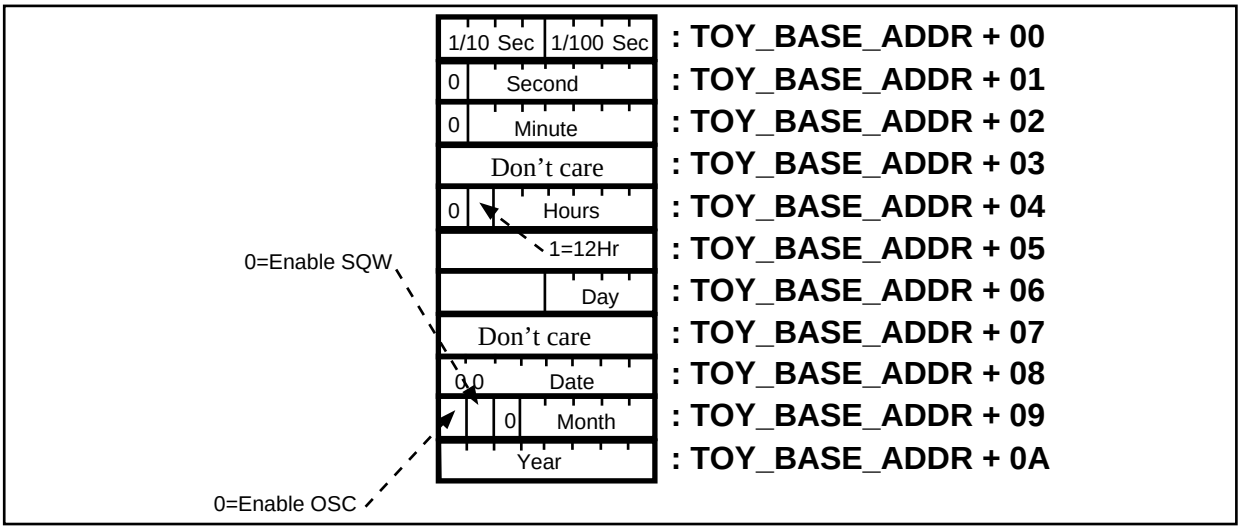
- Physical removal of the NVRAM from one AXPvme module to another
- Replacement of a DS1386 in which the internal power source is no longer functional

1.13.1 TOY Clock Operation

Base address = TOY_BASE_ADDR = 8000h.

The DS1386 is interfaced to the AXPvme system on the ISbus. Time information is contained in 8x8-bit read/write registers offset from this base address (see Figure 1–53).

Figure 1–53 TOY Time Registers

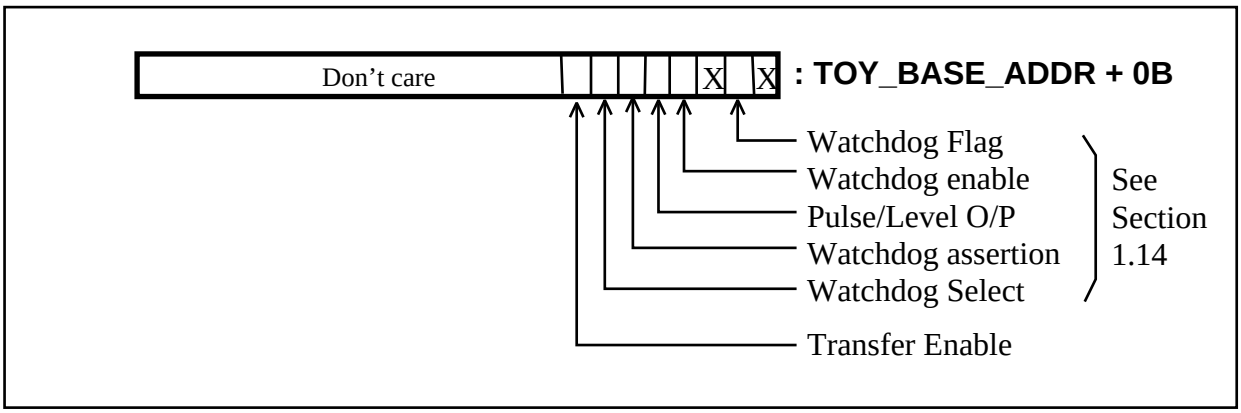


The time is stored in BCD. For example, a time of 29 minutes is stored in location (TOY_BASE_ADDR+02) as 29h.

The storage format of "hours" depends on the setting of the 12/24 hour mode bit (bit <6> of TOY_BASE_ADDR+04). When this bit is clear, the TOY stores the time hours as BCD from 00 to 23, however, in 12-hour mode (bit <6> = 1) the hours are 01 to 12 with bit <5> indicating AM (0) or PM (1).

The final register of interest in the DS1386 in relation to timekeeping is the Command register (see Figure 1–54), located at TOY_BASE_ADDR+0B.

Figure 1–54 TOY Command Register



The TOY read/write registers are updated once every 0.01 seconds. This update process is asynchronous to any access to the TOY so the register values may change during the read or may get incorrectly updated during a write. To overcome this problem, use the Transfer Enable in the TOY Command Register. When this bit is clear, the current value in the readable registers is frozen (even though the internal timing continues).

Thus, to read the TOY, disable register transfers by clearing bit <7>, read the time registers, and reenable updating (bit <7> = 1). To reset current time, once again disable the transfer mechanism by clearing bit <7>, write the new time value into the registers, and reenable time transfers to execute the update.

The TOY chip internal oscillator can be disabled to conserve the lithium source during transport, storage, or during any long period of non-use. The Oscillator Enable Bit is bit <7> of the "month" register at TOY_BASE_ADDR+09. When this bit is a zero, the TOY operates in full timekeeping mode. If set, the internal oscillator is disabled (factory default).

1.13.2 Fixed Frequency "Heartbeat" Output

The fixed frequency square wave output is enabled by clearing bit <6> of the "month" register at TOY_BASE_ADDR+09. This bit must be cleared to use this clock as the heartbeat interval timer interrupt delivered through the SIO's internal PIC.

1.13.3 Standby Power

Even though the DS1386 chip has an on-chip power source for operation of at least 10 years in the absence of external power supply, some applications may require the TOY (and SRAM) to operate from an external UPS. To this end, an AXPvme onboard switch (E44 switch 1) is included to allow a selectable connection of the TOY to the 5 V standby connection on the VMEbus (5VSTDBY). When switch 1 is closed, VME 5VSTDBY is connected to the TOY supply through isolation diodes.

1.14 Interval Timers

AXPvme's timer/counters are based on the 82C54 device. This section briefly outlines the 82C54 operation (for more detail, see the vendor/DEC chip specification) and emphasizes the hook-up of each of three timer elements.

The 82C54 is made up of 3 independent but identical 16-bit counter/timers. These timers are named Timer #0, #1, and #2.

Timer #0 must be externally clocked via P2 pin C17, optionally its gate input can also be driven by an external source via P2 pin C18. The output of timer #0 causes the assertion of an interrupt request when it makes a low-to-high transition. The IRQ can be dismissed by an access to the timer interrupt status register.

Timer #1 is configured to operate as a rate generator with its output being driven off module via P2 pin C16. This timer is clocked by a fixed 10 MHz. Note that the output of this timer is also routed directly to VIC local IRQ input <3>.

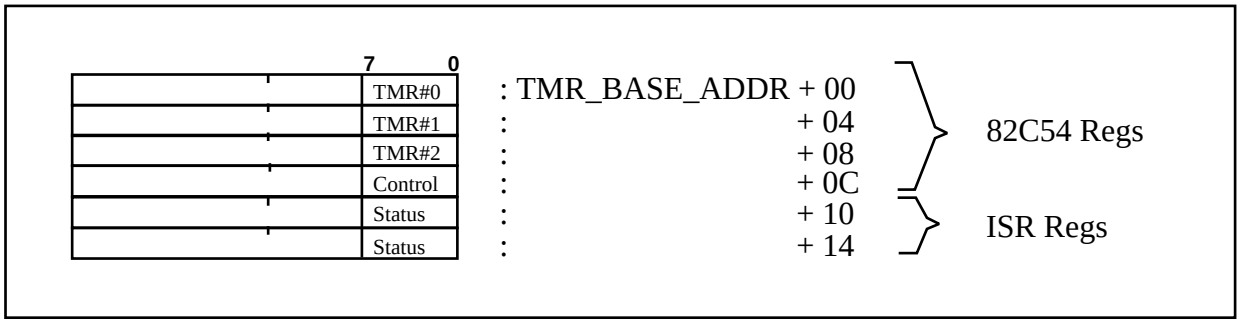
Timer #2 is configured to operate as a rate generator with the same 10 MHz input clock and its output is connected to P2 pin C15. The output of timer #2, however, can also be used on module to generate an interrupt request. Low-to-high transitions of the timer output can be enabled to cause the timer interrupt to assert. As with timer #0, the interrupt can be dismissed with an access to the ISR.

The timers are implemented by a single 82C54-2 and some register/interrupt logic. The programming interface is bytewide in the ISbus region of PCI I/O space.

Base address = TMR_BASE_ADDR = 4000h.

The timer interface takes up the least significant byte of six adjacent longwords in ISbus space (see Figure 1–55). The first four are the standard four bytewide registers of the 82C54, while the other two bytes are an interrupt status register (described later).

Figure 1–55 Timer Memory Map



On powerup, the 82C54-2 is in an undefined state and must be initialized before use.

1.14.1 82C54 Operation

To program the timer device for initialization or during normal operation, the control byte (TMR_BASE_ADDR + 0C) is written. To access (read or write) the individual timer count values, the separate timer data registers are used (TMR_ BASE_ADDR +00 to +08).

The three timers in the 82C54 are identical in function yet are fully independent.

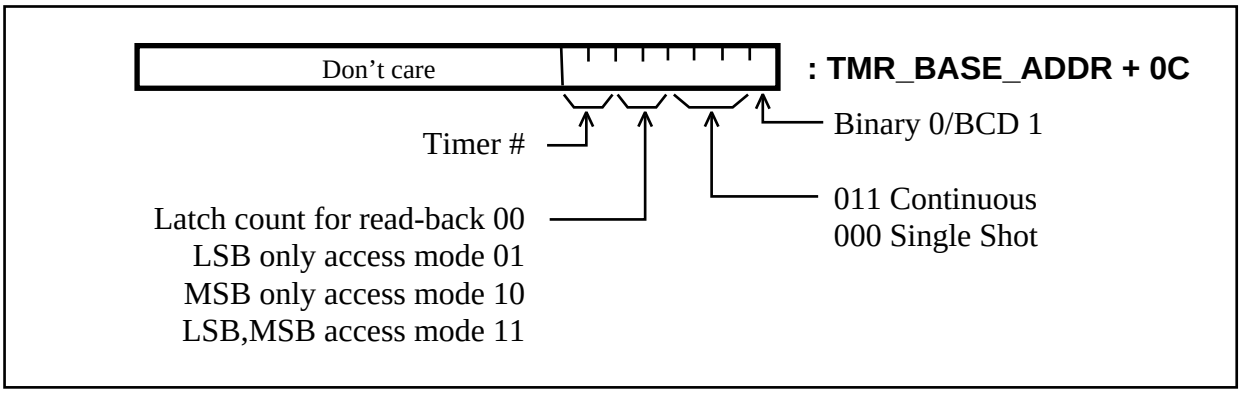
Each timer element is a 16-bit presettable synchronous down counter. Typically, the device will assert or pulse the corresponding output pin when a counter reaches a zero count.

As mentioned above, the timer has only a single byte in the 82C54 address space dedicated to it. This byte is used to access the full 16-bit counter value, thus two accesses are required, in the form least-significant byte, most-significant byte, to operate on the full 16 bits. Sometimes it may only be required to modify or read only the LSB or MSB without affecting the other byte. This mode of operation is also supported.

1.14.1.1 Control Byte

The control byte (see Figure 1–56) allows basic mode selection and access control operations to be performed on individual timer elements.

Figure 1–56 82C54 Control Byte



Bits <7:6> define which timer (0, 1, or 2) is to be configured by this control byte. Note that a "11" in this field signals the control byte as a Status Read command (described in Section 1.14.1.3) rather than a Timer Control operation.

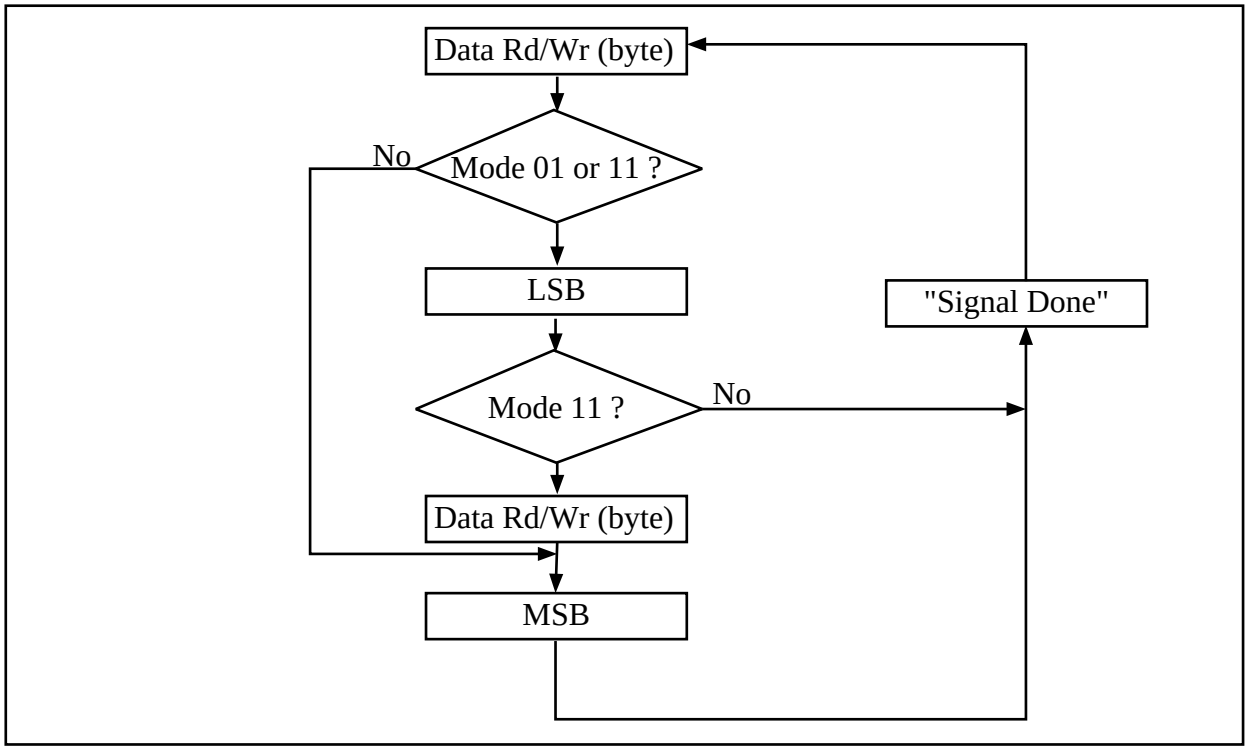
Bits <3:1> configure the operational mode for the timer referenced in bits <7:6>. Only modes 0 and 3 are supported by the AXPvme 64, AXPvme 64LC and the AXPvme 160. all other AXPvme modules support modes 1 and 5 by <7> of the control module register 2.

Bit <0> sets the timer's 16-bit counter to work in either binary or BCD.

Bits <5:4> configure the data interface to expect one or both of the bytes of the 16-bit counter to be accessed when a read or a write to that timer is executed. Note that once the access mode is set, all operations to the data register of the timer will be in the format set until a new mode is set with another control byte to that timer.

Figure 1–57 shows a conceptual view of the operation of the timer bytewide data interface. Note that the "signal done" action is important where the completion of a data access becomes an implicit start/go command to the timer.

Figure 1–57 82C54 Timer Data Access



1.14.1.2 Timer Modes

Although the 82C54 itself supports 6 counting modes (modes 0–5), the AXPvme implementation adds it own set of restrictions. Most of these arise from the resolution of the synchronizing logic used after the 82C54. This logic sets the minimum pulse width any output of the 82C54 can generate and still be recognized by onboard status registers. Additional constraints are required to allow for the hardwired "high" gates of timers 1 and 2. Table 1–15 outlines the various counting modes and restrictions relevant to the AXPvme implementation.

Table 1–15 Timer Modes

Mode	Description	Restrictions	Timers
0	Software retriggerable one-shot timer	N>=3	1, 2
1	Hardware retriggerable one-shot timer	N>=2, CLK<3 MHz	0 only[1]
2	Periodic rate generator	Do not use	None
3	Periodic square wave generator	N>=5	1, 2
4	Software triggered strobe	Do not use	None
5	Hardware triggered strobe	CLK<3 MHz	0 only[1]

[1]See Section 1.14.2.4 for Timer 0 restrictions.

Note that for timers #0 and #2, which can cause timer interrupts through either the SIO or VIC64 (reported through the timer interrupt status register), an output low-to-high transition is considered to be the timer expiration causing a status bit to be set and, if enabled, the interrupt request to be asserted. This is important to remember when working with the pictures in the 82C54 specification.

Timer #1 can cause an interrupt through the VIC64 Local IRQ3 only. While the VIC64 can be programmed to accept either assertion level at it's Local IRQ input, it is normally configured to generate an interrupt on the rising edge of timer #1 output.

Mode 0 - Software Retriggerable One-Shot

This mode allows a value to be written to the timer, which will then count down, asserting the output (high) when it reaches zero. In this mode, it will take N+1 clock ticks from the end of the counter value write cycle until the output makes its active transition.

The timer output is initially high. When the timer value is written, the output is driven low. The counter decrements to zero where it drives the output high.

If a new count value is written during the counting sequence, it will be loaded on the next clock pulse and counting will continue from the new value. This means the count is software retriggerable.

Mode 1 - Hardware Retriggerable One-Shot

This mode allows a value to be written to the timer which will be used when a hardware trigger has been received [TMR_MAJOR_IP L (P2 pin C18) transitions from a high to a low].

The timer output is initially high. A trigger results in loading the Counter and setting the output low on the next clock pulse, thus starting the one-shot. An initial count of N will result in a one-shot pulse of N clock cycles in duration. The output will be driven high when the counter reaches zero.

The one-shot is retriggerable, hence the output will remain low for N clocks after any trigger. The one-shot pulse can be repeated without rewriting the same count into the counter.

If a new count value is written to the counter during a one-shot pulse, the current one-shot is not affected unless the counter is retriggered. In that case, the counter is loaded with the new count and the one-shot pulse continues until the new count expires.

Mode 3 - Continuous, Square Wave Output

This mode generates a square wave output of period N clock ticks. Typically this is used to generate a rate output or a regular interrupt request to the CPU.

Note that for odd count values, the output will be high for (N+1)/2 and low for (N-1)/2 counts. A count value of 1 is illegal.

For timer #0, the Gate input in this mode has a synchronizing or reset effect. If the gate goes low, the counter is reloaded with its original value and the counting restarts.

Mode 5 - Hardware Triggered Strobe

Placing timer #0 in this mode generates a single clock wide pulse delayed by N+1 clocks. The output will initially be high. Counting is triggered by a high-to-low transition of TMR_MAJOR_IP L (P2 pin C18). The output of timer #0 will go low for one clock period after N + 1 clock pulses. The counting sequence is retriggerable. Timer #0's output will not strobe low for N+1 clocks after any strobe.

1.14.1.3 Status Read

The timer control byte can be used to freeze the state of the timers for read-back. Information pertaining to the assertion state of the output pin, the mode of operation, the read-write access mode, and so forth, is then available by reading the timer data register.

See the 82C54 data sheet for full details.

1.14.2 Interval Timers in AXPvme

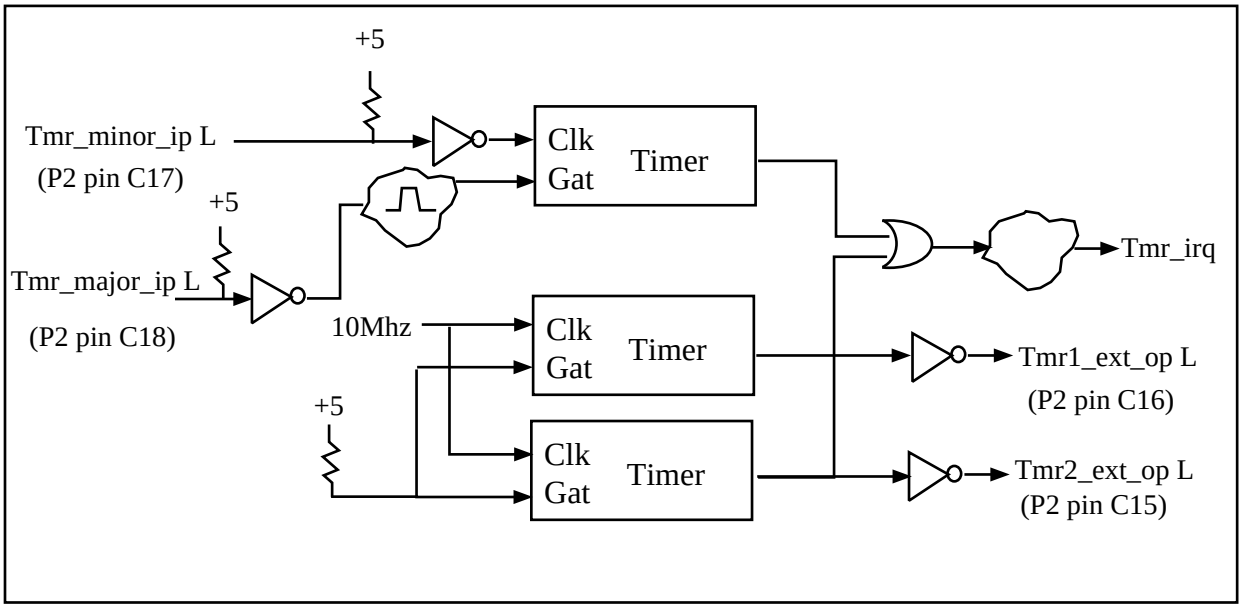
AXPvme allows for a number of configurations using the three independent 16-bit timer units. The expiration of timers #0 and #2 are recorded in a timer status register. The asserted state of either or both of these status bits can be enabled to assert an interrupt request.

The active low outputs of timer #1 and #2 are routed to P2 connector pins. The active low clock and gate inputs of timer #0 are also tied to P2 connector pins.

TMR1_EXT_OP L = P2 pin C16 (Timer #1 output)
TMR2_EXT_OP L = P2 pin C15 (Timer #2 output)
TMR_MINOR_IP L = P2 pin C17 (Timer #0 clock input)
TMR_MAJOR_IP L = P2 pin C18 (Timer #0 gate input)

Figure 1–58 shows the timer inputs and outputs.

Figure 1–58 Timer Clocking



1.14.2.1 Timer Clocking

The clock inputs to timer #1 and #2 are a fixed 10 MHz source. The clock input of timer #0 is from a P2 pin (TMR_MINOR_IP L) only.

The gate inputs for timers #1 and #2 are permanently asserted. This means that 82C54 modes 1 and 5 are disabled on timer #1 and timer #2.

The timer #0 gate input is driven from a P2 pin through some synchronization and edge detect logic. This signal conditioning means that when the gate input to the module makes a high-to-low transition, a synchronized single clock-tick pulse is presented to the gate input of the 82C54 (see details of the 26V12 PAL for exact timing information associated with this gate function).

1.14.2.2 Timer Outputs

The main timer interrupt request line from timers #0 and #2 through the timer interrupt status register logic is routed to both the system interrupt controller in the VIC chip (as local IRQ <4>) and to the SIO interrupt control (as IRQ <5>).

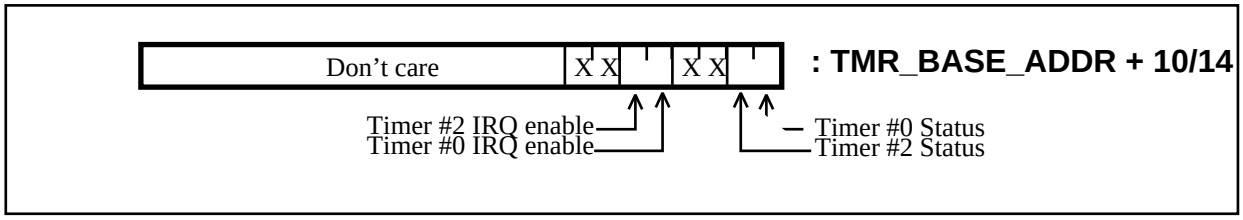
The TMR_IRQ interrupt line is asserted for a low-to-high transition of a timer's output pin when that timer is enabled in the control and status register to cause an interrupt. The interrupt is held asserted until the timer status summary register is read (clear on read). Note that the corresponding timer expiration status bit is always set by a low-to-high on the timer output but this only causes the IRQ line to be asserted if the corresponding interrupt enable bit is set.

In addition, the output of timer #1 is brought to the VIC IRQ <3>. As this is the straight output from the 82C54, the VIC should be programmed for an edge-sensitive input for this interrupt (all other interrupts in the system are level). This interrupt mechanism will not be discussed further here; it should just be noted that the hardware connection is made and it may be useful for some applications in the future.

1.14.2.3 Timer Interrupt/Expiration Control and Status Register

The timer status register is aliased as the bottom byte in two contiguous longwords (as shown in Figure 1–55). The action of the register is slightly different, depending on the address at which it is accessed and whether the access is a read or a write. Figure 1–59 shows the Timer Interrupt/Expiration Status Register.

Figure 1–59 Timer Interrupt/Expiration Status Register



When reading the register, bits <5:4> always reflect the status of the enable bits for timer #2 and #0 respectively. The effect of the read on the output status bits (register bits <1:0>) is different at the two addresses, however. When the timer interrupt status register is read at TMR_BASE_ADDR+14, the read clears both status bits at the end of the read cycle (read to clear). A read from TMR_BASE_ ADDR+10 does not affect any of the bits in the register.

Remember, the interrupt request line is given combinationally by:

```
IRQ = (BIT <0> and BIT <4>) or (BIT <1> and BIT <5>)
```

So it can be seen how the read to clear bits <1:0> dismiss the interrupt.

A status bit set shows that the corresponding timer output line has made an active transition.

Unfortunately, bits <5:4> are not directly writable. Rather, a write to address TMR_BASE_ADDR+10 toggles bit <4> only (all other bits in the register are unaffected) and a write to TMR_BASE_ADDR+14 toggles bit <5> only.

1.14.2.4 Timer #0 Restrictions (AXPvme 64, AXPvme 64LC, and AXPvme 160 modules only)

This section describes the limitations on the use of timer #0 due to hardware constraints. These limitations will be eliminated as new revision components become available. Please contact your Field Application Engineer for the latest status on this constraints.

Timer #0 Does Not Support Modes 1 and 5

Timer #0 does not function properly in modes 1 and 5. These modes are needed to support the "Distributed Timer" functionality across the VME backplane. The circuitry supporting timer #0 will be changing to enable modes 1 and 5. When this change occurs, all modes other than 1 and 5, will be disabled. As a result, this timer should not be used until this problem has been corrected.

1.14.3 1024 Hz Heartbeat

One additional interrupt generating timer exists in the standard AXPvme system.

The 1024 Hz square wave clock output of the TOY is fed through an edge-detecting resettable register to IRQ1 of the SIO. Every time the clock makes a low-to-high transition, the IRQ1 input is asserted and held asserted. The interrupt request input is only deasserted by writing to the "Clear Heartbeat" register at address 2000h on the ISbus. This interrupt input is handled by the PICs internal to the SIO.

1.15 Watchdog Timer

The watchdog timer is included to allow hardware to bring AXPvme back to some known state when software fails to function correctly.

The operation of the timer is straightforward. The watchdog is initialized with some time value (in the range 0.01 to 99.9 seconds). If left unaccessed, the timer will decrement towards zero. If allowed to reach zero, the watchdog will first halt the system (jump to Halt entry firmware) and then force the module into hardware reset (some 300 ms later). The module can be "kept-alive" by periodically accessing the watchdog registers. Any access to these registers will reset the time back to the initialized value. Therefore, as long as the worst case time between watchdog access is less than the programmed timeout value, the module will function normally.

In addition to the hardware support for the watchdog operation, console firmware can be configured to dispatch to user code or continue with its default reset action on watchdog timeout. Firmware can detect the expiration of the watchdog during reset code by examining the hardware Reset Reason Register in the module register space of the ISbus (see Section 1.17.6 for further details). The jump to Halt code just before causing a hardware reset enables firmware to snap-shot the processor state (GPRs, and so forth) at the time of the watchdog before the full hardware reset.

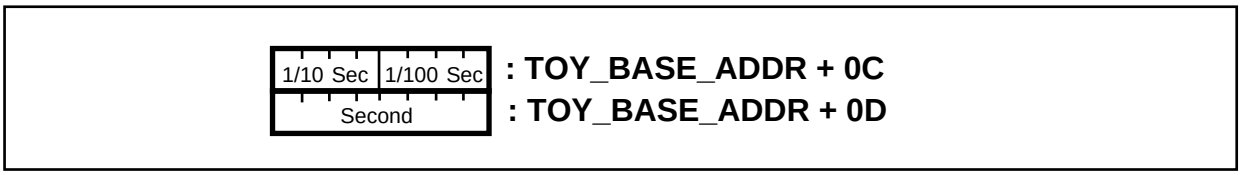
The watchdog functionality is located in the DS1386 TOY clock chip, which resides on the ISbus.

1.15.1 Watchdog Operation

Watchdog operation is controlled by four registers - three in the DS1386 chip itself and a single enable bit in the Module Control Register.

The watchdog timeout time is set in BCD in two bytewide registers in the TOY's address space, as shown in Figure 1–60.

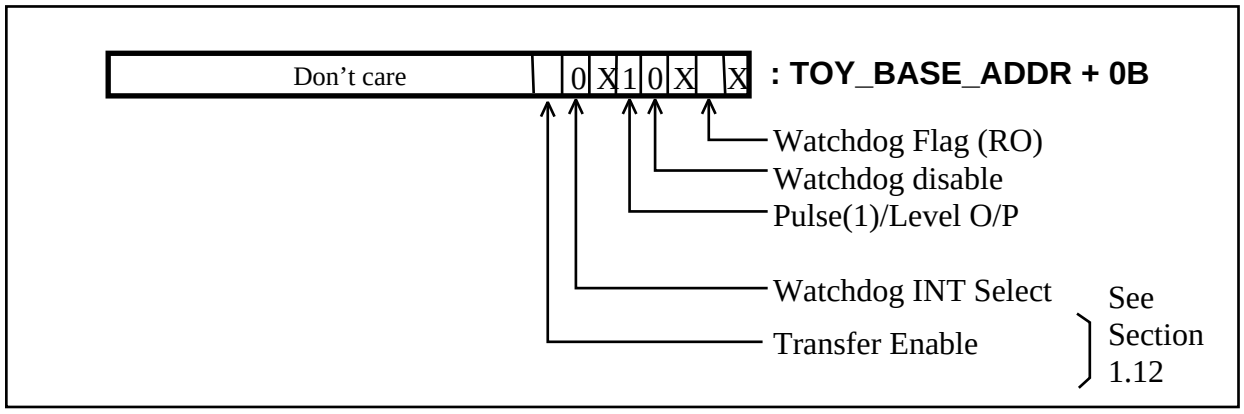
Figure 1–60 Watchdog Time Registers



Operation of the watchdog must be configured in the TOY Command Register (TOY_BASE_ADDR+0B) and enabled in the module control register (MOD_ CNTRL_REG).

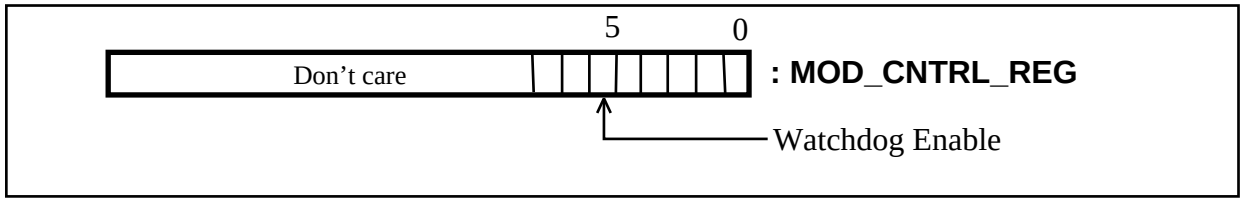
Within the TOY chip, the interrupt line and the pulse/level assertion of that interrupt line for the watchdog are selectable. The hardware design of AXPvme dictates the setting for these parameters (watchdog must use INTA in pulse mode). In addition, the watchdog function can be enabled or disabled via the TOY command byte, bit <4>. Figure 1–61 shows the required setup for operation of the watchdog in a AXPvme.

Figure 1–61 TOY Command Register (Watchdog)



Because there exists the possibility to set up the watchdog in such a way that it would constantly drive the module into reset (by setting the watchdog output to level rather than pulse, for example), an external enable, which defaults to disabled on powerup, is included. This bit is found in the Module Control Register (see Figure 1–62), which is fully described in Section 1.10.3. When the watchdog has been fully and correctly initialized, this bit should be set to allow normal watchdog operation.

Figure 1–62 Module Control Register (Watchdog)



The reset generated by the watchdog timer is "one-shot" as the Module Control Register is cleared, thus disabling the watchdog reset, when the hardware reset is asserted.

1.16 Nonvolatile RAM

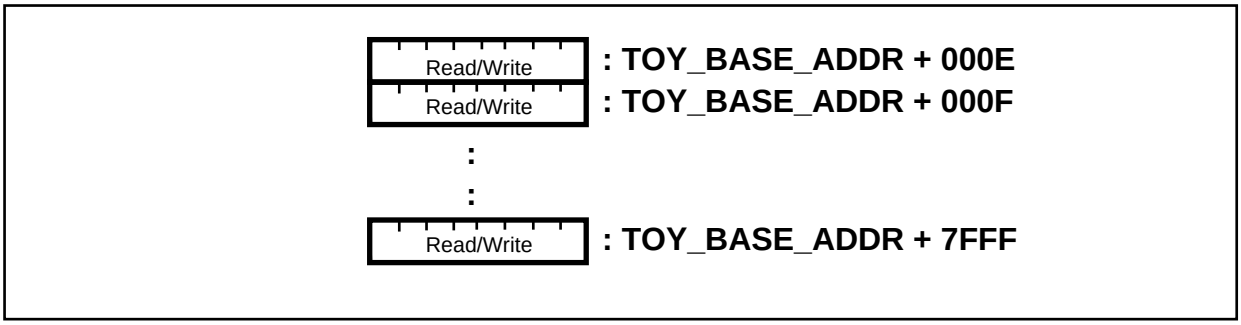
AXPvme offers just under 32 Kbytes of battery backed-up SRAM on-board. The RAM is provided by the DS1386 chip and is held nonvolatile by the built-in lithium battery source.

The memory is read/write accessible in ISbus space. In effect, the DS1386 chip (TOY, Watchdog, and NVRAM) contains 32K read/write byte elements. The lowest 14 of these bytes have special register functions for operation of the TOY

and watchdog. The remaining bytes, 32K-14, are usable as general-purpose bytewide read/write RAM.

This RAM is organized as contiguous bytes starting at TOY_BASE_ADDR+0E (see Figure 1–63).

Figure 1–63 NVRAM Access



As for the TOY clock operation, module switch 1 allows the VMEbus 5VSTDBY to be connected to the DS1386 giving RAM backup that is independent of both the normal 5 V supply and the internal lithium battery.

1.16.1 Note on Usage

The AXPvme firmware uses an amount of the NVRAM for module parameters and settings, as well as for storage of error and failure information.

For the present, the lowest 16 Kbytes of the battery backed-up RAM should be considered to be reserved for firmware usage. Thus, user and O/S code should not access NVRAM below the address of TOY_BASE_ADDR+4000h.

1.17 Interrupts and Reset

This section describes the hardware of the interrupts only. See the firmware /PALcode description for details of the interrupt handling and delivery. Some suggestions are made here with respect to the use of the interrupts but these are only suggestions for firmware/PALcode development.

There are numerous interrupt sources in the AXPvme system, however, they can be broken down into three main categories.

- Device interrupts
  - UART
  - SCSI
  - NI
  - Interval timers
  - PCI option interrupts
- VMEbus interrupts
  - Standard VME IRQ*
  - Auto-vectored
- VMEbus interface status interrupts

The complete interrupt operation of AXPvme is made up of both hardware and PALcode.

The DECchip 21066 supports three separate interrupt request inputs. If enabled (in the HICR), the assertion of any of these lines will cause a jump to PALcode. The status of these interrupt lines can be seen in PALcode via the HISR register. AXPvme uses the three interrupt inputs for:

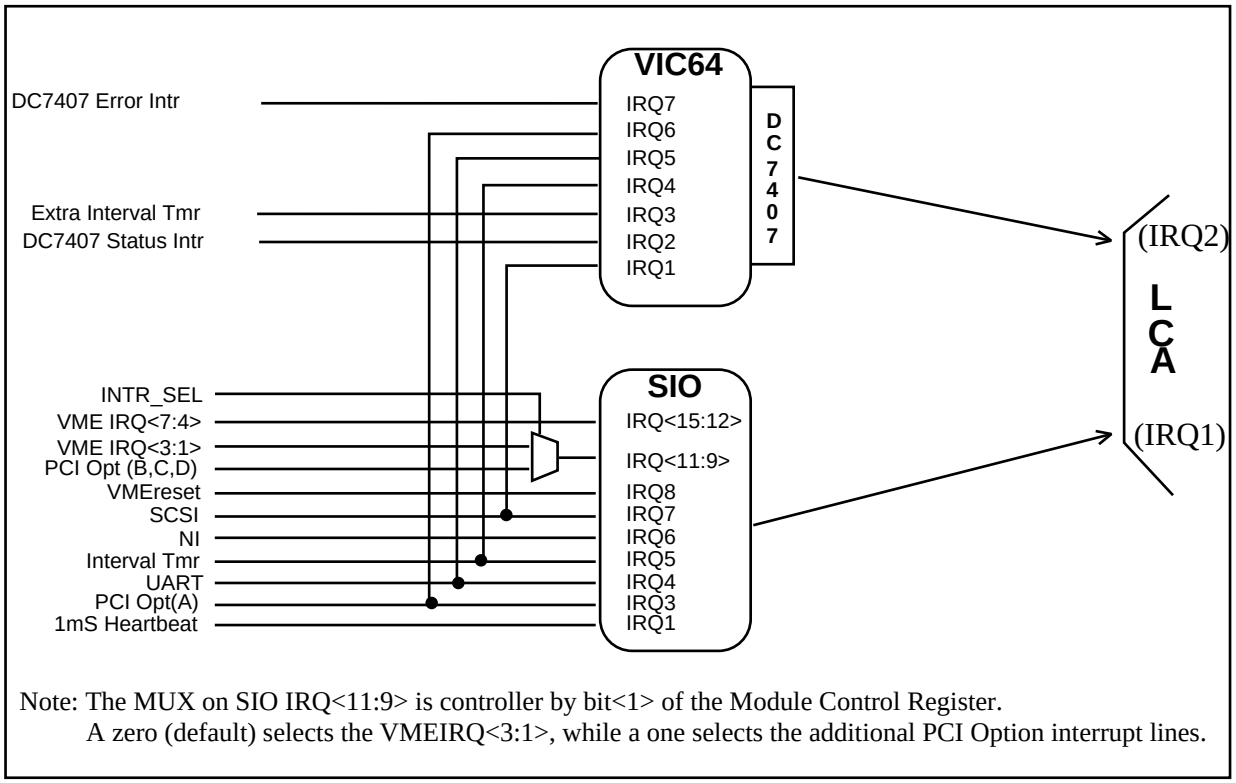
1. Nonmaskable systems events—HALT, SERR (IRQ0)
2. VIC64 system interrupt controller (IRQ2)
3. Programmable interrupt controller in SIO (IRQ1)

1.17.1 Interrupt Overview

Figure 1–64 shows a schematic overview of the interrupt structure in the AXPvme system.

Notice that most local interrupts are routed to the VIC64 (which has an internal interrupt controller) and to the 8259 PIC cores in the SIO. Each interrupt request should only be enabled in one of these controllers at any one time. However, the choice of two interrupt controllers with different characteristics within the design affords the system more interrupt flexibility.

Figure 1–64 Interrupt Overview



Note: The MUX on SIO IRQ<11:9> is controller by bit<1> of the Module Control Register.
A zero (default) selects the VMEIRQ<3:1>, while a one selects the additional PCI Option interrupt lines.

1.17.2 VIC64 System Interrupt Controller

The VIC64 system interrupt controller operation is described in the VIC64 documentation.

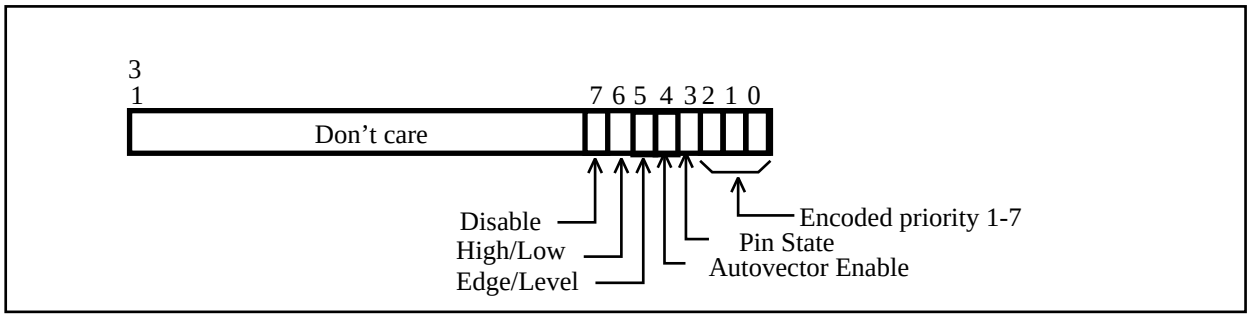
AXPvme's use of the VIC64 as an interrupt controller is modified slightly by the operation of the DC7407. The IPL lines from the VIC are passed to the DC7407 gate array, which then passes any valid interrupt request on to the LCA. The DC7407, however, offers logic to mask out interrupts at IPLs lower than a programmed level. This allows PALcode to work with the hardware to implement a more flexible, lower overhead IPL interrupt delivery scheme.

It is also important to note that vectors returned from the VIC as system interrupt controller will be "pre-pended" (using bits <10:8>) with the interrupting IPL.

1.17.2.1 Basic Operation

The VIC64 handles 19 interrupt sources. Each one of these can be individually programmed to any of the seven interrupt priority levels (IPLs) in the controller's Interrupt Control Registers (ICRs). The generic form of the ICR register is shown in Figure 1–65.

Figure 1–65 Generic Interrupt Control Register

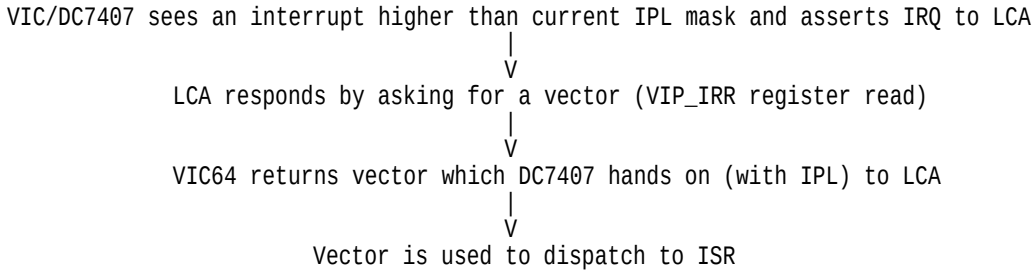


A fixed relative ranking for requests is defined. This ranking is shown in Table 1–16, and will be used to decide which interrupt is reported if many interrupts are pending.

Once there is at least one active (and enabled) interrupt request higher than the current DC7407 IPL mask, the VIC64/VIP signals the presence of a valid interrupt to the DECchip 21066 IRQ <2> interrupt pin.

The normal response to the assertion of this interrupt pin will be the initiation of a VIP_IRR register read to retrieve the vector from the VIC/DC7407. The read of the VIP_IRR generates a Local bus IACK cycle at the pins of the VIC64. When the VIC64 detects the IACK cycle, it will respond with the vector and IPL of the winning interrupter. The controller determines the highest ranking active interrupt request to be the winning interrupt. The IPL mask, within the DC7407, will update to the current IPL. The vector returned from the VIC64 and the current IPL are concatenated and returned to the processor.

A typical VIC/DC7407 interrupt flow might be:



Note that the DC7407 can be programmed to respond to a PCI IACK cycle if required (using bit VIP_CR <0>). Asking for a vector from the VIC/DC7407 when there is no valid pending interrupt will result in a passive release vector (zero) to be returned.

Table 1–16 VIC64 Interrupt Ranking

RANK	Interrupt Description	CSRs
19	DC7407 Error	VIC_LICR7, VIC_LIVBR
18	VME Interface Status/Error	VIC_EGICR, VIC_EGIVBR
17	PCI Option IRQA	VIC_LICR6, VIC_LIVBR
16	UART	VIC_LICR5, VIC_LIVBR
15	Interval Timer	VIC_LICR4, VIC_LIVBR
14	Rate Generator (Timer #1 output)	VIC_LICR3, VIC_LIVBR
13	DC7407 Status	VIC_LICR2, VIC_LIVBR
12	SCSI	VIC_LICR1, VIC_LIVBR
11	Interprocessor Comms Global Switch	VIC_ICGSICR, VIC_ICGSIVBR
10	Interprocessor Comms Module Switch	VIC_ICMSICR, VIC_ICMSIVBR
9	VMEbus IRQ7*	VIC_IRQ7ICR
8	VMEbus IRQ6*	VIC_IRQ6ICR
7	VMEbus IRQ5*	VIC_IRQ6ICR
6	VMEbus IRQ4*	VIC_IRQ6ICR
5	VMEbus IRQ3*	VIC_IRQ6ICR
4	VMEbus IRQ2*	VIC_IRQ6ICR
3	VMEbus IRQ1*	VIC_IRQ6ICR
2	DMS status	VIC_DSICR, VIC_EGIVBR
1	VME IACK	VIC_IICR, VIC_EGIVBR

1.17.3 VIC64 Interrupt Sources

The following sections describe the VIC64 interrupt sources.

1.17.3.1 Device Interrupts

There are seven external/system interrupt sources controlled by the VIC64 interrupt controller.

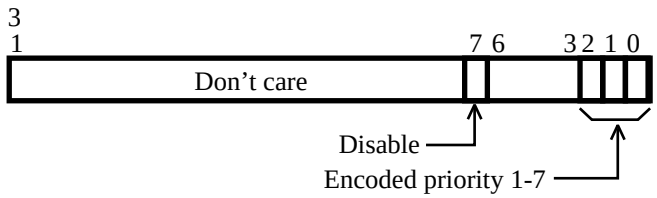
- $SCSI^{1}_{(12)}$

[1] Subscript numbers in parenthesis show the ranking values of the various interrupts.

- DC7407 Status$_{(13)}$
- Timer #1 rate generator$_{(14)}$
- Interval Timer$_{(15)}$
- UART$_{(16)}$
- PCI IRQA (option card)$_{(17)}$
- DC7407 Errors$_{(19)}$

Each of the seven interrupt sources has an associated Interrupt Control Register that allows the interrupt to be programmed with an individual IPL or to be disabled. Figure 1–66 shows these ICRs.

Figure 1–66 Device Interrupt Control Registers

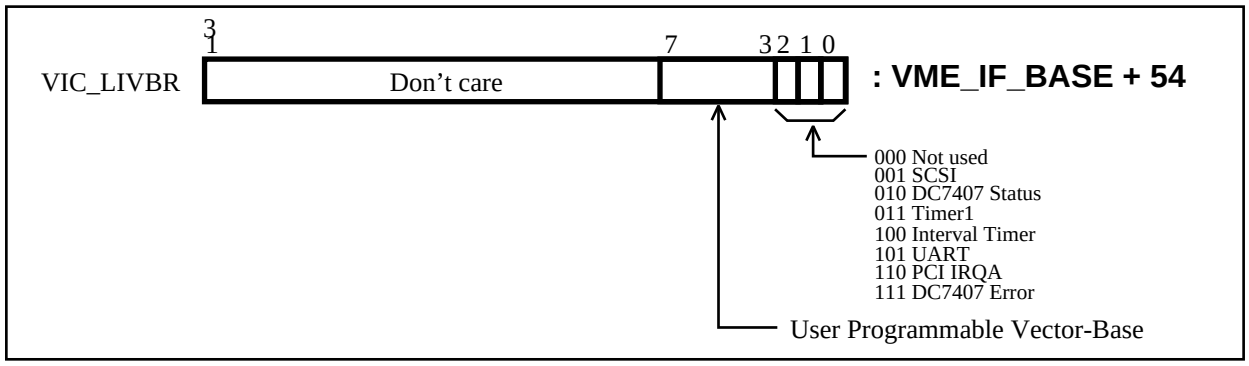


Address	Register	Device
:VME_IF_BASE + 24	VIC_LICR1	SCSI
:VME_IF_BASE + 28	VIC_LICR2	DC7407 Status
:VME_IF_BASE + 2C	VIC_LICR3	Timer#1
:VME_IF_BASE + 30	VIC_LICR4	Interval Tmr
:VME_IF_BASE + 34	VIC_LICR5	UART
:VME_IF_BASE + 38	VIC_LICR6	PCI IRQA
:VME_IF_BASE + 3c	VIC_LICR7	DC7407 Error

The vectors associated with these seven interrupt inputs have a single common root that is modified to give a unique vector for each device. Bits <7:3> of this common 8-bit vector are programmable while bits <2:0> uniquely identify the winning interrupt.

Figure 1–67 shows the local interrupt vector base register.

Figure 1–67 VIC Local Interrupt Vector Base Register

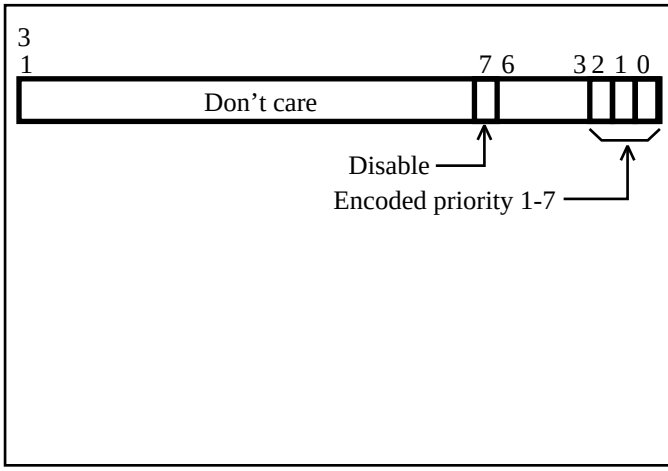


1.17.3.2 VMEbus IRQs

The VIC64, when configured as system controller, will handle the standard seven-level prioritized interrupt scheme of the VMEbus.

As for the module-based interrupt sources described above, each of the seven VMEbus IRQ's lines has its own Interrupt Control Register to allow individual disable and priority (IPL) assignment (see Figure 1–68).

Figure 1–68 VME IRQ* Interrupt Control Registers



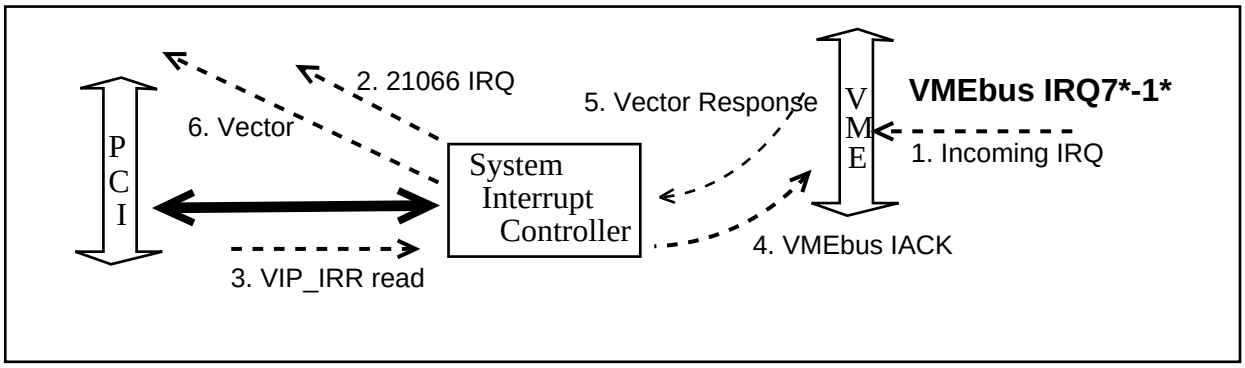
Address	Register	Line
:VME_IF_BASE +04	VIC_ICR1	IRQ1*
:VME_IF_BASE + 08	VIC_ICR2	IRQ2*
:VME_IF_BASE + 0C	VIC_ICR3	IRQ3*
:VME_IF_BASE + 10	VIC_ICR4	IRQ4*
:VME_IF_BASE + 14	VIC_ICR5	IRQ5*
:VME_IF_BASE + 18	VIC_ICR6	IRQ6*
:VME_IF_BASE + 1c	VIC_ICR7	IRQ7*

Within the AXPvme, VMEbus interrupts compete (based on IPL and ranking) with other system interrupts. If, during a Local bus IACK, a VMEbus source is deemed to be the IRQ winner, the VIC64 will initiate a VMEbus IACK cycle to retrieve the bus interrupter's vector. The VMEbus vector response is passed back to the DECchip 21066 in response to the system read of the VIP_IRR register.

Note that it is assumed that the VMEbus interrupter will release the IRQ line either on seeing the VME IACK or because of the action (register write, and so forth) of the interrupt service routines.

Figure 1–69 shows the interrupt handling flow for a standard VMEbus interrupt.

Figure 1–69 VMEbus Interrupt Handling



1.17.3.3 Status/Error Interrupts

Internal to the VIC itself there are a number of conditions and errors that can be reported via an interrupt request.

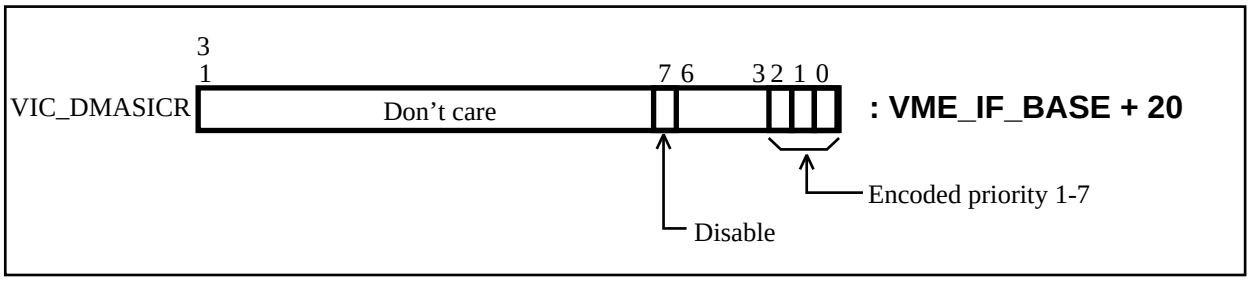
The conditions that can be enabled to cause system interrupts are:

- VMEbus SYSFAIL* assertion
- VMEbus ACFAIL* assertion
- VMEbus arbitration timeout
- VIC write post failure
- DMA completion
- VMEbus IACK cycle in response to a AXPvme generated VMEbus interrupt

These conditions are divided into three cases.

The first "case" is DMA completion. There is an Interrupt Control Register associated with this event, VIC_DMASICR (see Figure 1–70), which allows the signaling of DMA completion. If enabled, an interrupt is generated at the programmed IPL upon DMA completion.

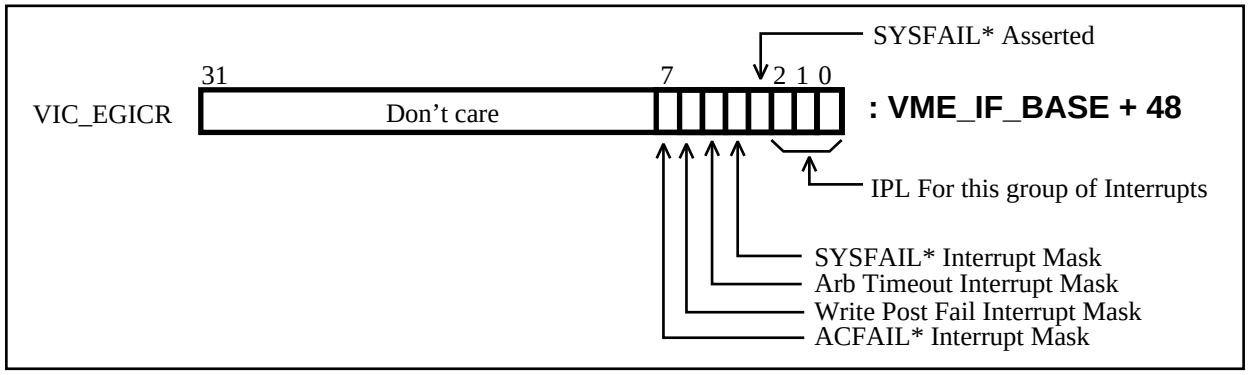
Figure 1–70 DMA Status Interrupt Control Register



The second case is a grouping that encompasses the SYSFAIL assertion, arbitration timeout, write posting failure, and ACFAIL conditions. The interrupt control register (VIC_EGICR) associated with this group (see Figure 1–71) is a little different than the ICRs already discussed. Here, a single IPL is assigned

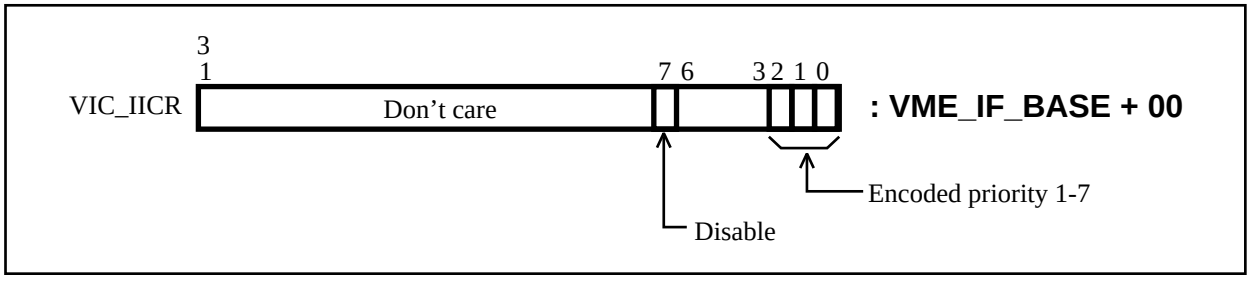
for all of the events, while the higher order register bits (<7:4>) allow individual conditions to be selectively disabled.

Figure 1–71 VIC Error Group Interrupt Control Register



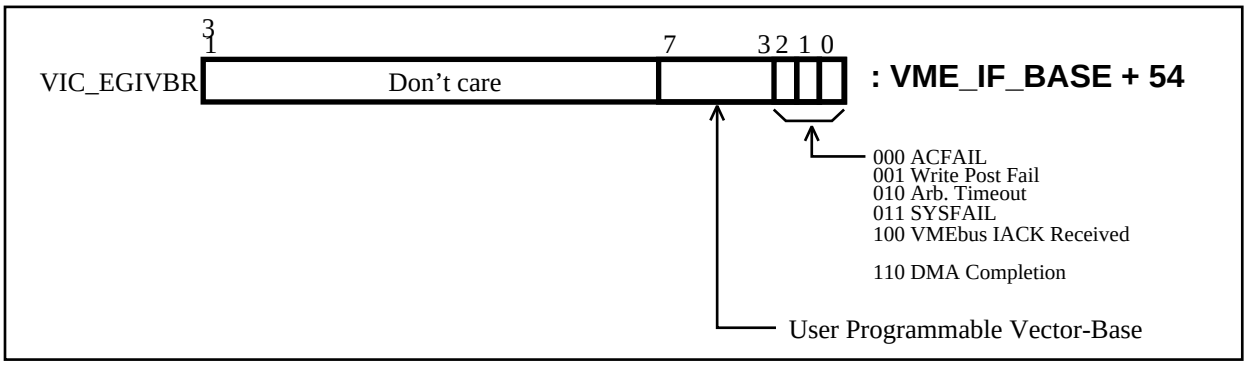
Finally, a local (on-board) interrupt is generated by the VIC64 when the VME interface sees a VMEbus IACK cycle to itself. In other words, the VIC64 can notify the CPU when the VME interface, as a VMEbus interrupter, is IACK'ed. Once again there is an associated interrupt control register, VIC_ VIICR (see Figure 1–72), to set the IPL and allow the condition to be disabled from generating its local interrupt.

Figure 1–72 VMEbus Interrupter Interrupt Control Register



There is a single interrupt vector base register for the error-group DMA and "interrupter-sees-IACK" interrupts (see Figure 1–73). In a similar way to the device interrupts outlined above, the vector root (vector bits <7:3>) is user programmable while the least significant 3 bits are different for each condition. In this way, there is a unique interrupt vector for each of these error/status events.

Figure 1–73 VIC Error Group Interrupt Vector Base Register



1.17.4 SIO Programmable Interrupt Controller

There is a second interrupt controller present on the AXPvme module. There are two main reasons for this logic:

1. To support "auto-vectored" VMEbus IRQs
2. To provide a second system interrupt controller path that is less affected by VMEbus block transfers

This Programmable Interrupt Controller (PIC) is implemented as two cascaded 8259 cores in the SIO. The resolved output of the PIC drives the DECchip 21066 IRQ <1> interrupt pin.

1.17.4.1 Auto-Vectored VMEbus IRQs

Auto-vectoring of VMEbus IRQ*s is not part of the VMEbus standard, but is an enhancement offered by AXPvme.

Auto-vectoring allows VMEbus IRQs to be used to signal events without the need for a VMEbus slave to be able to respond to an IACK cycle. The idea is to tie a local (onboard) vector to the assertion of a given IRQ line.

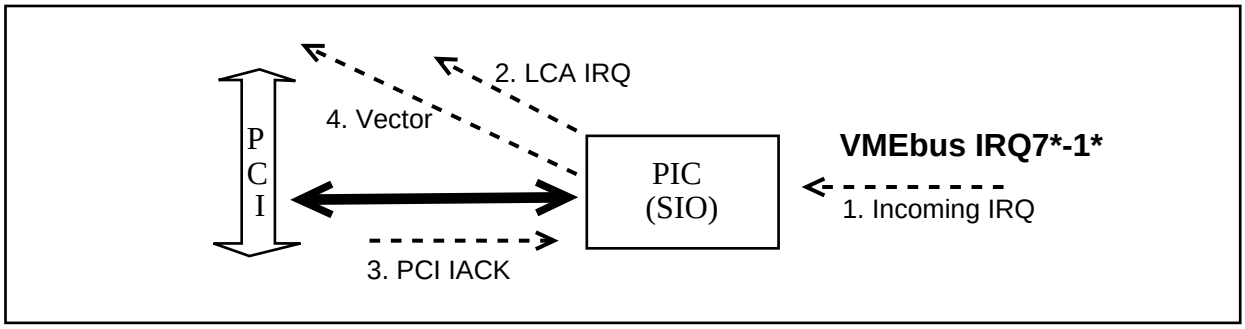
The use of a VME IRQ as a local auto-vectored interrupt precludes its use for any other system functions. Remember, no IACK cycle will be initiated on the VMEbus and there will be no use of the IACK daisy chaining.

If the IRQ is asserted, the processor will simply use a local "auto-vector" to jump to an ISR.

By programming the PIC device, selected VMEbus IRQ* lines can be used to cause a PIC interrupt to the processor. When the PIC is "IACK-read," it will return the programmed vector associated with the winning (enabled) IRQ input.

The auto-vectored flow for a VMEbus IRQ* is shown in Figure 1–74 (refer back to Figure 1–69 for comparison).

Figure 1–74 VMEbus Auto-Vector Interrupt Handling



Note that VME IRQ <7:4> are always available for use as "auto-vectored" VME interrupt requests, but VME IRQ <3:1> can only be used in this way when bit <1> of the Module Control Register is clear, which selects the VME interrupt sources instead of the additional PCI option interrupts.

1.17.4.2 PIC as Alternative System Interrupt Controller

The 82378's PIC can also be used to handle interrupt requests from important on-board interrupt sources. This allows interrupt delivery to be decoupled from the VIC64 and its Local bus, which can be tied up during long and/or slow block transfers.

1.17.4.3 Programming the SIO's 8259 Cores

For programming details of the 8259, see the SIO (Intel 82378IB) and 8259 data sheets. Also see the Digital purchase specifications.

1.17.5 Nonmaskable System Events

The final category of interrupts (the last of the three LCA interrupt pins) is the nonmaskable interrupts.

The front panel HALT button, the watchdog HALT, and a PCI SERR are the only such events.

These three system events are handled through the single remaining interrupt pin of the LCA. Thus, when an NMI hits the processor, it must have some mechanism to determine the source of the event. The two categories of AXPvme nonmaskable events (halt and SERR) are handled through the SIO, which contains a status register that can be polled to determine the NMI reason. This register is the NMI Status and Control Register at PCI I/O Address 00000061h.

All NMI events should cause a jump to the console entry point without destroying the software context, and SERR should report an error. If the interrupt reason is a HALT, the firmware should also read the Reset Reason register (PCI I/O 3000h) to see if the watchdog bit is set. If it is, then the HALT must be treated as a "save-software-context" watchdog HALT.

It should be noted that the nonmaskable description refers to the processors operation. PALcode will never mask the NMI input pin and the events will be considered highest priority. However the SIO by default disables the generation of the interrupt to the processor so they must be enabled by init code. Also, if any firmware wants to operate in "HALT-protected" space, it can do so by disabling the NMI delivery either at the HIER or SIO level.

1.17.5.1 NMI Status and Control Register

Bit <7> SERR# Status: Bit <7> is set if a system SERR has occurred. The interrupt in response to this event is enabled by clearing bit <2> of this register to a 0. Bit <7> is READ ONLY, and can only be cleared by setting the SERR Enable bit (bit <2>) to a 1 and then back to a 0. Always write this bit as a zero.

Bit <6> HALT Status: Bit <6> is set when either the watchdog expires (and is enabled) or the HALT switch is toggled. This interrupt is enabled by clearing bit <3> of this register to 0. Bit <6> is READ ONLY, and should always be written as a zero. To clear this status bit, set bit <3> and then clear it again to reenable this NMI event reporting.

Bit <5:4> = Ignore on read. Writes must be zero.

Bit <3> HALT Enable: When set to a one, HALTs are disabled and the halt status bit in this register is cleared. When cleared (reset default), HALTs are enabled as NMI events.

Bit <2> SERR Enable: When set to a one, SERR reporting is disabled and the SERR status bit in this register is cleared. When cleared (reset default), SERRs are enabled as NMI events.

Bit <1:0> = Ignore on read. Writes must be zero.

Note that when reading the SIO specification, the AXPvme HALT events are reported via the SIO's IOCHK# pin.

1.17.6 Module Reset

The AXPvme module can be reset by four distinct events:

1. Powerup
2. Front panel switch
3. Watchdog timeout
4. VMEbus SYSRESET* assertion (if enabled)

All on-board logic, except the module-level Reset Reason register, are hardware reset by all of these reset events.

The VMEbus SYSRESET* assertion will generate a module reset only if E41 switch 3 is closed. This prevents a module configured as a VME System Controller from locking itself into a reset state when it issues a VME SYSRESET* under software control.

If E41 switch 3 is open, the VIC64 will still be reset (all internal registers will return to their default state, current transactions will be aborted) but the module reset will not be generated. In order to allow detection of this condition (VIC64 only reset), the VME SYSRESET* signal is tied to SIO IRQ <8>.

1.18 Error Handling

This section describes error handling registers and procedures. AXPvme error conditions can be broken into three main areas:

- LCA chip—I/O errors, internal errors, memory controller errors
- VME interface errors
  - DC7407 errors—PCI, VIC/VME interface errors
  - VIC errors—VMEbus errors, DMA errors

- Miscellaneous system errors

1.18.1 LCA Chip Errors

See the 21066/68 Chip specification.

1.18.2 VME Interface Errors

The following sections describe the VME interface errors.

1.18.2.1 DC7407 Errors

The errors and events that are reported by the DC7407 are summarized in Vme Interface Processor Bus Error/Status Register (VIP_BESR, at VME_IF_ BASE+104h). The conditions recorded in this register can be broken into two categories.

- Status events
- Error events

Status Events

These signals are included to allow notification to the processor of accesses targeted at AXPvme, which are inappropriate for VME transactions to the module, such as write lock on page accessed. These events are indicated by setting one or more bits in VIP_BESR <17:7>.

Access Errors

When the VME interface is acting as a master on either the PCI or the VMEbus (as well as driving data on the PCI even as a target), there are a number of conditions that must be reported and handled as errors. In these cases, the appropriate bit will be set in VIP_BESR <6:0>. PCI address and the local-bus addresses associated with the transaction in error will be frozen in Vme Interface Processor error address registers (VIP_PCIERTADR, VIP_PCIERTCBE, VIP_ PCIERIADR, VIP_LERADR) as appropriate.

Typically, the PCI address will be used to work back to the S/G index used for the transfer.

1.18.2.2 VIC64 Errors

Within the VIC64 error categories, there are two subdivisions.

- VMEbus Conditions
  - VME transaction/bus conditions
  - VME special conditions (ACFAIL, SYSFAIL, and so forth)
- DMA Conditions

VMEbus Conditions

Failure or error completion of VMEbus transactions are flagged by the VIC64 by assertion of BERR* on the VME and simultaneously LBERR on the local bus (DC7407<->VIC64). This LBERR assertion will cause the DC7407 to detect the error condition, which will cause a DC7407 status interrupt (if enabled) and a PCI target abort. In this case, looking at the VME Interface Processor Bus Error Status register (VIP_BESR) and the VIC64 Bus Error Status register (VIC_ BESR) should give a picture of the error condition.

The special VMEbus conditions like the assertion of ACFAIL, SYSFAIL, and so forth, are covered by the VIC Error Group Interrupt facility (VIC_EGICR, VIC_ EGIVBR).

DMA Conditions

Completion of a DMA operation is flagged by an interrupt (see VIC_DMASICR, and so forth). Looking at the DMA status register (VIC_DMASR) in the VIC64 will indicate the nature of any errors encountered.

1.19 Environment

1.19.1 Operating Conditions

Temperature:	0°C to 50°C with 200 lfm of air flow over the CPU heatsink
Humidity:	70% relative humidity
Altitude:	8000 feet above sea level

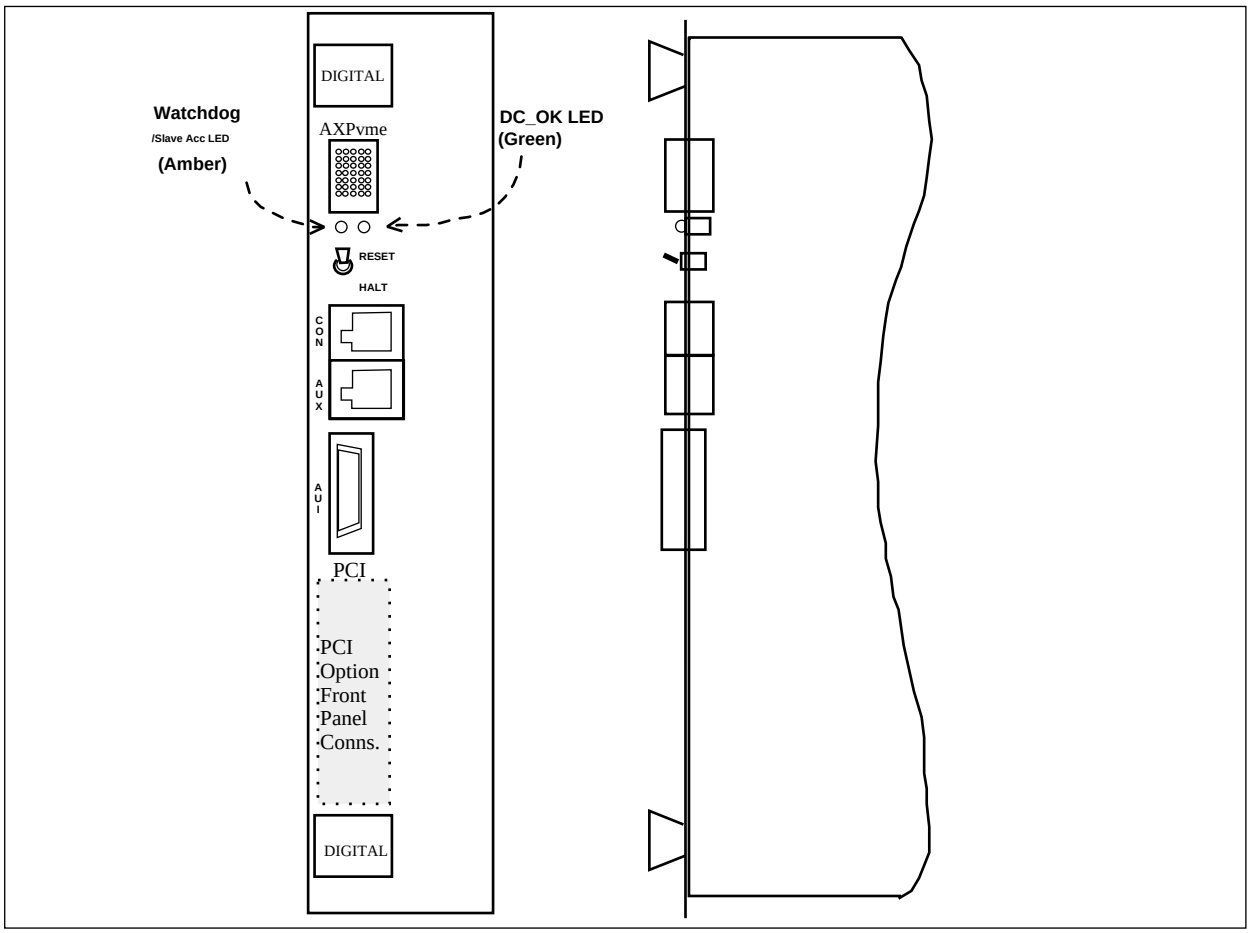
1.19.2 Storage Conditions

Temperature:	-40°C to 66°C
Humidity:	10% to 95% (noncondensing)

1.20 Physical Specification

The AXPvme Single-slot is a single 6U VME card with P1 and P2 connectors and a standard VME card handle/front panel. AXPvme dual-slot will occupy two 6U VME slots (see Figure 1–75).

Figure 1–75 Module and Front Panel



The front panel has two MMJ connectors and an AUI connector, as well as a status display, a reset and halt switch, and two LED indicators.

The module is double-side populated, but with passives only on side 2.

Two daughtercard buses are provided. One is for main memory and the other is for PCI options.

1.21 AXPvme Breakout Modules

The AXPvme family of products have “companion” modules that enhance the functionality, allow increased CPU speed, interconnect to SCSI, and provide interconnect to various AXPvme specific signals. These modules are called “breakout” modules.

There are two versions of the breakout module. The single-slot version is required for use with the single-slot modules. The double-slot version is required for use with the dual-slot modules.

Caution

In order to function properly, each AXPvme module installed MUST have its corresponding breakout module installed properly. AXPvme has made numerous connections to the VME user-defined pins. Installing a “non-AXPvme” VME module in a slot that has a AXPvme breakout module installed may DAMAGE the VME module, the backplane, or both.

1.21.1 Breakout Module Functionality

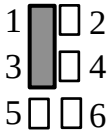
The AXPvme breakout modules provide the following functions:

- Connection to SCSI signals
- SCSI termination and control
- External reset pin
- Connection to additional power and ground pins
- Connection to AXPvme specific external timing signals
- Connection and control of watchdog timeout signal
- Connection to manufacturing test port
- External VME Master Selector Pin (AXPvme 66, AXPvme 100, and AXPvme 231 modules only)

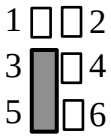
There are two functions controlled by jumpers on the AXPvme breakout modules. These are SCSI Termination and Watch_Dog timeout.

SCSI Termination Jumper

Active SCSI termination is provided on all versions of the AXPvme breakout modules. If the AXPvme module is configured at the end of a SCSI cable, the terminators should be enabled. SCSI termination is enabled by placing a jumper between pins 1 and 3. If the AXPvme module is not connected at the end of the SCSI cable, SCSI termination should be disabled by placing a jumper between pins 5 and 3.



SCSI Termination
Enabled (default)

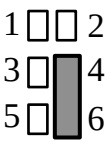


SCSI Termination
Disabled

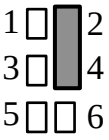
Watch_Dog Timeout Jumper

AXPvme indicates the status of the onboard Watch_Dog timer on a signal called WD_STATUS_OC H. This signal is driven low whenever an onboard watchdog timer is allowed to expire. The device driving this signal is a 74LS05 open-collector inverter. It is capable of sinking a (IoL) 8 mA maximum.

This signal may be pulled up to the 5 V rail (by a 2K ohm resistor) by placing a jumper between pins 4 and 6. Placing the jumper between pins 2 and 4 disconnects the 2K ohm resistor from the 5 V rail.



Wd_Status_Oc H
2k Ohm Pull-up connected to +5V
(default)



Wd_Status_Oc H
Pull-up Disconnected

1.21.2 AXPvme Single-Slot Breakout Module

The AXPvme 64, AXPvme 64LC, AXPvme 66, and AXPvme 100 are single 6U slot versions of the AXPvme product. They all use the same version of the single-slot breakout module. Figure 1–76 diagrams the proper way to install the AXPvme single-slot breakout module. It must be installed in the same slot as the AXPvme single-slot but on the opposite side of the P2 connector.

Figure 1–76 AXPvme Single-Slot Breakout Module Installation

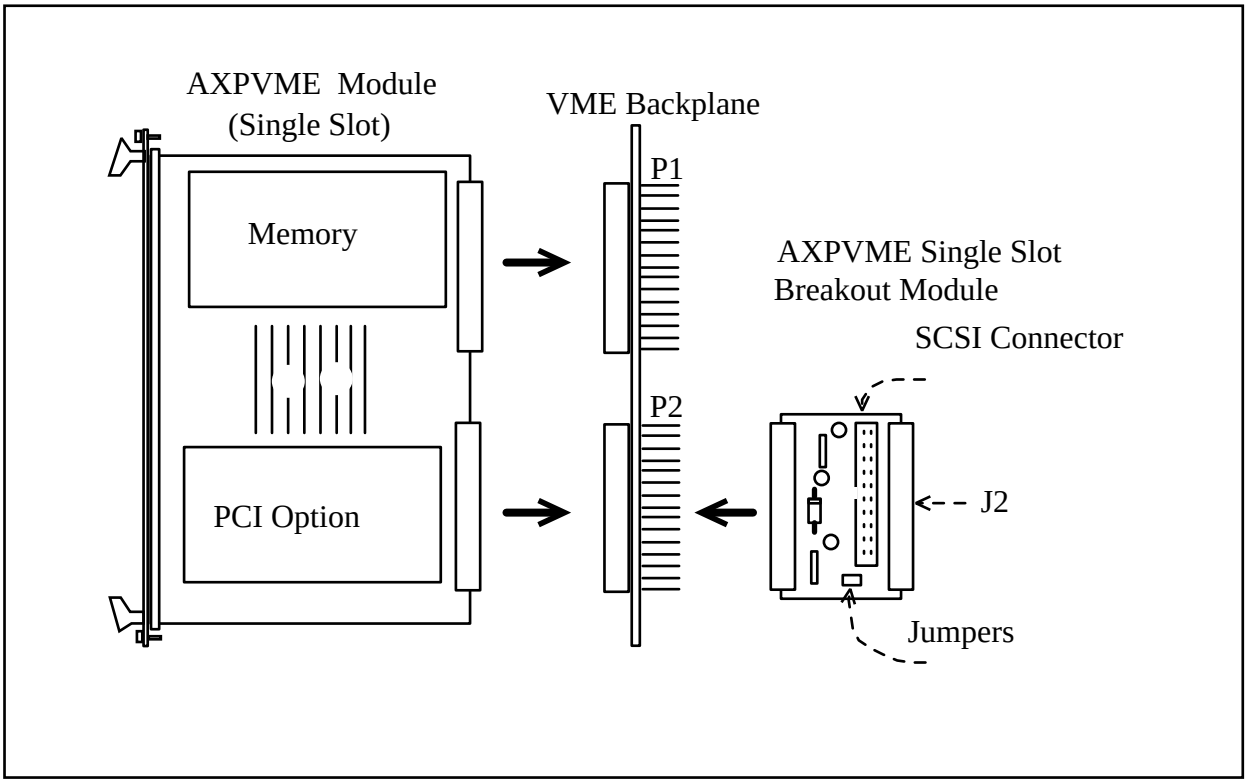
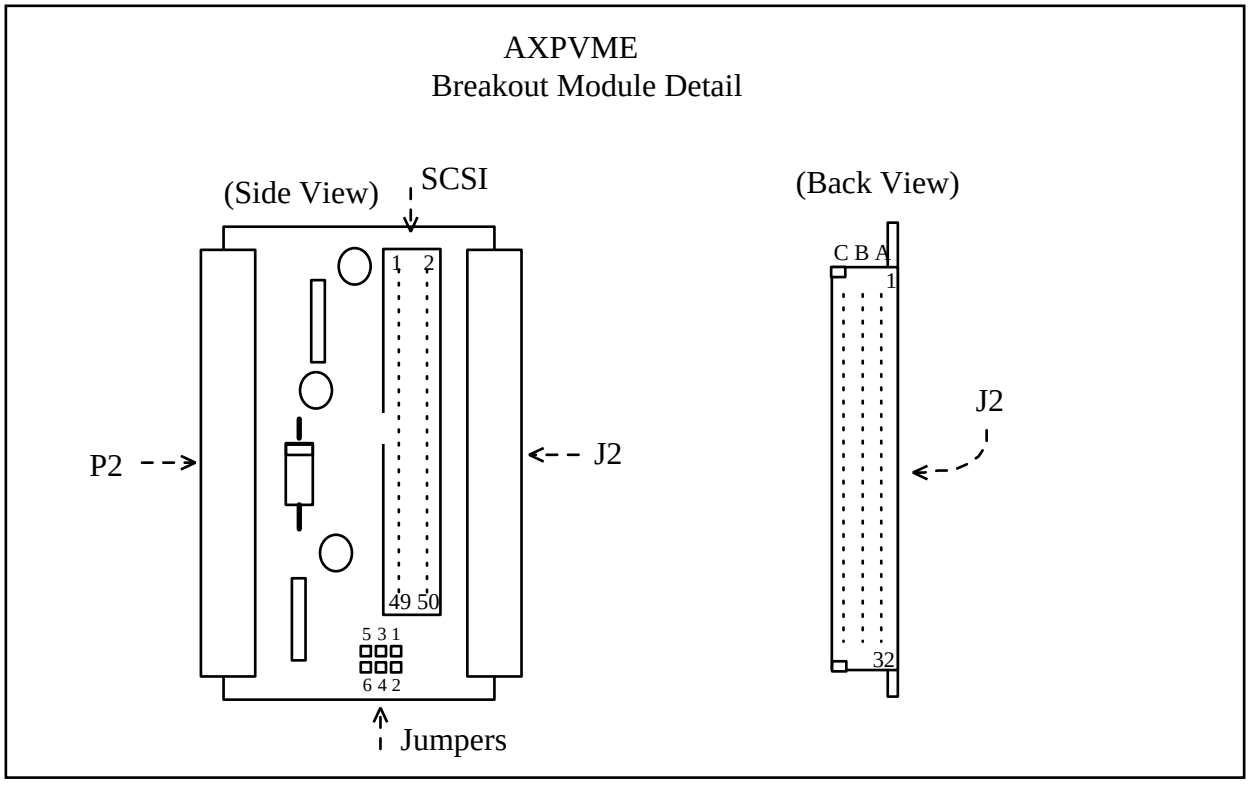


Figure 1–77 details the location of the AXPvme single-slot breakout module connectors and jumpers. The female connector (P2) plugs onto the back of the VME backplane opposite the CPU module. The 50-pin SCSI connector brings out SCSI signals to an industry standard form factor. The male connector (J2) makes the remaining signals accessible for interconnect by the customer and/or manufacturing. Jumpers are used to select SCSI termination scheme and watchdog signal termination.

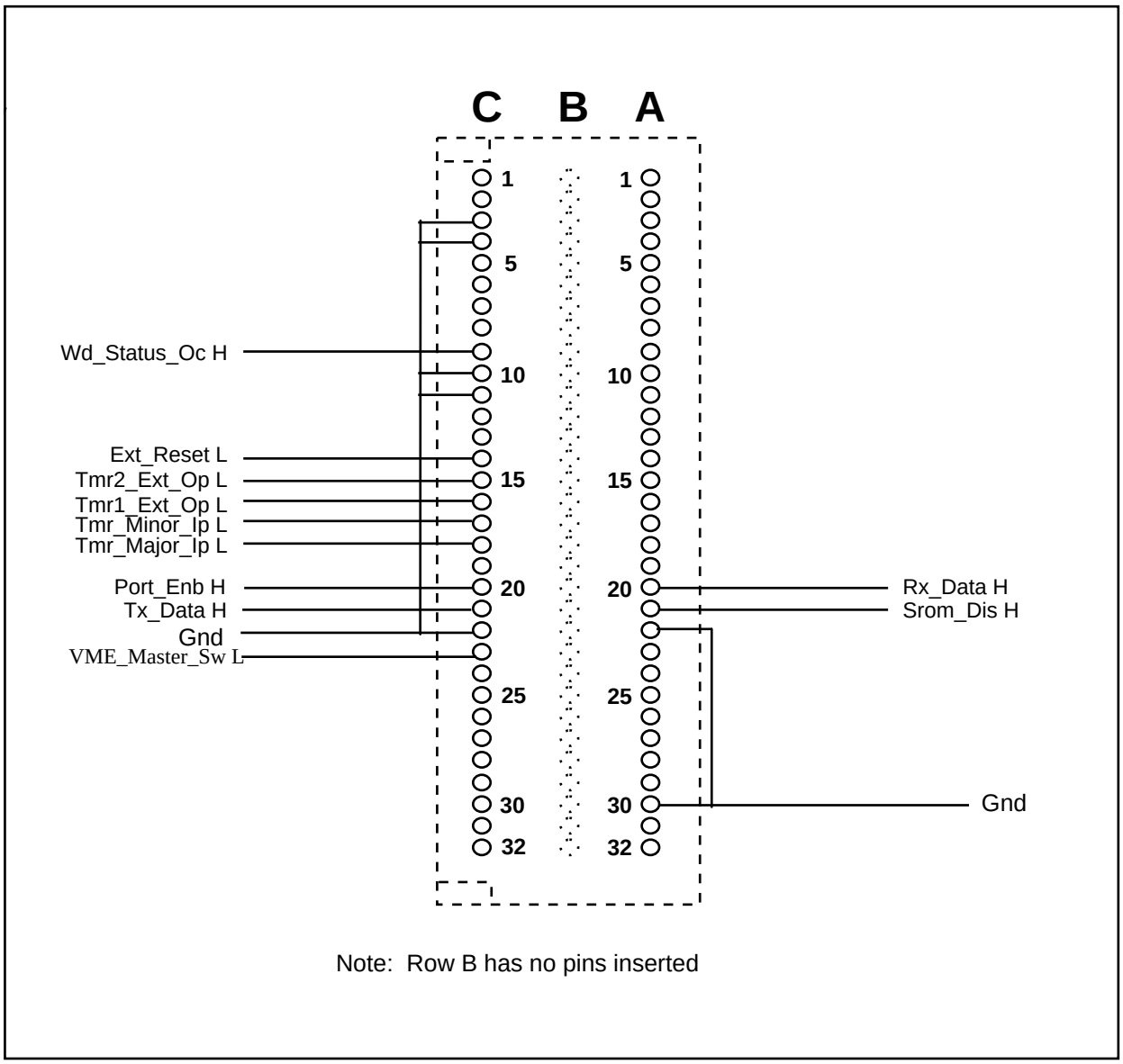
Figure 1–77 AXPvme Single-Slot Breakout Module Detail



1.21.3 AXPvme Single-Slot Breakout J2 Connector

Figure 1–78 details the pinning for rows A and C of the AXPvme single-slot breakout J2 connector. Rows A and C make several AXPvme specific signals available for external interconnect. Row B pin locations are unpopulated.

Figure 1–78 AXPvme Single-Slot Breakout J2 Connector Pinout



1.21.4 AXPvme Dual-Slot Breakout Module

The AXPvme dual-slot version of the product has increased power requirements (over the AXPvme single-slot product), therefore a dual-slot version of the breakout module is needed. Figure 1–79 shows the proper way to install the AXPvme breakout module. It must be installed such that the female connector whose tails form J2 is immediately opposite the CPU connector (in the same slot as the dual-slot AXPvme but on the opposite side of the backplane). The female connector, without the long tails, should line up and plug in to the second slot occupied by the module.

Figure 1–79 AXPvme Dual-Slot Breakout Module Installation

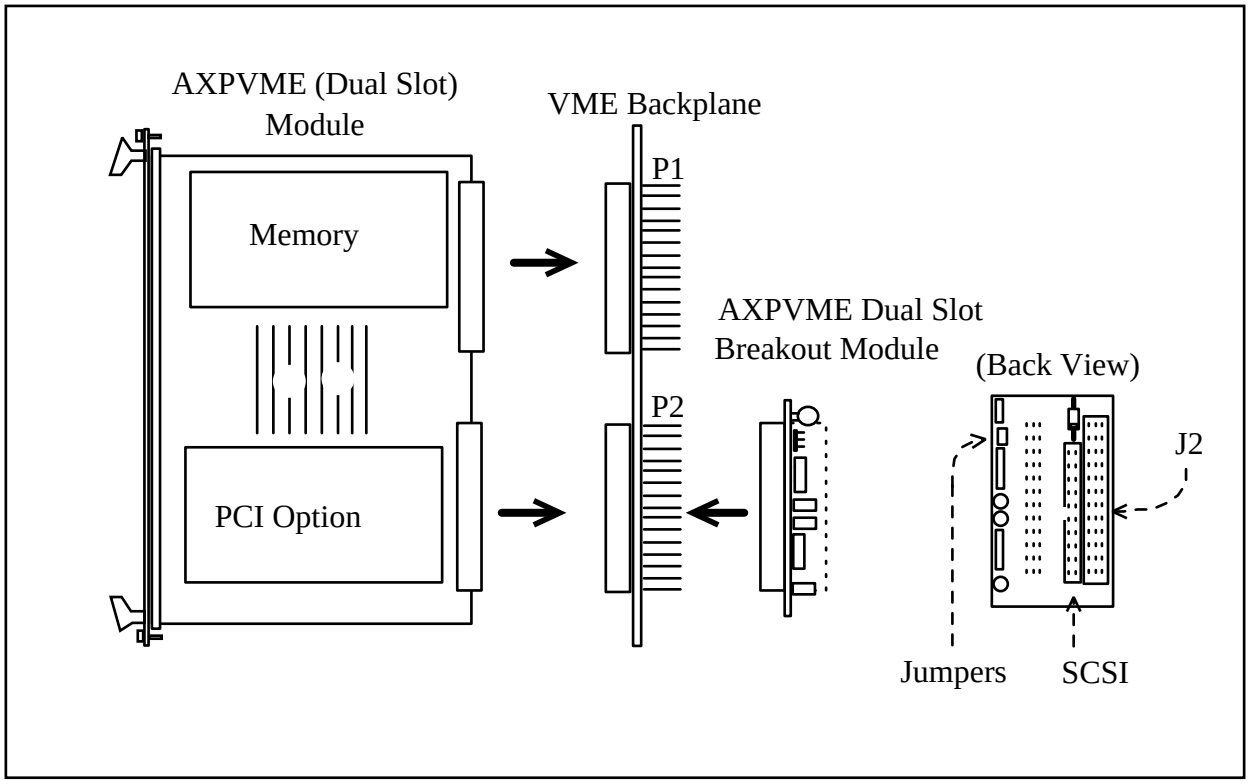
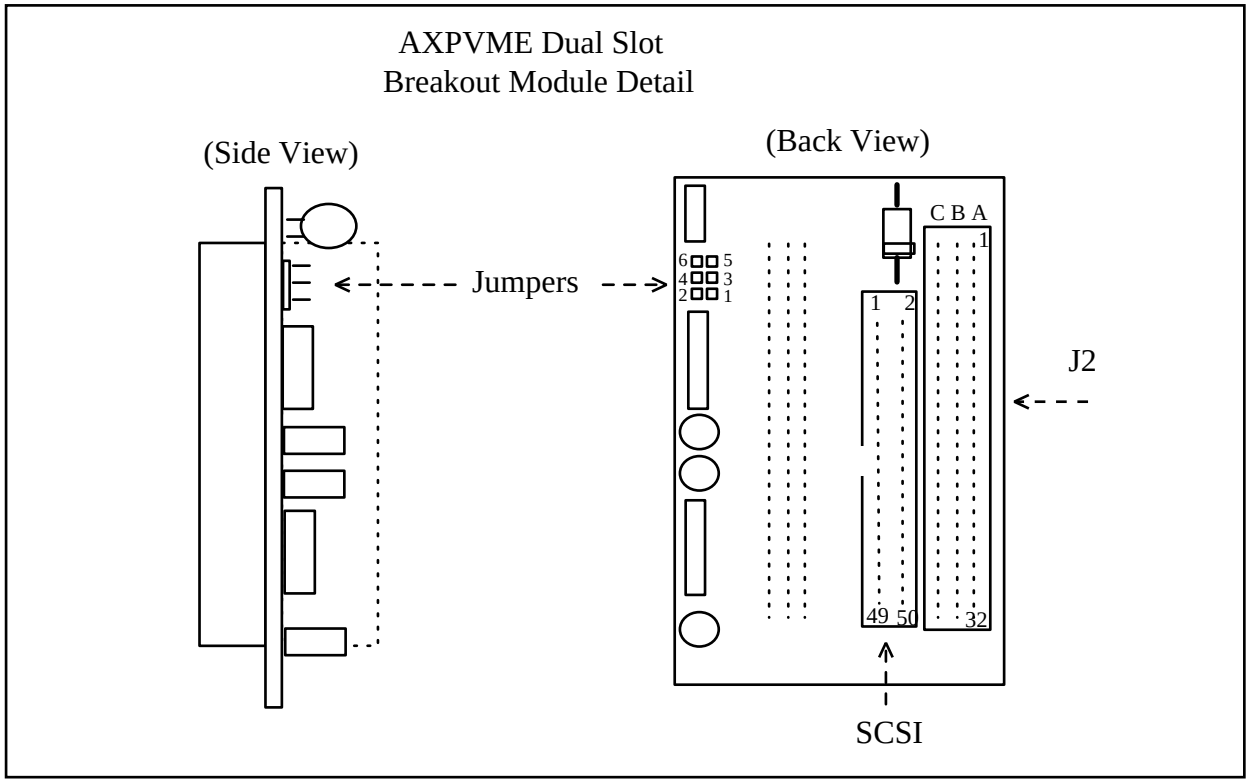


Figure 1–80 shows the location of the AXPvme dual-slot breakout module connectors and jumpers. The long-tailed female connector (P2/J2) plugs onto the back of the VME backplane opposite the CPU module. The 50-pin SCSI connector brings out SCSI signals to an industry standard form factor. The male connector (J2) makes the remaining signals accessible for interconnect by the customer and/or manufacturing. The jumpers are used to select SCSI termination scheme and watchdog signal termination.

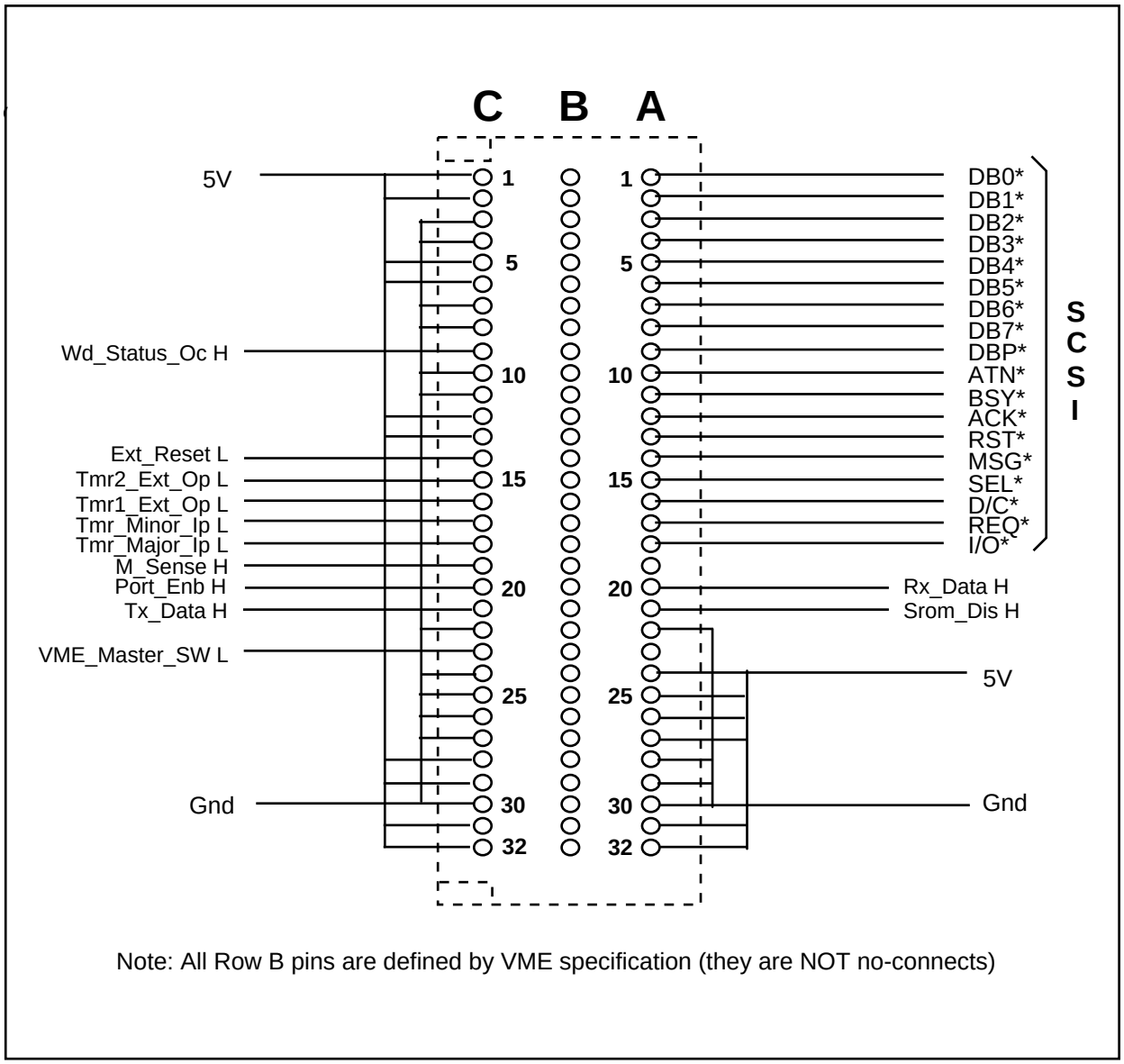
Figure 1–80 AXPvme Dual-Slot Breakout Detail



1.21.5 AXPvme Dual-Slot Breakout J2 Connector

Figure 1–81 shows the pinning for rows A and C of the AXPvme dual-slot J2 connector. Rows A and C make several AXPvme specific signals available for external interconnect. Row B pin locations are populated and connected to VME signals as defined in the VME specification. No connections should be made to row B pins.

Figure 1–81 AXPvme Dual-Slot Breakout J2 Connector Pinout



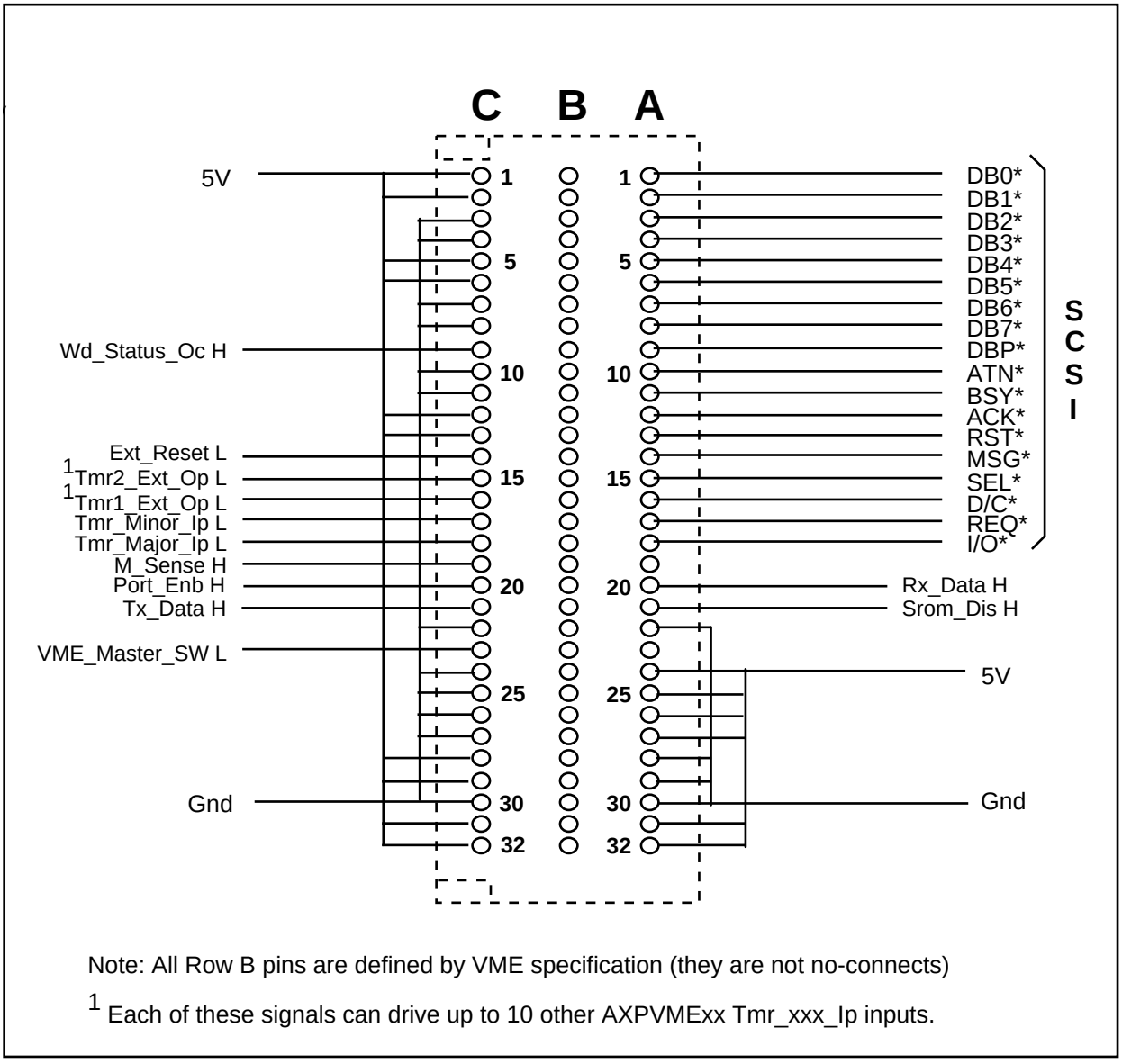
Note: All Row B pins are defined by VME specification (they are NOT no-connects)

1.21.6 AXPvme P2 Connector Usage

Figure 1–82 gives the pinning for rows A and C of the AXPvme P2 connector. Rows A and C are specified as "user-defined" pins by the VME specification. AXPvme has used these pins to interconnect AXPvme specific signals and to connect additional power/ground. All row B pins are defined by the VME specification and are not included in this diagram.

Standard 96-pin VME backplane connectors are of the type known as "press pin" connectors. These connectors have long pins that protrude through the back of the backplane. Placing a protective shroud around these pins form male connectors (P1 and P2). The P2 connector has many of the 96 pins left as "user-defined" by the VME specification. Basically, rows A and C are "user-defined" and row B is completely used/reserved by the VME.

Figure 1–82 AXPvme P2 Connector Pinout, Dual-Slot and Single-Slot



1.22 Power and Environmental Requirements

The following topics provide the power and environmental requirements for the AXPvme module.

AXPvme Physical and Environmental Requirements

The AXPvme module requires a VME chassis with sufficient cooling. It requires at least 200 linear feet/minute (lfm) of airflow at an ambient temperature of not more than 50°C (122°F) across the processor (E38) heatsink at the center of the module.

Table 1–17 shows the physical and environmental specifications for the AXPvme module. Stresses beyond those specified may cause permanent damage to the module.

Table 1–17 Physical and Environmental Specifications

Characteristic	Specification
Industry Standard	VME 6U module
Operating temperature	0°C to 50°C (32°F to 122°F)
Storage temperature	-40°C to 66°C (-40°F to 151°F)
Temperature change	20°C/hour (36°F/hour)
Relative humidity	10% to 95% (noncondensing)
Airflow	A minimum of 200 linear feet/minute over the large processor heatsink at the center of the AXPvme module

Table 1–18 shows the heat dissipation at specific processor frequencies.

Table 1–18 Power and Heat Dissipation at Processor Frequencies

Processor Clocks	Maximum Heat Dissipation	Power Amps @ $+5V$, 5% Regulation
230 MHz	48 W	9.6 A
160 MHz	38 W	7.6 A
100 MHz	30 W	6.0 A
66 MHz	25 W	5.0 A

2
Console Primer

2.1 Introduction

This chapter highlights some of the AXPvme console features and describes the use of some basic commands for performing console tasks.

The AXPvme console achieves much of its power and flexibility because it supports traditional U*X functionality, in that it looks and feels like U*X. If you have a good working knowledge of OpenVMS, this primer will help you make a smooth transition from OpenVMS to U*X. Unless you are already familiar with U*X, read this chapter and practice some of the examples before attempting a terminal session. This chapter describes the fundamental features of the console and highlights the use of commands compared to OpenVMS. Reading this chapter will give you an understanding of the basic functions of the U*X-like kernel, various utilities and tools, the user interface, and how these compare with the structure of OpenVMS.

2.1.1 Console Features

The AXPvme console firmware provides common services and functionality: operator interface, operating system bootstrap, operating system restarts, self-test diagnostics, and extended functional diagnostics. Although it may take a while to get used to (unless you are already familiar with U*X), you will discover that it is an extremely powerful, yet simple environment. This paradigm was chosen for its simplicity: the console provides a platform of simple tools easily combined to solve complex problems.

The AXPvme console is much more flexible than previous consoles because it supports traditional U*X functions such as pipes, I/O redirection, command-level scripting, and control functions. Built around a multitasking kernel, the console provides an excellent environment for support of much more complex functions, such as systems exercisers, MOP listener, and remote console.

Three separate, traditional VAX firmware components—console, diagnostics, and virtual monitor boot (VMB)—have been integrated in the firmware to avoid redundant functionality. All components of the firmware use the same kernel services and I/O drivers. For example, you use the same drivers when performing diagnostics as when you perform bootstrap or normal console operations.

2.1.2 Command Overview

The AXPvme console is a hybrid of a VAX console and U*X shell. Although the command line parser expects U*X style commands, many commands are very similar to VAX console commands. Some commands are unique to this console. Table 2–1 lists frequently used commands from each of these groups. Chapter 3 is an alphabetized command reference section containing complete descriptions of the each of the console commands.

By cloning certain U*X functions and carrying along some VAX console functions, Digital has taken advantage of existing paradigms rather than reinventing or renaming similar functions. Due to popular demand, however, the prompt has remained the same as VAX consoles, the triple angle prompt, ">>>".

Instead of VAX like **/qualifiers**, the AXPvme console uses U*X like **-options**. For example, a VAX console command, such as **e/b 0**, must be typed **e -b 0**. Note that you must use a space to separate the option from the command. If you type **e-b 0**, the console issues an error message.

Table 2–1 Frequently Used Commands

VAX like Commands	U*X like Commands	Unique Commands
boot	cat	edit
examine	echo	exer
deposit	eval	memexer
help	grep	memtest
set	hd	nettest
show	ls	sa
test	man	
	ps	
	sleep	

2.1.3 Console Shell Operators

The AXPvme console is similar to a U*X *shell*. A *shell* is a command line interpreter, an interface between the operator and the firmware. The lexical analyzer and parser for the AXPvme console implement a subset of the Bourne shell with some minor modifications.

An integral part of the console is its set of *shell* operators. These operators qualify the operation of commands, permit redirection of I/O, and allow for sequencing of operations. These operators are described in Table 2–2.

Table 2–2 Console Shell Operators

Operator	Name	Form	Description
>	Output creation	>destination	Write output to destination.
>>	Output append	>>destination	Append output to destination.
<	Input redirection	<source	Read input from source.
<<	Here document	<<string...	Read input from standard input until string is seen at the beginning of a line.
|	Pipe	cmd1 | cmd2	Pipe output of first command to input of second command.
;	Sequence	cmd1 ; cmd2	Run first command to completion before running second command.
\	Line continuation	cmd1 \ _> cmd2	Continue command on the next line. The command line prompt changes to "_>" until the command is completed.
#	Line comment	# text	The text following the number sign is ignored. This is useful for imbedding comments in command scripts or logs.
&	Background	cmd &	Run command in background, do not wait for command to complete.
&a	Affinity	&a m	Sets the processor affinity mask to allow this process to run on the CPUs defined by mask m. Multiple processors may also be specified as a list or range.
(),{}	Grouping		Used to override precedence of pipe, sequence, and background operators.
*,?,[...]	Pattern specifiers		Characters used to form a regular expression for pattern matching where "*" matches any character or characters or none, "?" matches any single character, and "[...]" matches any of the enclosed characters.
$string	Environment variable substitution		The string is treated as a legal environment variable and translated.
"xxx"	String with no substitution		The string is passed untouched.
"String"	String with substitution		The string is passed after wildcards and environment variables are expanded.
"cmd"	Command substitution		Treat the string as a command string, execute it, and substitute in the resulting output.

In addition to shell operators, the console uses the following reserved words:
if, then, else, elif, fi, case, in, esac, for, while, until, do, and done.

2.2 Getting Information About the System

The following commands may be used to provide information about resident software and hardware in the system:

```
>>>show version
version                 V12.0-0 Oct 26 1994 12:58:38
>>>show pal
pal                     VMS PALcode X5.48-80, OSF PALcode X1.35-53
>>>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
```

2.3 Online Help

The AXPvme console also provides online **help**. In ROM-based images of the console, *brief* help is supplied for each command. The *brief* help displays a one-line description and the syntax for the command. The syntax line lists all possible options and arguments for the command. For instance, in the following example, the command requests help on **help**:

```
>>>help help
NAME
        help or man
FUNCTION
        Display information about console commands.
SYNOPSIS
        help or man [<command>...]
                    Command synopsis conventions:
                    <item> Implies a placeholder for user specified item.
                    <item>... Implies an item or list of items.
                    [] Implies optional keyword or item.
                    {a,b,c} Implies any one of a, b, c.
                    {a|b|c} Implies any combination of a, b, c. )

>>>
```

The console also supports the U*X alias **man** for online help. If you do not specify a help topic when invoking the **help** or **man** command, a complete list of commands is displayed in addition to the help **help** brief text.

```
>>>help
NAME
        help or man
FUNCTION
        Display information about console commands.
SYNOPSIS
        help or man [<command>...]
                    Command synopsis conventions:
                    <item> Implies a placeholder for user specified item.
                    <item>... Implies an item or list of items.
                    [] Implies optional keyword or item.
                    {a,b,c} Implies any one of a, b, c.
                    {a|b|c} Implies any combination of a, b, c. )

The following help topics are available:
alloc           bcache_diag     boot            bpt             break
cat             chmod           chown           clear           clear_log
continue        crc             date            deposit         display_diag
ds1386_diag     dynamic         echo            edit            enet_diag
eval            examine         exer            exit            false
flash_diag      free            grep            hardware        hbeat_diag
hd              help or man     i8254_diag      init            init_ev
kill            line            ls              mblt_diag       mememcc_diag
memexer         memtest or mem_ modcnfg_diag    modctrl_diag    ncr810_diag
net             nettest         nicsr_diag      niil_diag       ps
pwrup           rm              sa              sblt_diag       semaphore
set             set led         set mode        set reboot srom set toy sleep
shell           show            show config     show device     show hwrpb
show led        show map        show mode       show_iobq       show_log
sleep           sort            sp              start           stop
true            update          vip_diag        vme_rundown     vme_setup
vmeslave_diag   wdog_diag       z8530_diag
>>>
```

You can specify multiple topics with the **help** command, as shown below. Type a space between topics to keep them separate:

```
>>>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>

>>>
```

The **help** command also supports a type of wildcarding. In the following example, help on any command beginning with the letters **da** will be displayed:

```
>>>help da
NAME
        date
FUNCTION
        Set or display the current time and date.
SYNOPSIS
        date [<yymmddhhmm.ss>]

>>>
```

To display help for all commands, type **help ***.

```
>>>help *
```

Full help is available in loadable versions of the console. With full help, all the information provided in Chapter 3 is also available in the console. However, due to space restrictions in the firmware ROMs, only brief help is available by default.

2.4 Examining and Depositing to Memory or System Registers

In the AXPvme console, many commands act on byte streams. A byte stream is similar in concept to a VAX console address space and may represent an extent of memory, a set of registers, a device, or a file. The console manipulates these byte streams by performing typical device operations—open, read, write, close. Therefore, throughout this document, the term *device* will be used to refer to any such byte stream or address space regardless of its actual physical implementation. Therefore, a traditional VAX address space, /P, can be accessed as a *device*, PMEM.

Hence, the **examine** and **deposit** commands manipulate *devices* when accessing data within the system. The default *device* is physical memory, which sticks (all subsequent implicit references access that *device*) until explicitly changed. When another *device* is specified, that *device* becomes the default.

Internally, the console uses *drivers* as the access mechanism for referencing different *devices*. Specifically, the console provides *drivers* for the following Alpha *devices*:

pmem: —physical memory
vmem: —virtual memory
gpr: —general-purpose registers
fpr: —floating-point registers
ipr: —internal processor registers

In this paradigm, the address argument of a VAX console command becomes a byte *offset* within a *device* in a AXPvme console command. For example, **pmem:0** explicitly refers to the location in physical memory at offset zero, that is, physical address 0. If no *device* name is supplied, the *offset* implicitly applies to the last *device* referenced (**pmem** by default). In the remaining discussions, however, the terms *address* and *offset* will be used synonymously.

In the console, there is also the notion of a last referenced address. An **examine** or **deposit** command without an explicit address will always reference the next address (computed as the last referenced address plus the current data size). The characters **+**, *****, and **-** are symbolic addresses for the next, current, and previous addresses, respectively.

Data width options are analogous to the corresponding VAX qualifiers. That is, **-b, -w, -l, -q,...** options correspond to the size of the accessed data—byte, word, longword, quadword, and so forth.

2.4.1 Accessing Memory

Before randomly experimenting with memory, it is important to find a "safe" area in memory to alter. Since the console itself and other critical data structures reside in memory, care should be taken not to alter them. The **alloc** command may be used to allocate a 1000-byte block of memory, as shown in the following example:

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

The **alloc** command returns the address of the allocated block, in this case, 03FFF000. Hence, in the following examples, this block will be used for experimentation.

The next example shows the **examine** and **deposit** commands to the **pmem:** device (physical memory) at the allocated address.

```
>>>deposit pmem:3fff000 1                # Deposit a 1 at address 3fff000.
>>>examine pmem:3fff000                  # Examine the location.
pmem:          3FFF000 00000001
>>>
```

The next example shows the use of the abbreviated form of the commands, **e** and **d**. Abbreviations for commands are permitted and typically, as in this case, the device specifier is absent. Assuming the state left by the previous example, the current device is still physical memory (or **pmem:**).

```
>>>d 3fff000 abcdef12                    # Deposit new data there.
>>>e 3fff000                             # Check it out.
pmem:          3FFF000 ABCDEF12
```

Below is an example using a console command option.

Note

Remember, the console uses U*X-like **-options**, not VAX-like **/qualifiers**. Also notice the inconspicuous use of white space.

In this example, the **-n** option is used to specify a repeat count. Each 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
>>>
```

The console provides a *hex dump* command, **hd**, as an alternate method for dumping memory (or other devices or files). Here, the **-l** option specifies the number of bytes to display. (Why **-l** for **hd** and **-n** for **examine**? Because **examine** is a VMS like command and **hd** is a U*X cloned command.)

```
>>>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' >$$ss
00000010  0a 65 63 68 6f 20 27 2d 2d 2d 27 20 3e 3e 24 24 .echo '---' >>$$
>>>
```

2.4.2 Examining Registers

The following examples show the **examine** and **deposit** commands used to reference registers. Registers may be addressed:

- *Symbolically,* for instance, **r0** or **ksp**
- *Explicitly,* as offsets within device address space; for instance, **gpr:0** or **ipr:0**
- *Implicitly,* as offsets within the current device address space; for instance, **0**

Also notice the usage of the symbolic relative addresses **+**, *****, **-**, and the implied address increment (no address specified):

```
>>>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                                   #                 R1,
gpr:                8 (    R1) 000000000000C408
>>>e                                     #    and the next R2.
gpr:               10 (    R2) 0000000000000000
>>>e ipr:0                               # Try 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.
ipr:                1 ( ASTEN) 0000000000000000
>>>e ksp                                 # One by name...
ipr:               12 (   KSP) 0000000000000F30
>>>e                                     #    and the next.
ipr:               13 (   ESP) 0000000000000000
>>>
```

The **examine** and **deposit** commands support symbolic representation of certain processor registers. In the following example, **pc**, **sp**, and **ps** are abbreviations for program counter, stack pointer, and process status longword:

```
>>>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
>>>
```

2.5 Using Pipes (|) and grep to Filter Output

The **grep** command is a convenient means of searching for information by filtering an input according to the expression argument supplied. A *pipe* (|) enables the output of one command to be the input to the next command without creating an intermediate file. Because the **grep** command requires input, a pipe is used to channel the output of the **examine** command into the **grep** command.

In the following example, **grep** is used 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. The **grep** command also can be used 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.

```
>>>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
>>>
```

2.6 Using I/O Redirection (>)

Although default output goes to the console, you can redirect output to other devices or files by using the redirection operator, >. In the following example, the output of an **examine** command is redirected to file *foo*, which is created dynamically out of the console's memory heap. The console **cat** command, similar to the OpenVMS **copy** command, is used in this example to display the contents of the created file *foo*. The **rm** command, similar to the OpenVMS **delete** command, is used to delete *foo*.

```
>>>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                                 # Does foo exist now?
foo
>>>cat foo                                # Yes! List foo.
pmem:          3FFF000 0000000000000000
pmem:          3FFF008 0000000000000000
>>>rm foo                                 # Remove (delete) file foo.
>>>ls foo                                 # Does foo exist now?
foo no such file
>>>                                       # No.
```

2.7 Running Commands in the Background "&"

In a design verification testing (DVT) environment, the ability to run tasks in the background is an especially helpful feature. You can execute any command in the background by placing the background operator **&** at the end of the command. This capability alone makes it extremely easy to generate random activity in a system.

In the following example, three processes are started in the background. The first process, invoked with the console **exer** command, performs reads to block 0 of disk *dub2* (which you determine by using the console **show device** command). Next, two instantiations of the console memory test are created, using the **memtest** command. In all three cases, the console immediately returns with the console prompt and awaits further commands.

```
>>>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.
>>>
```

2.8 Monitoring Status

The console monitors process status in several ways. The **ps** command is similar to the OpenVMS **show system** command, as shown below. The **grep** command can be used here to avoid unnecessary output.

```
>>>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
>>>
```

2.9 Killing a Process

To stop a process, use the process id from a **ps** command as the argument of the **kill** command.

```
>>>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
>>>
```

2.10 Creating Scripts

A *script* is a file that contains console commands. The console contains many built-in scripts that you may execute by simply typing the name of the script file. The powerup script is an example of such a built-in script.

The console also provides a crude means of creating scripts. In the following example, the **echo** command is used to write characters to file *foo* using the output creation operator, **>**. The script *foo* is then displayed and executed.

```
>>>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
>>>
```

In the next example, addition characters are appended to *foo*. Note the usage of the single quote **'** grouping character that encloses the desired text. The use of single quotes in the command line prevents the command-separator character (**;**) from prematurely terminating the **echo** command. Also notice that the output append operator (**>>**) is used to extend *foo*.

```
>>>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 may enter a much longer script by reordering the command. Open a string with the single quote ('), enter the script (on several lines), then close the string with a second single quote ('). For example:

```
>>>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
>>>
```

2.11 Using Flow Control

The console supports a limited number of flow control structures at the shell command level. The syntax for these constructions is as 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

Conditional Branching

Conditional branching in **if, while, until** loops is determined by the exit status of the command sequence following the control structure. In general, an exit status of zero indicates *success* and results in the execution of the *true* path.

In the following example, the **eval** command is used to extract an exit status from variable **junk**. The variable is initialized with the console **set** command.

```
>>>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
>>>
```

Byte Swapping

Byte swapping is a useful function when dealing with other bus architectures, such as Futurebus/Plus. Byte swapping is the transposition of bytes in a bus address. A simple script called **bswap** is created, which will transform a longword value by swapping most significant bytes to least significant bytes.

The **for..do..done** construction is used in conjunction with arbitrary environment variables **aa** and **bb** to pass arguments to the script. The **for** variable **aa** takes on the values of all the arguments on the command line at the invocation of **bswap**. The **set** command is used to create **bb**, which is simply **aa** with **0x** prepended to it. (This permits the user to enter hexadecimal numbers without having to specify the radix prefix **0x**.)

In following example, the **eval** command is used to perform the transformation. The **eval** command uses:

- **&** operator for logical AND function
- | operator for logical OR function
- >> operator for logical shift-right function
- << operator for logical shift-left function

A close inspection of the postfix expression should reveal how it works.

```
>>>echo > bswap 'for aa; do
_>set bb 0x$aa
_>eval -x "$bb 0xff & 24 << $bb 0xff00 & 8 << $bb 0xff0000 & 8 >> $bb 0xff000000 & 24 >> | | |"
_>done'
>>>bswap 12345678
78563412
>>>bswap 12 1234 123456 12345678
12000000
34120000
56341200
78563412
>>>
```

In the following example, a simple **for** loop is used to create a more generic process status command:

```
>>>echo > stat 'for i
_>do ps | grep $i
_>done'
>>>cat stat
for i
do ps | grep $i
done
>>>stat memtest
00000131 00114e80 2          0 00000001 0      memtest ready
>>>stat memtest ps
00000131 00114e80 2          0 00000001 0      memtest ready
00000167 00108ea0 3          0 00000001 0           ps running
>>>
```

2.12 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 a target AXPvme system.

The first step in this process is to create a script of console commands in the familiar OpenVMS environment, using your favorite editor to create the script. In this simple example, you use the OpenVMS **create** command to create a command script called **sample.**.

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

The next step in the process is to take **sample.** (a generic text file) and make it network loadable via the MOP load protocol. To accomplish this, you must run a little fixup program, **add_header.exe**, which appends a one-block header to the file to make it compatible with the MOP load server. For ease of access this executable program is provided on the Firmware Update CD at *[AXPVME]ADD_HEADER.EXE*. To simplify matters, it may be copied to the SYS$LOGIN area and defined as a foreign command, in this instance, **addhead**. To run the program, invoke **addhead,** supplying the desired script file as input and a name for the resulting network loadable output file.

Note

The current MOP load protocol only supports 15-character file names. Since the MOP server will assume a file extension of **.sys** whenever one is not specified with a request, **.sys** is the recommended file extension. This allows all fifteen characters to be used in the name.

In the following example, a directory of **sample** shows the difference in size of the two files. In the example, the CD-ROM device is attached to the system at dka2.

```
$ copy dka2:[axpvme]add_header.exe sys$login:*.*
$ addhead :== $sys$login:add_header.exe
$ addhead sample. sample.sys
$ dir /size sample

Directory USER:[SMITH]

SAMPLE.SYS;1              2
SAMPLE.;1                 1

Total of 2 files, 3 blocks.
$
```

You must place the resulting output file, in this example **sample.sys**, in the MOP server's load file directory, MOM$LOAD. On our system, the logical MOM$LOAD is a directory search list that points to one of the local directories, **wrk:[mopload]**. So, **sample.sys** is copied there. Now, when the MOP server sees a request for **sample.sys**, it can find it in its service area.

```
$ show logical mom$load
   "MOM$LOAD" = "MOM$SYSTEM" (LNM$SYSTEM_TABLE)
        = "WRK:[MOPLOAD]"
1  "MOM$SYSTEM" = "SYS$SYSROOT:[MOM$SYSTEM]" (LNM$SYSTEM_TABLE)
$ copy sample.sys wrk:[mopload]
$
```

At this point, the script file is available on the Ethernet segment of the MOP server. If the target AXPvme system is on the same Ethernet segment as the MOP server, the following console **cat** command will copy **sample.sys** over the network:

```
>>>cat mopdl:sample.sys/eza0          # Be patient!  The MOP protocol is slow.
show version
ls -l sample
>>>
```

The above string, **mopdl:sample.sys/eza0**, is the console's way of specifying that the file, **sample.sys**, may be accessed over the Ethernet device, **eza0**, using the MOP download protocol driver, **mopdl:**.

In the next example, the file extension **.sys** has been omitted. The MOP server will add the **.sys** to **sample**.

The **>** operator may be used to redirect the output of the **cat** command into a local file, in this case **sample**.

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

Once the **>>>** prompt returns, the file copy has completed. The resident script file, **sample** may then be displayed and executed using the following sequence of console commands:

```
>>>cat sample
show version
ls -l sample
>>>sample
version             V12.0-0 Oct 26 1994 12:58:38

rwx-   rd                512/2048           0   sample
>>>
```

3
Console Commands

Console mode provides the user interface that you enter when the power-up self-test (POST) completes. Console mode provides the following prompt:

>>>

You can also enter console mode in the following ways:

- You press the Halt or Reset switches on the front panel. Depending on your operating system and applications running at the time, this could damage application files.
- The module receives a VMEbus reset signal and switch 3 of the configuration switches on the AXPvme module is enabled. Depending on your operating system and applications running at the time, this could damage application files.
- You enter the operating system command to go to console mode.
- The operating system executes a HALT instruction.
- The operating system encounters a fatal error.
- The watchdog timer is enabled and the system software allows the timer to time out.

You leave console mode by issuing the boot, start, or continue commands.

The code that supports console mode is built-in to the AXPvme module and stored in the flash ROMs.

3.1 Console Commands

3.1.1 Special Keys

The following keys perform special functions:

- **Ctrl/U**—Ignores the current command line
- **Backspace/Delete**—Deletes a character within the command line
- **Ctrl/S and Ctrl/Q**—Ctrl/S suspends output to the console terminal; Ctrl/Q resumes output to the console
- **Ctrl/C**—Aborts the current command, if possible. The console program has no control over this once control has been passed to another program such as an operating system or loadable diagnostic.
- **Ctrl/R**—Retypes the current command line
- **Ctrl/O**—Causes the console code to throw away output characters rather than send them to the terminal; entering another Ctrl/O resumes sending output characters
- **Up and Down Arrow**—Used for command-line recall

3.1.2 Command Line Characteristics

The character sequence used for the prompt (>>>) is 0Dh, 0Ah, 0Dh, 3Eh, 3Eh, 3Eh, 20h which is <CR>, <LF>, <CR>, <quote>(>>>)-, <SP>. Host software executing a binary load on the console terminal port can look for this character string to determine when it may respond.

Commands are limited to 80 characters. Characters entered after the 80th character replace the last character in the buffer. Depending on your terminal, these 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.

Characters with codes greater than 7Fh are rejected by the parser. These characters are acceptable in comments.

Type-ahead is not supported. Characters received before the console prompt are checked for special characters (Ctrl/S, Ctrl/Q, Ctrl/C) but are otherwise discarded.

3.1.3 Radix Control

Numbers that you enter are, by default, interpreted as hexadecimal. You can change the radix of input with the set radix command or by entering %x before a number to specify hexadecimal or %d for decimal.

3.1.4 Console Command Dictionary

The following commands are supported by the AXPvme console program.

— comment character

The pound sign (#), wherever it appears on the line, prefixes a comment. The remainder of the line is ignored by the command interpreter.

Examples

1. `>>> chmod +x script # makes file script executable`

   The text "makes file script executable" is a comment.

alloc — allocates a block of memory

Exports the "malloc" routine out to the shell, so that users may allocate a block of memory from the heap. The resulting block may then be used simultaneously by several test routines (there can be several readers but only one writer).

Syntax

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

Arguments

size
Specifies the size (hex) in bytes of the requested block.

modulus
Specifies the modulus (hex) for the beginning address of the requested block.

remainder
Specifies the remainder (hex) used in conjunction with the modulus for computing the beginning address of the requested block.

Options

-flood
Flood memory with 0s. By default, alloc does not flood.

-z *heap_address*
Allocate from the memory zone starting at address *heap_address*. This address is usually obtained from the output of a "dynamic" command.

Examples

1.
```
>>> alloc 200
00FFFE00
>>> free fffe00
>>> set base 'alloc 400'
>>> show base
base                    00FFFC00
>>> memtest $base
>>> free $base
>>> clear base
```

Related Commands

dynamic, free

boot — bootstrap the system

Initializes the processor, loads a program image from the specified boot device, and transfers control to that image. If you do not specify a boot device in the command line, the default boot device is used. The default boot device is determined by the value of the *bootdef_dev* environment variable.

If you specify a list of devices, a bootstrap is attempted from each device in order. Then control passes to the first successfully booted image. In a list, always enter network devices last, since network bootstraps only terminate if a fatal error occurs or an image is successfully loaded.

The *-flags* option can pass additional information to the operating system about the boot that you are requesting.

The *-protocol* option allows you to select either the DECNET MOP or the TCP/IP BOOTP network bootstraps. The keywords mop or bootp are valid arguments for this option.

You can set the default protocol for a port by setting the environment variable, *ewa0_protocols* to the appropriate protocol. Explicitly stating the boot flags or the boot device overrides the current default value for the current boot request, but does not change the corresponding environment variable. **TFTP and BOOTP**

The following paragraphs give an overview of how the console implements TFTP and BOOTP clients to support network bootstrapping and file transfers in an Internet environment.

An important point to note is that Internet booting is a two-stage operation. First, BOOTP provides the client with information needed to obtain an image. Then, the client uses a second protocol: TFTP, to obtain the image. Both BOOTP and TFTP use UDP (User Datagram Protocol) as the primary transport mechanism to send datagrams to other application programs.

BOOTP is a standard protocol in the TCP/IP suite. It operates in the client-server paradigm and requires only a single packet exchange. The machine that sends the BOOTP request is the client and any machine that sends a reply is the server. A 300 byte database in the same format as the BOOTP message is used to store the received packet. Once a BOOTP packet is broadcast and received, the database is marked as initialized, thus ending the first-stage of the Inet Booting operation.

The second stage of Inet booting uses *TFTP* protocols to get the memory image. This protocol simply takes the information in the bootp packet (or uses a filename specified in the command string or *boot_file*) and gets the file from the server.

The first packet sent requests a file transfer and establishes the connection between client and server. The packet specifies a file name and whether the file will be read (transferred to the client) or written (not currently supported).

TFTP depends only on the unreliable, connectionless datagram delivery service (UDP).

TFTP accepts one parameter: the host address concatenated to the file name of the remote file to be read.

Internet Booting Hierarchy

The following prioritized list shows the different ways of Internet Booting from an initialized system.

1. Filename specified as named boot; e.g. *boot -file "filename" ewa0*.

If the filename includes a "/", it must be specified as "//". Here is an example:

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

This format is only used when specifying the named boot via *-file filename* or loading the filename into the environment variable, *boot_file*.

2. Filename placed in environment variable *boot_file*.

This really operates as a named boot, the only difference being that the file name it has is taken from *boot_file*. For example:

```
>>> set boot_file //var//adm//ris//ris0.alpha//vmunix
>>> boot ewa0
```

3. Filename placed in ev *ewa0_inetfile*.

Only TFTP protocol is used in this case; the bootp packet must have been initialized. When a file name has been placed in this environment variable, only the second stage of an Inet boot will occur. All other fields of the bootp packet should contain valid information; either from a previous Inet boot or via manual loading.

4. Filename placed in ev *ewa0_bootp_file*.

The file name specified in this environment variable will become the filename specified in the outgoing bootp request packet. For example:

```
>>> set ewa0_bootp_file /var/adm/ris/ris0.alpha/vmunix.old
```

5. Filename not specified - e.g *boot ewa0*.

None of the above environment variables are written, so a 2-stage boot occurs (any server that receives the request will reply).

A complete description of the Internet protocols is beyond the scope of this document. An excellent description may be found in Douglas Comer *Internetworking with TCP/IP, Vol I, Principles, Protocols and Architecture*, second edition, Prentice Hall.

Note that behavior of this firmware depends in part upon behavior of the software running on the server host, which varies from server to server. For example, the exact format of the file specification used with TFTP depends on the server: the Ultrix TFTP server requires a partial path name, and the OSF server requires a complete path name. Ultrix systems frequently name the TFTP server "tftpd" and the BOOTP server "bootpd"; see the appropriate system documentation for server details.

For bootp and tftp to operate reliably, several network parameters, contained in environment variables, must be properly configured. The Internet protocols are robust and thus may work intermittently if the parameters are misconfigured, which can make debugging a misconfiguration difficult. So here is what you need to know to get the Inet software working. Note that each network interface has a complete set of variables to itself and so each variable is prefixed with the name of the interface. The following discussion uses "ewa0" for specificity.

The variable *ewa0_protocols* should include the string "BOOTP" to enable BOOTP, TFTP, etc. (The variable may also include "MOP" to enable MOP.) Leaving the variable empty or including both strings will enable all protocols (currently just Inet and decnet). In particular, if not enabled the inet software will not

be invoked for booting. Also, the network driver may not enable reception of broadcast packets, which breaks ARP.

Each interface has a small database of information required to operate the inet software on that interface. Internally the database is kept in a 300 byte structure having the same format as a BOOTP packet. This database can be directly read and written in binary form through the BOOTP protocol driver; described later. The four most important fields of the database can be accessed in a friendlier fashion through the environment variables *ewa0_inetaddr, ewa0_sinetaddr, ewa0_ginetaddr,* and *ewa0_inetfile.* The first three are the Internet addresses for the interface (ewa0), the remote server host, and remote gateway host, respectively. These variables use Internet standard dotted decimal notation; e.g., "16.123.16.53". *ewa0_inetfile* contains a file to be booted and is formatted simply as a string.

The most important of these four is the local address, *ewa0_inetaddr*. TFTP and ARP will not operate properly without the correct address. *ewa0_ginetaddr* is the address of an Internet gateway on the local network. TFTP cannot communicate beyond the local network if this gateway address is not correct. *ewa0_sinetaddr* is the address of a server, which may or may not be on the local network. Ordinarily this is the server from which to boot. This is the default remote host contacted by TFTP. *ewa0_inetfile* is ordinarily the file to be booted. This should be a fully qualified file name, according to whatever rules are specified by the TFTP server on the remote host. This is the default file name requested by TFTP.

The interface database must be initialized somehow before TFTP can be used. The database can be initialized by manually setting the four database variables, by explicitly invoking BOOTP, or automatically on the first invocation of TFTP. Whether initialization occurs on the first TFTP depends on whether the database has been marked as initialized. The database will be marked as initialized on the first occurrence of any of three events: the invocation of TFTP, the invocation of BOOTP, or the setting of any of the four database environment variables.

The most common case is the invocation of TFTP. When TFTP is invoked and the database has not been marked initialized then the database will be automatically initialized by one of two methods, as specified by the environment variable *ewa0_inet_init*. If *ewa0_inet_init* is set to *bootp* (the default) the BOOTP protocol driver will be invoked to initialize the database by broadcasting a BOOTP request and storing the response in the database. If *ewa0_inet_init* is set to "nvram" then the database will be initialized by copying the contents of four nonvolatile default variables into the four database variables. The four nonvolatile default variables are *ewa0_def_inetaddr, ewa0_def_sinetaddr, ewa0_def_ginetaddr* and *ewa0_def_inetfile.* These variables obviously must be set in advance, 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_inetfile bootfiles/vmunix
>>>set ewa0_inet_init nvram
```

When BOOTP is invoked (either explicitly or via the automatic initialization discussed above) the database is marked as initialized. In the usual case where BOOTP is successfully invoked without the *bootp/ewa0* or *bootp:broadcast/ewa0*) the received reply packet is copied into the database, thus initializing it. If the "nobroadcast" parameter is specified (that is, *bootp:nobroadcast/ewa0*) then no request is broadcast and thus no reply is received to copy into the database. However, the database is still marked initialized, so a following TFTP will not automatically initialize the database.

When one of the four database environment variables is set the database is marked as initialized. Thus a following TFTP will not automatically initialize the database, regardless of whether the environment variables were set to sensible values.

TFTP, BOOTP, and ARP all use retransmission to improve robustness. If an initial transmission is not answered appropriately, the protocol software will retransmit. Each protocol has an environment variable which controls the number of retries before giving up. The variables are named *ewa0_arp_tries*, *ewa0_bootp_tries*, and *ewa0_tftp_tries*. The default value of these is 3, which translates to an average of 12 seconds before failing (see the discussion of retransmission timing below). If the value of one of these variables is less than 1, the protocol will fail immediately. Machines located on very busy networks or associated with heavily loaded servers may need these variables set higher.

The retransmission algorithms use a randomized exponential backoff delay. If the first try fails a second try will occur about 4 seconds later. A third try would come after another 8 seconds, a fourth after 16 seconds, and so forth up to 64 seconds. These times are actually averages, however, since random jitter of about +/- 50% is added to each delay. This implies that with *ewa0_arp_tries* set to 3 ARP will fail if it does not get a response within about 12 seconds on average, but the actual timeout will be somewhere between 6 and 18 seconds.

The normal use of BOOTP and TFTP is for bootstrapping across a network. However, they may be explicitly invoked as protocol drivers. The bootp and tftp protocols must be followed by a network in the protocol tower.

When a BOOTP request is broadcast, the environment variable *ewa0_bootp_server* is copied into the "sname" field of the request packet and the variable *ewa0_bootp_file* is copied into the "file" field of the request packet. The exact interpretation of these fields depends on the BOOTP server. The "sname" field should be the name of a specific host which the local machine wants to boot from. If it does not matter which server answers, then the variable *ewa0_bootp_server* should be left empty. The server should use the "file" field in the request to decide which boot file to specify in the response. For example, the client could supply a generic name like "unix" or "lat", and the server would respond with the fully qualified file path to be used with TFTP. If a machine will always be booting the same file then *ewa0_bootp_file* can be left empty.

The tftp protocol driver is used to read files across the network. Tftp accepts one parameter, the host address concatenated to the file name of the remote file to be read. The host address is specified in dotted decimal notation and is separated from the file name by ":". If the file name includes "/" they must be doubled to "//". The following example displays the file "/usr/foo/bar" from the host whose address is 16.123.16.242:

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

For convenience the address could be saved in an environment variable:

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

If no parameter is specified tftp uses the file name and server address from the interface database (that is, *ewa0_sinetaddr* and *ewa0_inetfile*).

Note that when booting with tftp, the boot command passes the contents of the environment variable *boot_file* as the parameter to tftp. If *boot_file* does not have the correct format tftp will fail. The most common use is probably to leave *boot_file* empty in which case tftp will default to using *ewa0_sinetaddr* and *ewa0_inetfile*, as above.

Syntax

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

Arguments

boot_device
A device path or list of devices from which the firmware attempts to boot, or a saved boot specification in the form of an environment variable. Use the *set bootdef_dev* command to define the default boot device.

Options

-file *filename*
Specifies the name of a file to load into the system. Use the set boot_file command to specify a default boot file.

-flags *longword*[,*longword*]
Specifies additional information to the operating system.

-protocols *enet_protocol*
Specifies the Ethernet protocol(s) to to be used for the network boot. Either the keyword *mop* or *bootp* may be specified. If both are specified, each protocol is attempted to solicit a boot server.

-halt
Forces the bootstrap operation to halt and invoke the console program once the image is loaded and page tables and other data structures are set up. Console device drivers are not shut down when this qualifier is present. Transfer control to the image by entering the continue command.

Examples

1. `>>> boot`

   The system boots from the default boot device. The console program returns an error message if a default boot device has not been set.

2. `>>> boot ewa0`

   The system boots from the Ethernet port, ewa0.

3. `>>> boot -file dec_4000.sys ewa0`

   The system boots the file named dec_4000.sys from Ethernet port ewa0.

4. `>>> boot -fi //usr//local//bootfile//bl12 -protocol bootp ewa0`

   The system performs a TCP/IP BOOTP network boot from Ethernet port ewa0.

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

   The system boots from the default boot device using boot flag settings 0,1.

6. `>>> boot -halt dka0`

   The system loads the operating system from the SCSI disk, dka0, but remains in console mode. Subsequently, you can enter the continue command to transfer control to the operating system.

Related Commands

set, show

break — break from a program loop

Break from a for, while, or until loop. Exit the current shell with a status or return the status of the last command.

Syntax

break ***break_level***

Arguments

break_level
Specifies the status code to be returned by the shell.

Options

None.

Examples

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

Related Commands

None

cat — copy files

Concatenates files that you specify to the standard output. If you do not specify files on the command line, cat copies standard input to standard output.

You can also copy or append one file to another by specifying I/O redirection.

Syntax

cat **[-l *length*] *file1* [*file2 . . .*]**

Arguments

***file1* [*file2 . . .*]**
Specifies the name of the input file or files to be copied.

Options

-l *length*
Specifies the number of bytes (decimal) of each input file to copy.

Examples

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

   Creates the file foo with the echo command. Then uses the cat command to send the contents of the file to the standard output, the console terminal screen.

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

   Sends the first 6 bytes of the file foo to the standard output, the console terminal screen.

Related Commands

echo, ls, rm

chmod — change file attributes

Changes the specified attributes of a file. The *chmod* command is a subset of the equivalent U*x command.

Syntax

chmod { - | + | = } {r,w,x,b,z} *file1* [*file2 . . .*]

Arguments

***file1* [*file2 . . .*]**
Specifies the file(s) or inode(s) to be modified.

Options

–
A minus sign indicates to remove the specified attribute(s).

+
A plus sign indicates to add the specified attribute(s).

=
An equals sign indicates to set the specified attribute(s) and clear all other attributes not included in the command.

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`

   Adds the executable attribute to the file, script.

2. `>>> chmod =r errlog`

   Sets the file errlog to read only.

3. `>>> chmod -w dk*`

   Makes all SCSI disks non writeable.

Related Commands

chown, ls -l

chown — change ownership of memory block

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

Syntax

chown ***pid address1*** **[*address2 . . .*]**

Arguments

pid
Specifies the process identifier (PID) (in hex) of the new owner. You can display PIDs with the ps command.

address1 **[*address2 . . .*]**
Specifies the address (hex) or list of addresses of allocated block(s) for which ownership is to be changed.

Options

None.

Examples

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

   For the first argument to *chown,* the command uses the *ps* command to display processes, pipes the output to grep to find the idle process . . . The second argument to *chown* calls *alloc 200* to return the starting address of the first free block of 200 bytes.

Related Commands

alloc, dynamic, ps

clear — delete environment variable

Deletes an environment variable from the name space.

Note that some environment variables, such as *bootdef_dev*, are permanent and cannot be deleted.

Syntax

clear ***variable_name***

Arguments

variable_name
Specifies the name of the environment variable to be deleted.

Options

None.

Examples

1.
```
>>> clear foo
>>>
```

   Deletes the environment variable foo.

Related Commands

set, show

clear_log — clear error log in NVRAM

This command is used for clearing and initializing the area of NVRAM used for console error logging. The entire area of NVRAM where fault information is stored is cleared to zero. Miscellaneous pointers, counters and initialization flags used in the error logging process are reset accordingly.

Note that the current contents of the NVRAM error log area is destroyed and lost forever. Without the *-nc* command-line Option, the user is prompted before actually clearing the log area.

Note that console error logging is completely independent of the OS error logging.

Syntax

clear_log

Arguments

None.

Options

-nc
No Confirmation, when specified; the user is not prompted before the NVRAM log area is cleared.

Examples

1.
```
>>> clear_log
Error Log data in NVRAM will be destroyed!!
Continue (y/n)?
y
Initializing NVRAM Error Log...
```

   The user is prompted to continue, then the NVRAM Error Log is initialized.

Related Commands

show_log

continue — resume program execution

Resumes program execution. The processor begins executing instructions at the address currently contained in the program counter. The processor is not initialized.

The *continue* command is only valid if an operator has halted the system by pressing the Halt button on the control panel or by entering CTRL/P on the console terminal.

Note that some console commands, such as test and boot, may alter the machine state so that program mode cannot be successfully continued.

Syntax

continue

Arguments

None.

Options

None.

Examples

1. `>>> continue`

   The processor leaves console mode and returns to operating system mode.

Related Commands

start, stop

crc — generate CRC for a file

Calculates a cyclic redundancy check (CRC) value for a file.

Syntax

crc [-s *start_offset*] [-e *end_offset*] [-l *bytes*] *file*

Arguments

file
Specifies the file on which to calculate the CRC.

Options

-s *start_offset*
Specifies the starting offset within the file.

-e *end_offset*
Specifies the ending offset within the file.

-l *bytes*
Specifies the length of the file. Specifies that the CRC be calculated on the first *bytes* number of bytes of the file.

Examples

1.
```
>>> cat foo
hello world
```

   Creates a file, foo.

2.
```
>>> crc foo
0x00000466
```

   Generates a CRC for file, foo.

3.
```
>>> hd foo
00000000  68 65 6c 6c 6f 20 77 6f 72 6c 64 0a hello world.
```

   Dumps the contents of the file, foo, in hexadecimal.

4.
```
>>> eval -x \
_>0x68 0x65 0x6c 0x6c 0x6f 0x20 0x77 0x6f 0x72 0x6c 0x64 0x0a \
_>    +    +    +    +    +    +    +    +    +    +    +
00000466
```

5.
```
>>> d -b -n 8 pmem:0 0
```

   Deposits 8 bytes of 0's into physical memory starting at address 0.

6.
```
>>> hd -l 8 pmem
00000000  00 00 00 00 00 00 00 00 ........
```

   Dumps the first 8 bytes of NVRAM.

7.
```
>>> crc -l 8 pmem
0x00000000
```

   Generates a CRC for the first 8 bytes of NVRAM.

8.
```
>>> d -b -n 8 pmem:0 1
```

   Deposits 8 bytes of 1's into NVRAM starting at address 0.

9.
```
>>> hd -l 8 pmem
00000000  01 01 01 01 01 01 01 01 ........
```

   Dumps the first 8 bytes of NVRAM.

10.
```
>>> crc -l 8 pmem
0x00000008
```

   Generates a CRC for the first 8 bytes of NVRAM.

11.
```
>>> crc -s 4 -l 4 pmem
0x00000004
```

   Skips the first 4 bytes and generates a CRC for the next 4 bytes (bytes 4-7) of NVRAM.

date — display or change time

Displays or modifies the current date and time. If you include no arguments, displays the current date and time. If you do include arguments, modifies the current date and time stored in the time-of-year (TOY) clock.

Note

The date will not be preserved if the TOY battery has been disabled with the *set toy sleep* command. On the next powerup of the module, the battery will be reenabled and the date may need to be reinitialized.

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 VMS and OSF/1 operating systems.

Syntax

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

Arguments

yyyymmddhhmm.ss
Specifies the new date and time, where:

- *yyyy* (0000-9999) is the year
- *mm* (01-12) is the two digit month
- *dd* (01-31) is the two digit day
- *hh* (00-23) is the two digit hour
- *mm* (00-59) is the two digit minute
- *ss* (00-59) is the two digit second

When you modify the date or time, you must specify at least the hour and minute fields (4 digits). If you include 6 digits, that is interpreted as the day, hour, and minute fields. Omitted fields are inherited.

Options

None.

Examples

1.
```
>>> date 199208031029.00
>>> date
10:29:04  August  3, 1992
>>>
```

Related Commands

None

deposit — write memory data

Writes data to the specified address: a memory location, a register, a device, or a 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 be written to.

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 response.

Syntax

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

Arguments

[*device*:]
Selects the device name or address space to access. The following devices are supported:

pmem: Physical memory.
vmem: Virtual memory. All access and protection checking occur. If the access would not be allowed to a program running with the current PS, the console 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. The following symbols for *address* are recognized: r0, r1, . . . r31, ai, ra, pv, fp, sp, and rz.
fpr: Floating Point register set. The data size defaults to quadword. The following symbols for *address* are recognized: f0, f1, . . . f31.
ipr: Internal Processor register set. The size defaults to quadword. The following symbols for *address* are recognized: 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: PAL Temporary register set, PT:0-PT:31 or PT0:-PT31:. The data size defaults to quadword.
pcicfg: PCI configuration space
pcidmem: PCI dense memory space
pcismem: PCI sparse memory space
pciio: PCI IO space
eerom: Environment variable and error log NVRAM
ferom: Intel 28F020 firmware FEPROM

toy: DS1386 registers, clock chip, and NVRAM

address
Specifies the address into which the data is deposited. The address may be any valid hexadecimal offset in the device's address space or it may be a symbolic address.

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.

You cannot use a symbolic address if you include the *device:* field. The following are valid symbolic addresses:

- **gpr** General Purpose register 0
- **fpr** Floating Point register 1
- **ipr** Internal Processor register
- **pt** or *pt0* through *pt31* PALtemp registers 0-31. The data size defaults to quadword; the address space defaults to pt.
- **PC** Names the Program Counter (execution address register). The last address, size, and type are unchanged.
- **+** Names the location immediately following the last location referenced in an examine or deposit. 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 referenced plus one.
- **-** Names the location immediately preceding the last location referenced in an examine or deposit. 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 referenced minus one.
- ***** Names the location last referenced by an examine or deposit.
- **@** Uses the data at the last location referenced by an examine or deposit as the address.

data
The data to be deposited. If the specified data is larger than the deposit data size, the console ignores the command and issues an error. If the specified data is smaller than the deposit data size, it is padded with leading zeros before deposit.

Options

-b
The data type is byte.

-w
The data type is word.

-l
The data type is longword.

-q
The data type is quadword.

-o
The data type is octaword (8 words).

-h
The data type is hexaword (16 words).

-d
The data displayed is the decoded macro instruction. Alpha instruction decode (-d) does not recognize machine-specific PAL instructions.

-physical
The address space is physical memory. Same as specifying the pmem: device.

-virtual
The address space is virtual memory. Same as specifying the vmem: device.

-gpr
The address space is general purpose registers. Same as specifying the gpr: device.

-fpr
The address space is floating point registers. Same as specifying the fpr: device.

-ipr
The address space is internal processor registers. Same as specifying the ipr: device.

-n *count*
Specifies the number (hex) of consecutive locations to modify. The console deposits to the first address, then to the specified number of succeeding addresses.

-s *step*
Specifies the address increment size (hex). Normally this defaults to the data size, but is overriden by the presence of this option. This option is not inherited.

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 GPRs 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.

Related Commands

examine

dynamic — show memory

Show 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] [-setsize] [-extend *byte_count*] [-z *heap_address*]

Arguments

None.

Options

-c
Perform a consistency check on the default heap or the heap specified with -z.

-r
Repair a broken heap by flooding free blocks with DYN$K_FLOOD_FREE if and only if they have been corrupted. Repairing broken heaps is dangerous at best, as it is masking 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 specifies with -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 -z.

-setsize
Set the total memory in the system to this size. Add/subtract the memory to the end of memzone.

-extend *byte_count*
Extend the default memory zone by the byte count at the expense of the main memory zone. This command assumes that these two zones are physically adjacent.

-z *heap_address*
Operate on the specified heap.

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
   .
   .
   .
>>>
```

Related Commands

alloc, free

echo — outputs text

Sends the text you enter on the command line to the current output device, your screen by default. The echo command separates arguments (words) in the line by blanks. Echo adds a newline character to the end of the line.

Whenever specifying pipes or I/O redirection, be explicit by enclosing the text within single quotes.

Syntax

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

Arguments

args...
Specifies any arbitrary set(s) of character strings.

Options

-n
Suppress newlines from output.

Examples

1.
```
>>> echo this is a test.
this is a test.
>>>
```

   Echo sends the character string to your screen.

2.
```
>>> echo -n this is a test.
this is a test.>>>
```

   Echo sends the character string to your screen but with no newline separating the string from the next console prompt (>>>).

3.
```
>>> echo 'this is a test' > foo
>>> cat foo
this is a test
>>>
```

   The string is piped 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 echo can be used to create a file several lines long.

Related Commands

cat

edit — edit a file

Edit is a console text editor which behaves much like a BASIC line editor. With edit, lines of a file may be added, inserted, or deleted. Note that the file must already exist. If you need to create a new file, see the echo command.

The editor may be used to modify the user powerup script, nvram, or any other user created file, such as foo in the examples below. Note that nvram is a special script which is always invoked during the powerup sequence. Hence, any actions you wish to execute on powerup can be saved in the non-volatile memory using the script file, nvram.

The editor will handle arbitrarily large input streams, up to the size of the heap.

Syntax

edit ***file***

edit subcommands: help, list, renumber, exit, CTRL/Z, quit, *n*, *n text*

Arguments

file
Specifies the name of the file to be edited. Note that the file must already exist.

Options

help
Displays a brief help file.

list
Displays all of the lines in the current file.

renumber
Renumbers the lines of the file by 10's.

exit
Saves the file and then leaves the editor.

quit
Leaves the editor and closes the file without saving changes.

CTRL/Z
CTRL/Z is the same as EXIT.

n
Deletes line number *n*.

n text
Adds or replaces line *n* with *text*.

Examples

1.
```
>>> echo > foo 'this is a test'      # Create a sample file.
>>> cat foo
this is a test
>>> edit foo                         # Edit the newly created file.
editing 'foo'
15 bytes read in
```

   Creates the file foo with the echo command, displays the contents of the file with the cat command, and then calls the file into the editor with the edit command.

2.
```
*help
Think "BASIC line editor", and see if that'll do the trick
```

   Shows the HELP command in the editor.

3.
```
*list
  10  this is a test
```

   Displays the current contents of the file with the list command.

4.
```
*20 of the console BASIC-like line editor
*30 This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
*list
  10  this is a test
  20  of the console BASIC-like line editor
  30  This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
```

   Adds new lines 20 and 30 and then lists the contents of the file.

5.
```
*10 This is a test of the console BASIC-like line editor.
*20
*list
  10  This is a test of the console BASIC-like line editor.
  30  This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
```

   Replaces line 10, deletes line 20, and then lists the contents of the file.

6.
```
*15 It may be used to create scripts files at the console.
*list
  10  This is a test of the console BASIC-like line editor.
  15  It may be used to create scripts files at the console.
  30  This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
```

   Adds line 15 and then lists the contents of the file.

7.
```
*renumber
*list
  10  This is a test of the console BASIC-like line editor.
  20  It may be used to create scripts files at the console.
  30  This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
```

   Renumbers the lines and then lists the contents of the file.

8.
```
*exit
168 bytes written out
```

   Saves the file and exits from the editor.

9.
```
>>> edit foo
editing 'foo'
168 bytes read in
*20
*list
  10  This is a test of the console BASIC-like line editor.
  30  This editor supports HELP, LIST, RENUMBER, EXIT, and QUIT.
*quit
```

Reads file foo into the editor, deletes line 20, then quits without saving the changes to the file.

10.
```
>>> edit nvram                  # Modify user powerup script, nvram.
editing 'nvram'
0 bytes read in
*10 set boot_dev ewa0
*20 set boot_osflags 0,0
*exit
37 bytes written out
>>> nvram                       # Execute the silent script, nvram.
>>> edit nvram
editing 'nvram'
37 bytes read in
*15 show boot_dev
*25 show boot_osflags
*list
  10  set boot_dev ewa0
  15  show boot_dev
  20  set boot_osflags 0,0
  25  show boot_osflags
*exit
69 bytes written out
>>> cat nvram                    # List the modified file.
set boot_dev ewa0
show boot_dev
set boot_osflags 0,0
show boot_osflags
>>> nvram                       # Execute nvram, note the SHOWs.
boot_dev                ewa0
boot_osflags            0,0
>>>
                                #
                                # Reset system, note nvram execution.
                                #
Medulla powerup script start
boot_dev                ewa0
boot_osflags            0,0
Medulla powerup script end

Medulla console V12.0-0, built on Oct 26 1994 at 12:58:38
```

Shows how to modify the non-volatile user-defined power-up script, nvram.

Related Commands

cat, echo

eval — evaluate expression

Evaluates a postfix expression.

Syntax

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

Arguments

operand1
The first numeric value to be evaluated.

operand2
The second numeric value to be evaluated.

operator
One of the following:

- + Add the operands.
- - Subtract *operand2* from *operand1*.
- * Multiply the operands.
- / Divide *operand1* by *operand2*.

Options

-ib
The operands are entered in binary.

-io
The operands are entered in octal.

-id
The operands are entered in decimal.

-ix
The operands are entered in hexadecimal.

-b
Display the output in binary.

-o
Display the output in octal.

-d
Display the output in decimal.

-x
Display the output in hexadecimal.

Examples

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

   The sum of 5 plus 10 is 15.

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

   The sum of 5 plus 10 (hex) is 21 (decimal).

examine — display memory data

Displays the contents of the specified address: a memory location, a register, a device, or a 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 response.

The display line consists of the device name, the hexadecimal address (or offset within the device), and the examined data, also in hexadecimal.

EXAMINE uses the same options as DEPOSIT. Additionally, the EXAMINE command supports instruction decoding, the -d option, which disassembles instructions beginning at the current address.

Syntax

e[xamine] [**-b** | **-w** | **-l** | **-q** | **-o** | **-h** | **-d**] [**-physical** | **-virtual** | **-gpr** | **-fpr** | **-ipr**] **[-n** ***count*****]** **[-s** ***step*****]** **[***device***:]*****address data***

Arguments

[*device*:]
Selects the device name or address space to access. The following devices are supported:

pmem: Physical memory.
vmem: Virtual memory. All access and protection checking occur. If the access would not be allowed to a program running with the current PS, the console issues an error message. If memory mapping is not enabled, virtual addresses are equal to physical addresses.
gpr: General Purpose register set, R0-R31. The data size defaults to -q.
fpr: Floating Point register set, F0-F31. The data size defaults to -q.
ipr: Internal Processor register set
pt: PAL Temporary register set, PT0-PT31. The data size defaults to -q.
pcicfg: PCI configuration space
pcidmem: PCI dense memory space
pcismem: PCI sparse memory space
pciio: PCI IO space
eerom: Environment variable and error log NVRAM
ferom: Intel 28F020 firmware FEPROM

toy: DS1386 registers, clock chip, and NVRAM

address
Specifies the address into which the data is deposited. The address may be any valid hexadecimal offset in the device's address space or it may be a symbolic address.

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.

You cannot use a symbolic address if you include the *device:* field. The following are valid symbolic addresses:

- **gpr-***name* Names a General Purpose register. The size defaults to quadword; the address space defaults to gpr. The following symbols for *name* are recognized: r0, r1, . . . r31, ai, ra, pv, fp, sp, and rz.
- **fpr-***name* Names a Floating Point register. The size defaults to quadword; the address space defaults to fpr. The following symbols for *name* are recognized: f0, f1, . . . f31.
- **ipr-***name* Names an Internal Processor register. The size defaults to quadword; the address space defaults to ipr. The following symbols for *name* are recognized: 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 PALtemp register. The data size defaults to quadword; the address space defaults to pt. The following symbols for *name* are recognized: pt0, pt1, . . . 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 referenced in an examine or deposit. 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 referenced plus one.
- **-** Names the location immediately preceding the last location referenced in an examine or deposit. 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 referenced minus one.
- ***** Names the location last referenced by an examine or deposit.
- **@** Uses the data at the last location referenced by an examine or deposit as the address.

Options

-b
The data size is byte.

-w
The data size is word.

-l
The data size is longword.

-q
The data size is quadword.

-o
The data size is octaword.

-h
The data size is hexaword.

-d
The data displayed is the decoded macro instruction. Alpha instruction decode (-d) does not recognize machine-specific PAL instructions.

-physical
The address space is physical memory. Same as specifying the pmem: device.

-virtual
The address space is virtual memory. Same as specifying the vmem: device.

-gpr
The address space is general purpose registers. Same as specifying the gpr: device.

-fpr
The address space is floating point registers. Same as specifying the fpr: device.

-ipr
The address space is internal processor registers. Same as specifying the ipr: device.

-n *count*
Specifies the number of consecutive locations to examine.

-s *step*
Specifies the address increment size (hex). Normally this defaults to the data size, but is overriden by the presence of this option. This option is not inherited.

Examples

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

2.
```
>>> e -g 0
gpr:                0 (    R0) 0000000000000002
```
Examine GPR 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 (PC)

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

   Examine the GPR SP 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 R7 plus the 5 following GPR's.)

7.
```
>>> examine ipr:11
ipr: 00000011 (  SCBB) 2004A000
```

   Examine the SCBB, Internal Processor register (IPR) 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.

Related Commands

deposit, hd

exer — exercise one or more devices.

Exercise one or more devices by performing read, write, and compare operations. Optionally, report performance statistics.

A read operation reads from a device into a buffer. A write operation writes from a buffer to a device. A compare operation compares the contents of the two buffers.

The exer command uses two buffers, buffer1 and buffer2. A read or write operation can be performed using either buffer. A compare operation uses both buffers.

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

- An address range to test within the device(s)
- The packet size, also known as the IO size, which is the number of bytes read or written in each IO operation
- The number of passes to run
- The number of seconds to run
- A sequence of individual operations performed on the test device(s). You specify this with the action string qualifier.

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_name1* [*device_name2*]

Arguments

***device_name1* [*device_name2*]**
Specifies the name(s) of the device(s) or filestream(s) to be exercised.

Options

-sb *start_block*
Specifies the starting block number (hex) within the filestream. The default is 0.

-eb *end_block*
Specifies the ending block number (hex) within the filestream. The default is 0.

-p *pass_count*
Specifies the number of passes to run the exerciser. If 0, then run forever or until CTRL/C. The default is 1.

-l *blocks*
Length specifies the number of blocks (hex) to exercise. Option l has precedence over eb. If only reading, then specifying neither l nor eb defaults to read till end-of-file (EOF). If writing, and neither l nor eb are specified then exer will write for the size of device. The default is 1.

-bs *block_size*
Specifies the block size (hex) in bytes. The default is 200 (hex) except for tape drives which default to 800 (hex). The maximum block size allowed with variable length block reads is 800 (hex) bytes.

-bc *block_per_io*
Specifies the number of blocks (hex) per I/O. The default is 1.

-d1 *buf1_string*
Character string that is run through eval and then loaded into buffer1 to initialize the buffer. By default, the buffer is loaded with alternating 5's and A's (hex).

-d2 *buf2_string*
Character string that is run through eval and then loaded into buffer2 to initialize the buffer. By default, the buffer is loaded with alternating 5's and A's (hex).

-a *action_string*
Specifies an exerciser 'action string', which determines the sequence of reads, writes, and compares to various buffers. The default action string is '?r'. The action string characters are:

- **r** Read into buffer1
- **w** Write from buffer1
- **R** Read into buffer2
- **W** Write from buffer2
- **n** Write without lock from buffer1
- **N** Write without lock from buffer2
- **c** Compare buffer1 with buffer2
- **-** Seek to file offset prior to last read or write
- **?** First seek to a random block offset within the specified range of blocks. Next exer calls the program, random, to "deal" each of a set of numbers once. Then exer chooses a set which is a power of two and is greater than or equal to the block range. Each call to random results in a number which is then mapped to the set of numbers that are in the block range and exer seeks to that location in the filestream. Since exer starts with the same random number seed, the set of random numbers generated will always be over the same set of block range numbers.
- **s** Sleep for the number of milliseconds specified by the delay qualifier. If no delay qualifier is present, sleep for 1 millisecond. NOTE: times reported in verbose mode will not necessarily be accurate when this action character is used.

-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 passcount are processed.

-m
Specifies metrics mode, reports throughput at the end of the exercise.

-v
Specifies verbose mode, data read is also written to *stdout*. This is not applicable on writes or compares.

-delay *millisecs*
Specifies the number of milliseconds to delay when "s" appears in the action string.

Description

Exercise one or more devices. As described in the preceding overview section, exer uses two buffers, buffer1 and buffer2. Buffer1 and buffer2 are in main memory in the 'memzone' heap.

Both buffer1 and buffer2 are initialized to a data pattern before any IO operations occur. These buffers are never reinitialized, even after completing one or more passes. The data patterns that the buffers are initialized with are either a hex 5A in every byte of each buffer or it is specified via the string arguments to the optional data pattern qualifiers, -d1, -d2

The d1, d2 qualifiers use a postfix string argument to initialize a buffer's contents as follows. For each byte in the specified buffer, starting with the first byte, this postfix string is passed to the eval command which returns a byte value which is then written to the specified buffer.

Several exer qualifiers are used to specify the amount of device data to be processed. The qualifiers -sb, -eb, -l, -bs, and -bc specify, respectively: starting block, ending block, number of blocks, block size in bytes, and the number of blocks in a packet, where a packet is the amount of data transferred in one IO operation.

Reading, writing, comparing buffers, and other operations can be specified to occur in various combinations and sequences. These operations are specified by a string of one-character command codes known as the "action string". The action string is specified as an argument to the action string qualifier, -a.

Each command code character in the action string is processed in a sequence from left to right. Each time that exer completes all of the operations specified by the action string, exer will reduce the remaining amount of device data to be processed by the size of the last packet processed by the action string. The action string is repeatedly processed until the specified amount of device data has been processed.

Lower-case action string characters, rwn, specify operations that involve buffer1. Upper-case action string characters, RWN, specify operations that involve buffer2. The action string character, c, involves both buffers. The action string characters, -?, do not involve either buffer.

A random number generator can be used to seek to varying device locations before performing either a read or write operation. Randomization is achieved by calling the function, random, which uses a linear congruential generator (LCG) to generate the numbers. This algorithm isn't truly random, but it comes closest to meeting the needs of exer. Each time that random is called, it returns a number from a specified range. If the range of numbers is a power of two, then each subsequent call to random, 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, then random is used with an upper bound that is greater than the actual range size but is a power of two. Then a modulus

operation with the range size is done to the number that random returns, thereby ensuring that a random number is generated within the random range size.

The total number of bytes read or written on each pass of the exerciser is specified by the length in blocks or the starting/ending block address option arguments. If neither the ending address nor the length options are specified, then on each pass the number of bytes processed could vary depending on whether or not the filestream is being written to or just being read. If the filestream is not being written to by exer, then exer will read until EOF is reached. If exer will be writing to the file (as specified in the action string), then the number of bytes processed per pass is equal to the allocation size of the file which is usually larger than the length of the file for RAM disk files, but equal to the length for disk devices.

Note that disk device I/O will fail if the blocksize is not equal to 1 or a multiple of 512. Partial block read/writes are not supported so a length which is not a multiple of the blocksize will result in no errors, but the last partial block I/O of data won't occur.

Any combination of writing, reading, or comparing the buffer1 and buffer2 can be executed in the sequence as specified in the action string. Depending on the option arguments, one or two of these three operations (read/write/compare) may be omitted without affecting the execution of the other operations.

The *exer* command will return an error code immediately after a read, write, or compare error, if the *d_harderr* environment variable is set to "halt". When an error occurs and *continue* or *loop* on error is specified, then subsequent operations specified by the action string qualifier will occur except for compares. For instance, if a read error occurs, a subsequent compare operation will be skipped since a read failure preceding a compare operation guarantees that the compare will fail. If subsequent block I/O's succeed, then compares of those blocks will occur. When *exer* terminates because of completing all passes or by operator termination, the status returned will be that of the last failed write, read, or compare operation, regardless of subsequent successful IO's.

Examples

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

   Read all SCSI type disks for the entire length of each disk. Repeat this until 36000 seconds, 10 hours, have elapsed. All disks will be read concurrently. Each block read will occur 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 hex 5a's 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
```

A destructive write test over block numbers 0 through 100 on disk dka0. The packet size is 2048 bytes. The action string specifies the following sequence of operations:

1. Set the current block address to a random block number on the disk between 0 and 97. A four block packet starting at block numbers 98, 99, or 100 would access blocks beyond the end of the length to be processed so 97 is the largest possible starting block address of a packet.
2. Write from buffer1 (contains the previously read data) to the current block address.
3. Set the current block address to what it was just prior to the previous write operation.
4. From the current block address read a packet into buffer2.
5. Compare buffer1 with buffer2 and report any discrepancies.
6. Repeat steps 1 through 5 until enough packets have been written to satisfy the length requirement of 101 blocks.

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

A non-destructive write test with packet sizes of 512 bytes. The action string specifies the following sequence of operations:

1. Set the current block address to a random block number on the disk.
2. From the current block address on the disk, read a packet into buffer1.
3. Set the current block address to the device address where it was just before the previous read operation occurred.
4. Write a packet of hex 5a's from buffer1 to the current block address.
5. Set the current block address to what it was just prior to the previous write operation.
6. From the current block address on the disk, read a packet into buffer2.
7. Compare buffer1 with buffer2 and report any discrepancies.
8. Repeat the above 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. A packet size of 10 bytes is used. Since the length specified is also 10 bytes then only one write occurs. Delete the environment variable, myd. The hd, hex dump of foo shows the contents of foo after exer is run.

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
```

Related Commands

memexer

exit — exit current shell

Exit the current shell with the specified status or return the status of the last command executed.

Syntax

exit ***exit_value***

Arguments

exit_value
Specifies the status code to be returned by the shell.

Options

None.

Examples

1. `>>> exit`

   Exits returning the status of the previously executed command.

2. `>>> exit 0`

   Exits with success status.

3. `>>> test || exit`

   Runs test and exits if there is an error.

Related Commands

None

false — return failure status

Return a failure status.

Syntax

false

Arguments

None.

Options

None.

Examples

1.
```
>>> while false ; do echo foo; done
>>>
```

free — deallocate memory

Frees a block of memory that has been allocated from a heap. The block is returned to the appropriate heap.

Syntax

free ***address1*** **[*address2* . . .]**

Arguments

***address1 address2* . . .**
Specifies an address (hex) or list of addresses of allocated block(s) to be returned to the heap.

Options

None.

Examples

1.
```
>>> alloc 200
00FFFE00
>>> free fffe00
>>> free ‘alloc 10‘ ‘alloc 20‘ ‘alloc 30‘
>>>
```

Related Commands

alloc, dynamic

grep — search for regular expressions

Globally search for regular expressions and print any lines containing occurrences of the regular expression. A regular expression is a shorthand way of specifying a wildcard type of string comparison. Since grep is line oriented, it only works on ASCII files.

Syntax

grep [-c] [-i] [-n] [-v] { *expression* -f *file* } [*file1*] [*file2* . . .]

Arguments

expression
Specifies the regular expression to search for. If you include metacharacters, enclose the expression with quotes to avoid interpretation by the shell.

Grep supports the following metacharacters:

^ Matches the beginning of line
$ Matches end of line
. Matches any single character
[ ] Set of characters; [ABC] matches either A or B or C.

- A dash (other than first or last of the set) denotes a range of characters: [A-Z] matches any upper case letter.
- If the first character of the set is ^, then the sense of match is reversed: [^0-9] matches any non-digit.
- The following characters need to be preceded with backslash (\) if they occur in a set: \,], -, and ^.

* Repeated matching; when placed after a pattern, indicates that the pattern should match any number of times. For example, [a-z][0-9]* matches a lower case letter followed by zero or more digits.
+ Repeated matching; when placed after a pattern, indicates that the pattern should match one or more times. For example, [0-9]+ matches any sequence of one or more digits.
? Optional matching; indicates that the pattern can match zero or one times. For example, [a-z][0-9]? matches a lower case letter alone or followed by a single digit.
Quote character; prevents the character following the backslash from having special meaning.

file...
Specifies the file(s) to be searched. If omitted, then *stdin* is searched.

Options

-c
Prints only the number of lines that matched.

-i
Ignores case in the search. By default grep is case sensitive.

-n
Prints the line numbers of the matching lines.

-v
Prints all lines that do not contain the expression.

-f *file*
Take the regular expression from a file instead of 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
```

The output of the ps command (*stdin*) is searched 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
>>>
```

In this example, grep is used to search for all quadwords in a range of memory which are non-zero.

Related Commands

None

hd — dump file

Dump the contents of a file in both hexadecimal and ASCII.

Syntax

hd [-s *start_byte*] [-e *end_byte*] [-l *bytes*] *file1* [*file2 . . .*]

Arguments

file file2 . . .
Specifies the file or files to be displayed.

Options

-s *start_byte*
Specifies the starting offset within the file.

-e *end_byte*
Specifies the ending offset within the file.

-l *bytes*
Specifies length, the number of bytes to dump.

Examples

1.
```
>>> cat fred
a script called fred.
```

   Creates file, fred.

2.
```
>>> hd fred
00000000  61 20 73 63 72 69 70 74 20 63 61 6C 6C 65 64 20 a script called
00000010  66 72 65 64 0A fred.
```

   Dumps the contents of file, fred.

3.
```
>>> hd -l 16 foo
00000000  72 2d 2d 2d 20 20 20 6e 6c 20 30 30 30 30 30 30 r---   nl 000000
```

   Dumps the first 16 bytes of file, foo.

4.
```
>>> hd -s 512 -e 522 foo
00000200  20 20 72 64 20 30 30 30 31 37   rd 00017
```

   Dumps from bytes 512 to 522 in file foo.

5.
```
>>> hd -s 512 -l 10 foo
00000200  20 20 72 64 20 30 30 30 31 37   rd 00017
```

   Dumps file, foo, starting at byte 512 and dumping the next 10 bytes.

Related Commands

cat

help, man — help on commands

Defines and shows the syntax for each command that you specify on the command line. If you do not specify a command, displays information about the help command and lists the commands for which additional information is available.

For each argument (or command) on the command line, help tries to find all topics that match that argument. For example, if there are topics on exit, examine, and entry, the command "help ex" would display the help text for both exit and examine.

Wildcards are supported, so that "help *" generates the expected behavior. Topics are treated as regular expressions that have the same rules as regular expressions for the shell. For more information on regular expressions, see the grep command. Help topics are case sensitive.

When the help command describes command syntax, the following conventions are used.

- <item> Angle brackets denote a variable for which you must specify a value.
- [<item>] Square brackets enclose optional parameters, qualifiers, 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, c.

You can use the commands help and man interchangeably.

Syntax

help or man [*command1*] [*command2 . . .*]

Arguments

command1 command2 . . .
Specifies the command(s) or topic(s) for which you request help.

Options

None.

Examples

1. `>>> help` `# List all topics.`

   Requests a list of topics for which help is available.

2. `>>> help *` `# List all topics and associated text.`

   Requests help on all topics.

3. `>>> help ex`

   Requests help on all commands that begin with 'ex'.

4. `>>> help boot`

   Requests help on the boot command.

Related Commands

None

initialize — initializes console, a device, or the processor

Initializes the console, a device, or the processor.

Syntax

init[ialize] [-c] [-d *device*]

Arguments

None

Options

-c
Specifies that the console be initialized.

-d *device*
Specifies the device to be initialized.

Examples

1. `>>> init`

   Initializes the processor.

2. `>>> initialize -d ewa0`

   Initializes device ewa0.

init_ev — set all environment variables to their default values.

Sets all environment variables to their default values.

Once this command is issued, a system reset or the *init* command is required to set the environment variables to their default values.

Syntax

init_ev

Arguments

None

Examples

1.
```
>>> 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!!

>>>
```

   A system reset or the *init* command is now required.

kill — delete process

Delete the process(es) listed on the command line. Processes are killed by making a kernel call with the process id (PID) as the argument.

Syntax

kill ***pid1*** **[*pid2 . . .*]**

Arguments

pid1 pid2 . . .
Specifies the PID(s) of the process(es) to be killed. You can display PIDs with the ps command.

Options

None.

Examples

1.
```
>>> memtest -p 0 &
>>> ps | grep memtest
000000f1 00217920 2       9357 ffffffff 0      memtest ready
>>> kill f1
>>> ps | grep memtest
```

Runs memtest. Displays memtest's PID (f1) with the ps and grep commands. Deletes the process with the kill command. Displays the memtest process again to show that it is now gone.

Related Commands

ps

line — read a line

Copies one line (up to a new-line) from the standard input channel of the current process to the standard output channel of the current process. Always outputs at least a new-line.

Often used 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

Arguments

None.

Options

None.

Examples

1.
```
>>> line
type a line of input followed by carriage return
type a line of input followed by carriage return
```

   The line you typed is copied to your screen.

2.
```
>>> line >foo
type a line of input followed by carriage return
>>> cat foo
type a line of input followed by carriage return
```

   Shows line command used interactively.

3.
```
>>> echo -n 'continue [Y, (N)]? '
>>> line <tt >tee:foo/nl
>>> if grep <foo '[yY]' >nl; then echo yes; else echo no; fi
>>>
```

   Shows line command used within a script.

Related Commands

None

ls — list files

List 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] [*file1*] [*file2 . . .*]

Arguments

file1 file2 . . .
Specifies the file(s) or inode(s) to be listed. If omitted, lists all files and inodes on the system.

Options

-l
Specifies to list in long format. Each file or inode is listed on a line with additional information. By default just file names are listed.

Examples

1.
```
>>> ls examine
examine
```

   Lists the file, 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.

Related Commands

None

memexer — memory exerciser

Start the specified number of Gray code 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 pass count is 0 to run the started tests forever.

Nothing is displayed unless an error occurs.

Syntax

memexer [*number_of_tests*]

Arguments

number_of_tests
Specifies the number of memory test processes to start. The default is 1.

Options

None.

Examples

1.
```
>>> memexer 2 &
>>>
```

   Starts two memtests running in the background. Tests in blocks of 2 times the backup cache size across all available memory.

Related Commands

memtest

memtest — memory test

This exerciser contains a graycode memory test, a march memory test, a random memory test, and a victim block test. For the first test a graycode pattern and inverse graycode pattern are written, read and verified for the specified address range in test 1. For the second test, a marching pattern and inverse marching pattern are written, read, and verified for the specified address range. The third test will test random addresses within the specified range with random data of random length. The fourth test will perform block writes of data, victimize the data, and then read back the block and verify it. This will be performed for the specified address.

Detailed Description

When a starting address is specified, the memory is *malloc*'d beginning at the starting address - 32 bytes for the length specified. The extra 32 bytes that are malloc'd are reserved for the malloc header information. Therefore, if a starting address of 0xa00000 and a length of 0x100000 is requested, the area from 0x9fffe0 through 0xb00000 is reserved. This is transparent to the user, but may be confusing if the user attempts to begin two memtest processes simultaneously with one beginning at 0xa00000 for a length of 0x100000 and the other at 0xb00000 for a length of 0x100000. The second memtest process will state to the user that it is "Unable to allocate memory of length 100000 at starting address b00000". Instead, the second process should use the starting address of 0xb00020.

Memtest Test 1 - Graycode Test This test uses a graycode algorithm to test a specified section of memory. The graycode algorithm used is: data = (x>>1)^x. Where *x* is an incrementing value.

Three passes are made of the memory under test. The first pass writes alternating graycode inverse graycode to each longword. This will cause all but one data bit to toggle between each longword write. For example graycode(0)=0x00000000 while inverse graycode(1)=0xFFFFFFFE.

The second pass reads each location verifies the data and writes the inverse of the data. The read-verify-write is done one longword at a time. This will cause: all data bits to be written as a one and zero; all but one data bit toggle between longword writes; and will identify address shorts.

The third pass reads and verifies each location.

The *-f* "fast" option can be specified so the verify sections of the second and third loops is not performed. This will not catch address shorts but will stress memory with a higher throughput. The ECC/EDC logic will be used to detect failures.

Memtest Test 2 - March Test This test uses a marching 1's/0's algorithm to test a specified section of memory. The same range can be tested as in the graycode, test 1. The default data patterns used by this test are 0x55555555 and its inverse 0xAAAAAAAA. The data pattern can be altered by using the *-d* qualifier. The pattern entered and its compliment will then be used instead.

Three passes are also made of the memory under test. The first pass will write the data pattern entered (or default) beginning at the starting address and marching through for the entire length specified.

The next pass will begin again at the starting address and read the previously written data pattern and write back its inverse. This is done a longword at a time for the entire specified length.

The last pass will being at the end of the testing region and again read back the previously written inverse pattern and write back 0's. This will also be performed a longword at a time decrementing up through memory until the starting address is reached.

Memtest Test 3 - Random Test This test will perform writes with random data to random addresses using random data size, lengths, and alignments. The run time of the random test may be noticeably longer than that of the other tests because it requires two calls to the console firmware's random number generator every time data is written.

The random test will access every memory location within the boundaries specified by the *-sa* and *-l* qualifiers (as long as the length is less than 8MB—with lengths greater than 8 MB a modulo function is required on the seed and therefore some addresses may get repeated and some not tested at all). It first will obtain an address index into the Linear Congruential Generator structure dependent on the length specified. It will obtain the data index as a function of the entered random data seed and the maximum 32 bit data pattern. Using the address index and an initial address seed of 0, the random number generator is called to obtain a random address. It is then called again, using the data index and initial user entered data seed, (*-rs* qualifier), or default of 0, to get the longword of data to use in testing. The lower bit of the random data returned will also be used to determine whether to perform longword or quadword transactions. (Using the lower bit merely saves another call to the random function to help speed up the test). The data is then stored to the random address, and memory barrier is performed to flush it out to the B-Cache and then a read is done to read it back in. A compare is then done on the data written and the data read. In the case of quadword writes and reads, the longword of random data is shifted left by 32 and or'd with the original's compliment to form the quadword.

Memtest test 4 - Victim Eject Test The user must first set up a block of data to be used in the test. The address of this block of data will be read as an input to the test using the *-ba* qualifier. (default is block of data containing 4 longwords of 0xF's, then 4 longwords of 0's, then 4 longwords of 0xF's and lastly 4 longwords of 0's.)

First, the test will perform a write of the block of data to the specified starting address. It will then add 4 MB to the starting address and perform another write of arbitrary data. This will cause the original data to be 'victimized' to memory. A read will then be performed of the original starting address and verified that it is correct. The starting address is then incremented by a block and the write /write/read procedure repeated for the specified length of memory.

If MEMTEST is used to test large sections of memory, it may take a while for testing to complete. If a ^c or *kill <PID>* is done in the middle of testing, MEMTEST may not abort right away. For speed reasons, a check for a ^c or *kill* is done outside of any test loops. If this is not satisfactory, the user may run concurrent MEMTEST processes in the background with shorter lengths within the target range.

Syntax

memtest [-sa *start_address*] [-ea *end_address*] [-l *length*]
[-bs *block_size*] [-i *address_inc*] [-p *pass_count*]
[-d *data_pattern*] [-rs *random_seed*] [-rb]
[-f] [-m] [-z] [-h] [-mb] [-t] [-g] [-se]

Arguments

None.

Options

-sa *start_address*
Specifies the starting address for the test. The default is the first free space in memzone.

-ea *end_address*
Specifies the ending address for the test. The default is *start_address* plus *length*.

-l *length*
Specifies the length of the section to test in bytes. The default is *block_size*, except with the -rb option which uses the zone size. The -l option has precedence over -ea.

-bs *block_size*
Specifies the block size (hex) in bytes The default is 8192 bytes. This is only used for the random block test. For all other tests the block size equals *length*.

-i *address_inc*
increment value; This value will be used to increment through the memory to be tested. Default = 0 (no increment) This is only implemented for the graycode test. The increment value is in quadwords(ie. increment of 1 tests every other quadword). The -z flag must be set to test an unaligned start address. This option will be useful for multiple CPUs testing the same physical memory.

-d *data_pattern*
> used only for march test (2)—will use this pattern as test pattern, default = 5's

-p *pass_count*
Specifies the number of times to execute the test. If 0, then run forever or until CTRL/C. The default is 1.

-rs *random_seed*
Specifies the random seed. Used only with -rb. The default is 0.

-rb
Specifies to randomly allocate and test all of the specified memory address range. Allocations are done of *block_size*.

-f
Specifies fast mode. If -f is specified, the data compare is omitted. Only ECC/EDC errors are detected.

-m
Specifies to time the memory test. At the end of the test the elapsed time is displayed. By default the timer is off.

-z
Specifies the test will use the specified memory address without an allocation. This bypasses all checking but will allow testing in addresses outside of the main memory heap. It will also allow unaligned testing. **Warning:** This flag permits testing and corrupting *any* memory!

-h
Allocate test memory from the firmware heap.

-mb
Use memory barriers after each memory access. This flag is only used in the -f Gray code test. When set, an Alpha *mb* instruction will be done after every memory access. This will guarantee serial access to memory.

-t
test mask; default = run all tests in selected group. The invidual tests are as follows.

1. graycode test
2. march test
3. random test
4. victim eject test

-g
group name - [field, mfg, exception, dvt] only mfg specific tests are supported now.

-se
soft error threshold

Examples

1. `>>> memtest -sa 200000 -l 1000`

   Tests memory starting at 0x200000 (-sa) for 0x1000 bytes (-l).

2. `>>> memtest -sa 200000 -l 1000 -f`

   Tests memory from 0x200000 for 0x1000 bytes, but data is not verified (-f).

3. `>>> memtest -sa 300000 -p 10`

   Writes a default block size of 8192 bytes from 0x300000 for 10 passes (-p).

4. `>>> memtest -f -mb`

   Tests memory in arbitrary 8192 byte blocks without verification. After each read and write to memory an MB (memory barrier) instruction is executed (-*mb*).

5. `>>> memtest -sa 200000 -ea 400000 -rb`

   Tests memory from 0x200000 to 0x3fffff. Every block within this range is randomly allocated (-rb). With -rb memtest will not error if a block within the range can't be allocated.

6. `>>> memtest -h -rb -bs 100`

   Tests the console heap (-h) by randomly mallocing 0x100 byte blocks (-bs).

7. `>>> memtest -rb -p 0`

   Tests memory across all of memzone (all memory excluding the HWRPB, the PAL area, the console, and the console heap). It is run in the foreground until CTRL/C.

Related Commands

memexer

net — MOP function

Using the specified port, perform some maintenance operations protocol (MOP) function.

The net command performs basic MOP operations, such as, loopback, request IDs, and remote file loads. The net command also provides the means to observe the status of a network port. Specifically, the 'net -s' will display the current status of a port including the contents of the MOP counters. This is useful for monitoring port activities and trying to isolate network failures.

To display the Ethernet station address, enter "net -sa ewa0".

Syntax

net [-s] [-sa] [-ri] [-ic] [-id] [-l0] [-l1] [-rb] [-csr]
[-els] [-kls] [-cm *mode_string*] [-da *node_address*]
[-l *file_name*] [-lw *wait_in_secs*] [-sv *mop_version*]
port_name

Arguments

port_name
Specifies the Ethernet port on which to operate. If you do not specify a port the default port, ewa0, is used.

Options

-s
Display port status information including MOP counters.

-sa
Display the port's Ethernet station address.

-ri
Reinitialize the port drivers.

-ic
Initialize the MOP counters.

-id
Send a MOP Request ID to the specified node. You specify the destination address with -da.

-l0
Send an Ethernet loopback to the specified node. This command, l0, is "l" for loopback and zero. You specify the destination address with -da.

-l1
Do a MOP loopback requester.

-rb
Request to be rebooted by sending a MOP V4 request boot message to a remote boot node. You specify the destination address with -da.

-csr
Displays the values of the Ethernet port CSRs.

-els
Enable the extended design verification test (DVT) loop service.

-kls
Kill the extended DVT loop service.

-cm *mode_string*
Change the mode of the port device. The mode string may be one of the following abbreviations.

- nm = Normal mode
- in = Internal loopback
- ex = External loopback
- nf = Normal filter
- pr = Promiscious
- mc = Multicast
- ip = Internal loopback and promiscious
- fc = Force collisions
- nofc = Do not force collisions
- df = Default

-da *node_address*
Specifies the destination address of a node to be used with the -l0, -id, or -rb options.

-l *file_name*
Broadcast a MOP load request that requests the specified load file.

-lw *wait_in_secs*
Wait the specified number of seconds for the loop messages from the -l1 option to return. If the messages do not return in the time period, an error message is generated.

-sv *mop_version*
Set the preferred MOP version number for operations. Legitimate values are 3 or 4.

Examples

1.
```
>>> net -sa
-ewa0:  08-00-2b-1d-02-91
```

   Displays 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
>>>
```

Displays the ewa0 port status, including the MOP counters.

Related Commands

nettest

nettest — MOP loopback test

This network test can test the Ethernet port in internal loopback, external loopback, or live network loopback mode.

Nettest contains the basic options to allow you to run MOP loopback tests. Nettest is designed to be run from a script file. Many environment variables can be set to customize nettest. You may set these from the console before nettest is started. Listed below are the environment variables, a brief description, and their default values.

- **ewa0_loop_count** Specifies the number (hex) of loop requests to send. The default is 0x3E8 loop packets.
- **ewa0_loop_inc** Specifies the number (hex) of bytes the message size is increased on successive messages. The default is 0xA bytes.
- **ewa0_loop_patt** Specifies the data pattern (hex) for the loop messages. The following are legitimate values.
  - **0** All zeroes
  - **1** All ones
  - **2** All fives
  - **3** All 0xAs
  - **4** Incrementing data
  - **5** Decrementing data
  - **ffffffff** All patterns (default)
- **loop_size** Specifies the size (hex) of the loop message. The default packet size is 0x2E.

You can change other network driver characteristics by modifying the port mode. Refer to the -*mode* option.

Syntax

nettest [-f *file*] [-mode *port_mode*] [-p *pass_count*]
[-sv *mop_version*] [-to *loop_time*] [-w *wait_time*]
[*port*]

Arguments

port
Specifies the Ethernet port on which to run the test.

Options

-f *file*
Specifies the file containing the list of network station addresses to loop messages to. The default file name is lp_nodes_ewa0. The files by default have their own station address.

-mode *port_mode*
Specifies the mode to set the port adapter (TGEC). The default is ex (external loopback). Allowed values are:

df Default, use environment variable values
ex External loopback
in Internal loopback
nm Normal mode
nf Normal filter
pr Promiscious
mc Multicast
ip Internal loopback and promiscious
fc Force collisions
nofc Do not force collisions
nc Do not change mode

-p *pass_count*
Specifies the number of times to run the test. If 0, then run forever. The default is 1. Note that each pass of the test will send the number of loop messages set by the environment variable, ewa0_loop_count.

-sv *mop_version*
Specifies which MOP version protocol to use.

3 Use MOP V3 (DECNET Phase IV) packet format
4 Use MOP V4 (DECNET Phase V IEEE 802.3) format

-to *loop_time*
Specifies the time in seconds allowed for the loop messages to be returned. The default is 2 seconds.

-w *wait_time*
Specifies the time in seconds to wait between passes of the test. The default is 0 (no delay). The network device can be very CPU intensive. This option allows other processes to run.

Examples

1. `>>> nettest ewa0`

   Runs an internal loopback test on port ewa0.

2. `>>> nettest -mode ex`

   Runs an external loopback test on port ewa0.

3. `>>> nettest -mode ex -w 10`

   Runs an external loopback test on port ewa0, waiting 10 seconds between tests.

4. `>>> nettest -f foo -mode nm`

   Runs a normal mode loopback test on port ewa0 using the list of nodes contained in file, foo.

Related Commands

net, netexer

ps — show process

The ps command displays the system state in the form of process status and statistics.

Syntax

ps

Arguments

None.

Options

None.

Examples

1.
```
>>> ps
ID       PCB     Pri CPU 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
>>>
```

Related Commands

sa, sp

pwrup — run powerup diagnostics

This runs the powerup script. It initializes network environment variables, runs memory tests, and executes the contents of the NVRAM script.

Syntax

pwrup

Arguments

None.

Options

None.

Examples

1. `>>> pwrup`

   Runs the powerup script.

rm — remove file

Remove the specified file or files from the file system. Any allocated memory is returned to the heap.

Syntax

rm ***file1*** **[*file2 . . .*]**

Arguments

file1 file2 . . .
Specifies the file or files to be deleted.

Options

None.

Examples

1.
```
>>> ls foo
foo
>>> rm foo
>>> ls foo
foo no such file
>>>
```

   Lists file foo to show that it exists, removes file foo, lists file foo again to show that it is gone.

Related Commands

cat, ls

sa — set process affinity

Changes the affinity mask of a process. The affinity mask of a process specifies on which processors the process may run.

Syntax

sa ***process_id affinity_mask***

Arguments

process_id
Specifies the PID of the process to be modified.

affinity_mask
Specifies the new affinity mask which indicates which processors the process may run on. Bits 0 and 1 of the mask correspond to processors 0 and 1, respectively.

Options

None.

Examples

1.
```
>>> 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
>>>
```

Related Commands

ps, sp

semaphore — show system semaphores

Show all the semaphores known to the system by traversing the semaphore queue.

Syntax

semaphore

Arguments

None.

Options

None.

Examples

1.
```
>>> 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
>>>
```

Related Commands

None

set — set environment variable

Sets or modifies the value of an environment variable (EV). Some of the EVs are stored in non-volatile memory. Environment variables are used to pass configuration information between the console and the operating system. See Chapter 4 for more information about each of the EVs.

Syntax

set ***envar_name value*** **[-default] [-integer] [-string]**

Arguments

envar_name
The environment variable to be assigned a new value. Refer to the list of commonly used environment variables below (or in Chapter 4).

value
The value that is assigned to the environment variable. Either a numeric value or an ASCII string.

Options

-default
Restores an environment variable to its default value.

-integer
Creates an environment variable as an integer.

-string
Creates an environment variable as a string.

Some of the common environment variables are described in the following list:

Commonly Used Environment Variables

auto_action
Sets the console action following an error, halt, or power-up, to *halt*, *boot*, or *restart*. The default setting is *halt*.

bootdef_dev
Sets the default device or device list from which the system attempts to boot. For systems which ship with factory installed software (FIS), the default device is preset at the factory to the device that contains FIS. For systems which do not ship with FIS, the default setting is null.

boot_file
Sets the file name to be used when a bootstrap requires a file name. The default setting is null.

boot_osflags
Sets additional parameters to be passed to system software. The default setting is 0,0.

Examples

1. `>>> set bootdef_dev ewa0`

   Sets the default device from which the system attempts to boot to ewa0.

2. `>>> set auto_action boot`

   Sets the system's default console action to boot after error, halt, or power-up.

3. `>>> set boot_file vax_4000.sys`

   Sets the file name to be used when the system's boot requires a file name to vax_4000.sys.

4. `>>> set boot_osflags 0,1`

   Sets the system's default boot flags to 0,1.

5. `>>> set foo 5`

   Creates environment variable foo and sets its value to 5.

Related Commands

clear, show

set led — display char on LED

Displays a character on the front panel LED.

Syntax

set led ***char*** **[-b]**

Arguments

char
Specifies the character to display on the front panel LED. Metacharacters must be prefixed with a backslash (\).

Options

-b
Specifies that the character be displayed in bright mode. The default is dim mode.

Examples

1. `>>> set led "W" -b`

   Displays an uppercase W on the LED panel at full brightness.

Related Commands

show led

set mode — set the current mode for diagnostics

This command sets the mode setting. The mode values *fastboot* and *nofastboot* control the *level of testing done* at powerup or after console initialization. The *fastboot* value is used when the user wants minimal console diagnostics executed at powerup. Full diagnostics are run when the *nofastboot* value is set.

Note

When *fastboot* mode is enabled, hardware problems may not be detected since the power-on/self testing done during *fastboot* is limited.

Syntax

set mode

Arguments

None.

Options

None.

Examples

1. `>>> set mode nofastboot`

   Sets the mode to nofastboot which runs full diagnostics at poweron or after init.

Related Commands

None

set reboot srom — set reboot mode to Serial ROM Mini-Console

This command is used to enter the Serial ROM Mini-Console.

The only valid (and necessary) argument is *srom*, so that the complete command is *set reboot srom*. Once this command is issued, the Serial ROM Mini-Console is entered on the next system reset or powerup sequence.

Once issued, this command prevents further console boots until NVRAM bytes are altered using the Serial ROM Mini-Console. This is done by using the *wb* Serial ROM Mini-Console command to set either NVRAM location 0x8028 and/or 0x8029 to zero. Upon the next system reset or poweron, the console will be started.

Syntax

set reboot srom

Arguments

None.

Options

None.

Examples

1. `>>> set reboot srom`

   Sets the reboot flag to enter Serial ROM Mini-Console on next reset or poweron.

Related Commands

None

set toy sleep — disable the TOY clock's internal oscillator

This command disables the DS1386 TOY clock's internal oscillator, lengthening the shelf life of the device. When this command is executed bit 8 of the MONTH register of the device is set to 1, disabling the TOY clock's oscillator. The TOY clock's time registers will cease to advance and the life of the device's internal lithium battery will be lengthened. The next time the system is powered up the oscillator will be automatically reenabled by the console and time will once again be counted by the TOY device. This command is to be used by manufacturing at final test or by users who wish to put the system into storage. Note that the time and date will need to be reset once the module is powered up after disabling the battery. See Section 1.13.3 for operating the oscillator on standby power through the VME backplane.

Syntax

set toy sleep

Arguments

None.

Options

None.

Examples

1. `>>> set toy sleep`

   Sets the TOY into storage mode. Automatically re-enabled on subsequent initialisation.

Related Commands

None

sh — create new shell

The shell command creates another shell process. Each shell process implements most of the functionality of the Bourne shell.

Syntax

sh [**-x** | **-v** | **-d**] **[-l] [-r] [-p] [arg . . .]**

Arguments

arg
Any text string terminated with whitespace

Options

-v
Print lines as they are read in.

-x
Show commands just before they are executed.

-d
Delete *stdin* when shell is done.

-l
Trace lexical analyzer (show tokens as they are recognized).

-r
Trace parser (show rules as they fire).

-p
Trace execution engine (show routines called).

Examples

1.
```
>>> sh          # start a new shell
>>>             # the new shell's prompt

>>> 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 [{config, device, hwrpb, led, map, mode, pal, version}] [*envar_name*]

Arguments

config
Displays the system configuration.

device
Displays devices and controllers in the system.

hwrpb
Displays the Alpha HWRPB.

led
Displays character illuminated on the led.

map
Displays system virtual memory map.

mode
Displays current mode, *fastboot* or *nofastboot*

pal
Displays the version of PALcode.

version
Displays the version of the console firmware.

envar
Displays the value of the environment variable specified. Refer to the list of commonly used environment variables below (or in Chapter 4).

Options

None.

Commonly Used Environment Variables

auto_action
Displays the console action following an error halt or power-up. The action can be halt, boot, or restart.

bootdef_dev
Displays the device or device list from which bootstrapping is attempted.

boot_file
Displays the file name to be used when a bootstrap requires a file name.

boot_osflags
Displays the additional parameters to be passed to system software.

language
Displays the language in which system software and layered products are displayed.

Examples

1.
```
>>> show version
version                 V12.0-0 Oct 26 1994 12:58:38
>>>
```

   Displays the version of the firmware on a system. The firmware version is V12.0-0.

2.
```
>>> show auto_action
boot
>>>
```

   Displays the default system powerup action.

3.
```
>>> show bootdef_dev
ewa0
>>>
```

   Displays a system's default boot device, exa0 in this case.

Related Commands

set, show config, show device, show hwrpb, show led, show map, show mode

show config — show system configuration

Shows the system configuration.

Syntax

show config

Arguments

None.

Options

None.

Examples

1.
```
>>> show config

                            Digital Equipment Corporation
                                Digital AXPvme 64LC

        SRM Console X3.7-9224  VMS PALcode X5.48-59, OSF PALcode X1.35-41

                     MEMORY:    32 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           08-00-2B-E2-48-35

        slot 3  Intel SIO 82378

>>>
```

   Displays the system's configuration.

Related Commands

None

show device — displays devices

Shows the devices and controllers in the system. By default all devices and controllers which respond are shown.

The device naming convention is as follows.

```
dka0.0.0.0.0
| || | | | |
| || | | | +--- Hose # : Always zero for AXPvme
| || | | +---- 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
```

The output will display the device name, device ID, device type and device internal firmware revision information (if available).

Syntax

show device [*device_name*]

Arguments

device_name
Specifies the device name or an abbreviation of a device name. When an abbreviation or wildcard is used, all devices that match are shown.

Options

None.

Examples

1.
```
>>> show device
dkc0.0.0.2.0              DKC0                          RZ57
mke0.0.0.4.0              MKE0                         TLZ04
ewa0.0.0.6.0              EWA0            08-00-2B-1D-27-AA
p_a0.7.0.0.0                                       Bus ID 7
p_b0.7.0.1.0                                       Bus ID 7
pkc0.7.0.2.0              PKC0                 SCSI Bus ID 7
pke0.7.0.4.0              PKE0                 SCSI Bus ID 7
pud0.7.0.3.0              PID0                 DSSI Bus ID 7
>>>
```

   Shows all devices and controllers in the system. Note that the controllers p_a0 and p_b0 are indeterminant, that is, neither SCSI nor DSSI. This occurs when no devices or terminators are present.

2.
```
>>> show device e
ewa0.0.0.6.0              EWA0            08-00-2B-1D-27-AA
```

   Show devices that start with e.

3.
```
>>> show device *k*          # Show SCSI devices.
dkc0.0.0.2.0               DKC0                            RZ57
mke0.0.0.4.0               MKE0                           TLZ04
```

   Show all devices with k in the device name.

4.
```
>>> show device dk           # Show SCSI disks.
dkc0.0.0.2.0               DKC0                            RZ57
```

   Show all devices starting with dk (all SCSI disks).

5.
```
>>> show device mk           # Show SCSI tape drives.
mke0.0.0.4.0               MKE0                           TLZ04
>>>
```

   Show all devices starting with mk (all SCSI tapes).

Related Commands

None

show hwrpb — display HWRPB

Display the address of the Alpha HWRPB.

Syntax

show hwrpb

Arguments

None.

Options

None.

Examples

1.
```
>>> show hwrpb
HWRPB is at 2000
>>>
```

show led — show LED character

This command will show the current character being displayed by the front panel LED.

Syntax

show led [-hex]

Arguments

None.

Options

-hex
Display the contents of the LED register. If you do not specify -hex, the character being displayed is echoed to the console.

Examples

1. `>>> show led`

   Show the current character being displayed.

2. `>>> show led -hex`

   Show the contents of the LED register.

Related Commands

set led

show map — display memory map

Display the current system virtual memory map.

Note

The map will be empty after all console initialization. By typing "boot -halt" at the console prompt, the Page Table Entries will be filled in.

Syntax

show map

Arguments

None.

Options

None.

Examples

1.
```
>>> 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
>>>
```

show mode — display the current powerup mode for diagnostics

This command displays the mode setting. The mode values *fastboot* and *nofastboot* control the level of testing done at powerup or console initialization. The *fastboot* value is displayed when the console executes minimal diagnostics at powerup. Full diagnostics are run when the *nofastboot* value is set.

Syntax

show mode

Arguments

None.

Options

None.

Examples

1.
```
>>> show mode
Console is in fastboot mode
>>>
```

   Displays the mode setting.

Related Commands

None

show_log — display error log information from NVRAM

This command is used to display console-detected fault information that was previously stored in the Error Log area of NVRAM.

With no command-line qualifiers or options, the most-recent fault is displayed.

Note that console error logging is completely independent of the OS error logging.

Syntax

show_log [**-n [count]** | **-all** | **-new**]

Arguments

None.

Options

-n *count*
Displays *count* most-recent faults logged into the NVRAM Error Log area. *Count* defaults to 1 if it is omitted.

-all
Displays *all* faults logged into the NVRAM Error Log area. *All* faults are *marked* as *seen* so that new faults can be easily displayed using the *-new* Option. This command always displays *all* logged faults.

-new
Displays all *new* faults logged into the NVRAM Error Log area; displays all faults that have *not* been previously displayed by the *show_log -all* command.

Examples

1.
```
>>> show_log
============================== F A U L T #1 ==============================

Time of Error: 13:08:39  9-AUG-1994
Diagnostic   : Interval Timer
Pass Count   : 1     Test Number: 4     Failing Point: 18
Error Message: Interrupt not invoked and should have been
>>>
```

   By default, the most-recent fault is displayed.

2.
```
>>> show_log -n 3
============================== F A U L T #1 ==============================

Time of Error: 13:10:06  9-AUG-1994
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-1994
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

================================================================================

>>>
```

Displays the 2 most-recent faults since they are the only ones logged into NVRAM.

Related Commands

clear_log

sleep — suspend execution

The sleep command suspends execution of a console process for a specified number of seconds. It temporarily wakes up every second to check for and kill pending bits.

Syntax

sleep [-v] ***time_in_secs***

Arguments

time_in_secs
Specifies the number of seconds to sleep. The default is 1 second.

Options

-v
Specifies that the value supplied is in milliseconds. The default is 1000 (1 second).

Examples

1.
```
>>> (sleep 10; echo hi there)&
>>>
(10 seconds expire...)
hi there
```

   Sleep for 10 seconds then execute next command (echo).

2.
```
>>> sleep -v 20
```

   Sleep for 20 milliseconds.

Related Commands

None

sort — sort a file

Sort the lines of a file in lexicographic order and write the results to *stdout*. The size of the file that sort can handle is limited by the size of memory.

Syntax

sort ***file***

Arguments

file
Specifies the file to be sorted.

Options

None.

Examples

1.
```
>>> echo > foo 'banana
_>pear
_>apple
_>orange'
```

   Create file, foo, with 4 lines.

2.
```
>>> sort foo
apple
banana
orange
pear
```

   Sort file, foo, and send output to the console.

sp — set priority

Modifies the priority of a process. Changing the priority of process will impact the behavior of the process and the rest of the system.

Syntax

sp ***process_id new_priority***

Arguments

process_id
Specifies the PID of the process to be modified.

new_priority
Specifies the new priority for the process. Priority values range from 0 to 7 where 7 is the highest.

Options

None.

Examples

1.
```
>>> 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
>>>
```

Raises the priority of process 25 from 2 to 3.

Related Commands

ps, sa

start — start program

Starts program execution at the specified address or starts drivers.

Syntax

start [-drivers [*device_prefix*]] [*address*]

Arguments

address
Specifies the PC address at which to start execution.

Options

-drivers [*device_prefix*]
Specifies the name of the device or device class to stop. If no device prefix is specified, then all drivers are started.

Examples

1. `>>> start 400`

   Start program execution at address 400.

2. `>>> start -drivers`

   Start all the drivers in the system.

Related Commands

continue, init, stop

stop — stop CPU or device

Stops the processor or the specified device.

Syntax

stop [-drivers [*device_prefix*]] [*processor_num*]

Arguments

processor_num
Specifies the processor to stop. If included, must be 0.

Options

-drivers [*device_prefix*]
Specifies the name of the device or the device class to stop. If no device prefix is specified, then all drivers are stopped.

Examples

1. `>>> stop`

   Stops the processor.

Related Commands

continue, init, start

update — update flash ROMs on the system

Loads new firmware into the flash ROMs (FEPROMs). In order to modify the flash ROMs, DIP switch #2 on the AXPvme module must be *closed*. Refer to the Section 1.1.3.15 for the location of DIP switches.

The update process proceeds as follows:

1. The image is loaded from the specified *device* into system memory.
2. If the *target* specified is *console*, consistency checks are applied to the loaded image to ensure a valid console has been loaded. If the *target* specified is *userflash*, no checks are performed on the loaded image.
3. Once a valid image is loaded into memory, the user is prompted to confirm continuation of the update.
4. The FEPROMS are then re-programmed.

There are three steps to the programming process:

1. All Flash ROM bytes are programmed to '00'.
2. All Flash ROM bytes are then erased. The erased state is 'FF'.
3. All Flash ROM bytes are reprogrammed to the values in the image loaded into memory.

Each byte of the FEPROM is verified in each of the three steps. Each step provides for a certain number of re-tries 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 printed to the console.

If the programming operation is successful, a success message is printed on the console.

Note that you must reset or cycle power on the system to run the new image in the FEPROMs. Until so, the previous console image is executing out of memory.

Note

Be sure to disable the FEPROM writing after completing the update process by setting switch 2 to the *open* position.

Syntax

update [-file *filename*] [-protocol *transport*] [-device *source_device*] [-target *target_name*]

Arguments

None.

Options

-file *filename*
Specifies the name of the new FEPROM update image.

-protocol *transport*
Specifies the source transport protocol Valid protocols are *MOP* and *TFTP*. See the *BOOT* command for additional information on using the tftp protocol.

-device
Specifies the device from which to load the new FEPROM update image file from. Currently, the only valid device is *ewa0*.

-target *device*
Specifies the device which contains the FEPROMs to be upgraded. Valid targets are *console* and *userflash*.

Examples

1.
```
>>> update -file bl12 -device ewa0 -protocol mop -target console
                        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 OPERABLE STATE.

The program will take at most several minutes.

Programming flash device at pcimem:0, with image at pmem:f0020
...
Programming flash device at pcimem:40000, with image at pmem:130020
...
Verifying...

Update successful
>>>
```

The example above shows how to do an update using the MOP protocol.

2.
```
>>> update -fi //usr//local//bootfiles//bl12 -dev ewa0 -tar console -pro tftp
update -path tftp://usr//local//bootfiles//bl12/ewa0 -target console

                        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 OPERABLE STATE.

The program will take at most several minutes.

Programming flash device at pcimem:0, with image at pmem:f0020
...
Programming flash device at pcimem:40000, with image at pmem:130020
...
Verifying...

Update successful
>>>
```

The example above shows how to do an update using the TFTP protocol. Note that the *ewa0_bootp_server* environment variable must be set to the Internet address of the server.

4
Environment Variables

4.1 Overview

Environment variables provide a simple extensible mechanism for managing complex states. Such states may be variable length, may change with system software, may change as a result of console state changes, and may be established by the console presentation layer. Environment variables may be read, written, or saved.

Environment variables consist of an identifier (ID) and a byte stream value maintained by the console. The *Alpha Architecture Reference Manual* (ARM) defines three classes of environment variables:

- IDs 0-3F

  Common to all implementations

- IDs 40-7F

  Specific to a given console implementation

- IDs 80-FF

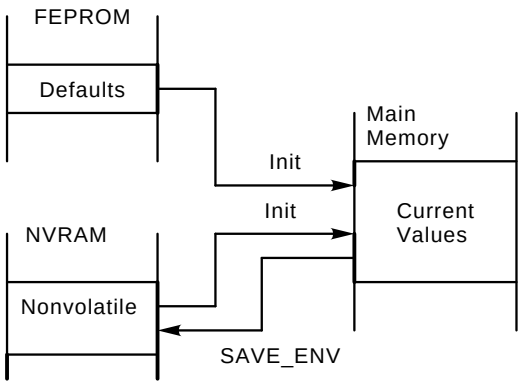
  Specific to system software

The value, format, and size of environment variables is dependent upon the environment variable and on the console implementation. The size is specified in bytes and the value consists of an ASCII string or a hexadecimal value. Operating system software uses console-provided callback routines to access environment variables. Each environment variable ID is resolved and the value of the associated environment variable is returned.

Environment variables can be in a number of locations as indicated by Figure 4–1. Default values for the environment variables are stored in flash EPROM. These default values are loaded when the console firmware is copied into main memory during initialization. Values for nonvolatile environment variables are stored in NVRAM, and will also be copied over during initialization. An NVRAM copy of any particular environment variable takes precedence over the flash EPROM copy.

Environment variables can be changed using the console set command (see Chapter 3). This command will change the copies in main memory and in NVRAM. Operating system software can manipulate environment variables by using the GET_ENV and SET_ENV callback routines, which manipulate the copies in main memory. SAVE_ENV will copy the current values in memory, which will be copied to NVRAM. The environment variables in NVRAM are protected by a checksum, which is recalculated every time a variable is changed. If a problem is detected in the checksum, the console reverts to using default values.

Figure 4–1 Storage Locations of Environment Variables



4.2 Application-Independent Environment Variables

The Alpha ARM defines a list of environment variables that are common to all implementations and must be supported by the console (see Table 4–1). These environment variables can be accessed from the console using the SET and SHOW commands and can be accessed from system software using the SET_ENV and GET_ENV callback routines. The ID for this class of environment variable is between 0 and 3F (hexadecimal).

Table 4–1 ARM Defined Environment Variables

Name	Meaning
auto_action	Defines console action following an error, halt, or powerup. Defined values are BOOT, HALT, and RESTART. The default value is HALT.
boot_dev	Device list 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 device list is specified, bootdef_dev. The console uses boot_dev without change on all bootstrap attempts that are not initiated by a BOOT command.
bootdef_dev	Device list from which bootstrapping is to be attempted when no path is specified by a BOOT command.
booted_dev	Devices used by the last or currently in-progress bootstrap attempt. Value is one of the devices in the boot_dev list.
boot_file	File name to be used when a bootstrap requires a file name and when bootstrap is not the result of a BOOT command or when no file name is specified on a BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
booted_file	File name used by the last or currently in-progress bootstrap attempt. The value is derived from boot_file or the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.

(continued on next page)

Table 4–1 (Cont.) ARM Defined Environment Variables

Name	Meaning
boot_osflags	Additional parameters to be passed to system software when the bootstrap is not the result of a BOOT command or when none is specified on a BOOT command. The console passes the value between the console presentation layer and system software without interpretation. The following parameters are used with the DEC OSF/1 operating system (the default is NULL): a Autoboot. Boots /vmunix from bootdef_dev, goes to multiuser mode. Use this for a system that should come up automatically after a power failure. s Stop in single-user mode. Boots /vmunix to single-user mode and stops at the #(root) prompt. i Interactive boot. Request the name of the image to boot from the specified boot device. Other flags such as -kdebug (to enable the kernel debugger) may be entered using this option. D Full dump implies "s" as well. By default, if DEC OSF/1 V2.1 fails, it completes a partial memory dump. Specifying "D" forces a full dump at system failure.
booted_osflags	Additional parameters passed to system software during the last or currently in-progress bootstrap attempt. The value is derived from boot_osflags or the current BOOT command. The console passes the value between the console presentation layer and system software without interpretation.
boot_reset	Indicates whether a full system reset is performed in response to an error halt or boot. Defined values are ON and OFF. The default value is OFF.
dump_dev	Device to write operating system crash dumps.
enable_audit	Indicates whether audit trail messages are to be generated during bootstrap. Defined values are ON and OFF. The default value is ON.
license	Software license in effect. Define values are MU—multi-user system and SU—single-user system.
char_set	Current console terminal character-set encoding. The default value is 0, ISO-LATIN_1. Other char_sets have yet to be defined in the ARM.

(continued on next page)

Table 4–1 (Cont.) ARM Defined Environment Variables

Name	Meaning
language	Current console terminal language (integer ID). The defined values are: 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
language_name	ASCII string of the current console terminal language code defined in the *language* environment variable.
tty_dev	Current console terminal unit. Indicates which entry of the CTB Table corresponds to the actual console terminal. The default value is 0 (30 hex).

4.3 Diagnostic Environment Variables

Table 4–2 lists the diagnostic-related environment variables that are not required by the Alpha ARM, but which are implemented on the AXPvme platform.

Note

These values are *volatile* in that they will not be preserved across a reset of any kind.

Table 4–2 Console Diagnostic Environment Variables

Name	Meaning
d_bell	Bell on error. The default is OFF.
d_cleanup	Cleanup code executed at diagnostic end. The default value is ON.
d_complete	Display diagnostic completion message. The default value is OFF.
d_eop	Display end-of-pass messages. The default is OFF (disable).
d_group	Diagnostic group to be executed. Defined values are FIELD, MFG, or other (up to 32 characters). The default value is FIELD.

(continued on next page)

Table 4–2 (Cont.) Console Diagnostic Environment Variables

Name	Meaning
d_harderr	Action following a hard error detection. The defined values are CONTINUE, HALT, and LOOP. The default value is HALT.
d_oper	Operator present. The default is OFF (no operator).
d_passes	Diagnostic pass count. The defined values are 0 (run indefinitely) or a user-defined value. The default is 1 pass.
d_report	Level of information provided by the diagnostic error reports. The defined values are SUMMARY, FULL, and OFF. The default value is FULL.
d_softerr	Action taken following a soft error detection. The defined values are CONTINUE, HALT, and LOOP. The default value is CONTINUE.
d_startup	Display diagnostic startup message. The default value is OFF (disable).
d_trace	Display trace messages. The default value is OFF (disable).

4.4 Console-Specific Environment Variables

The following environment variables are specific to the AXPvme console.

4.4.1 Ethernet Environment Variables

Table 4–3 Ethernet Configuration Environment Variables

Name	Meaning
ewa0_arp_tries	Number of transmissions that are attempted before the ARP protocol 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.
ewa0_bootp_file	Generic file name to be included in a BOOTP request. The BOOTP server will return a fully qualified file name for booting. There is no specified default file name.
ewa0_bootp_server	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.
ewa0_bootp_tries	Number of transmissions that are attempted before the BOOTP protocol 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.
ewa0_def_ginetaddr	Initial value for ewa0_ginetaddr when the interface's internal internet database is initialized from BOOTP (that is, ewa0_inet_ init is set to "bootp").
ewa0_def_inetaddr	Initial value for ewa0_inetaddr when the interface's internal internet database is initialized from BOOTP (that is, ewa0_inet_ init is set to "bootp").
ewa0_def_inetfile	Initial value for ewa0_inetfile when the interface's internal internet database is initialized from BOOTP (that is, ewa0_ inet_init is set to "bootp").

(continued on next page)

Table 4–3 (Cont.) Ethernet Configuration Environment Variables

Name	Meaning
ewa0_def_sinetaddr	Initial value for ewa0_sinetaddr when the interface's internal internet database is initialized from BOOTP (that is, ewa0_inet_ init is set to "bootp").
ewa0_inet_init	Determines whether the interface's internal internet database is initialized from NVRAM or from a network server (via the BOOTP protocol). Defined values are NVRAM and the default bootp.
ewa0_loop_count	Number of times each message is looped. The default value is "0x3e8".
ewa0_loop_inc	Amount that the message size is increased from message to message. The default value is "0xa".
ewa0_loop_patt	Type of data pattern to be used when doing loopback. Current patterns are accessed by the following: 0xffffffff = All the patterns (default) 0 = All zeros 1 = All ones 2 = All fives 3 = All As 4 = Incrementing 5 = Decrementing
ewa0_loop_size	Size of the loop data to be used. The default value is "0x2e".
ewa0_lp_msg_node	Number of messages originally sent to each node. The default value is "7".
ewa0_protocols	Network protocol enabled for booting and other functions. Defined values are "bootp", "mop" (default), and "bootp,mop". A null value is equivalent to "bootp,mop".
ewa0_tftp_tries	Number of transmissions that are attempted before the TFTP protocol 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.

4.4.2 Storage Environment Variables

Table 4–4 Storage Configuration Environment Variables

Name	Meaning
ncr*_setup	Defined values for "*" are 0, 1, 2, 3, or 4, corresponding to the storage bus adapters A, B, C, D, or E, respectively.

4.4.3 Console Configuration Environment Variables

Table 4–5 Console Configuration Environment Variables

Name	Meaning
pal	Versions of OpenVMS and OSF PALcode in the firmware.
pci_arb_mode	PCI arbitration mode. This environment variable sets the PCI arbitration scheme for the module. The value set in this environment variable is written to the Module Configuration Register when the value is set and during console initialization. The values 0, 1, 2 and 3 are valid. The list below shows the PCI arbitration priority assignment. The following acronyms are used in this list: LCA—LCA processor, NI—Network Interface, OPT—PCI mezzanine option, VME—PCI to VMEbus interface, and SCSI—Small Computer System Interface. 0 = LCA, NI, SCSI, OPT, VME 1 = OPT, LCA, VME, NI, SCSI 2 = VME, LCA, OPT, NI, SCSI 3 = VME, LCA, OPT, NI, SCSI
pci_park_dev	PCI park device. This environment variable sets the PCI parking scheme for the module. The value set in this environment variable is written to the Module Configuration Register when the value is set and during console initialization. The values 0, 1, 2, and 3 are valid. The list below shows the park device selection. 0 = Park with LCA processor 1 = Park with Network Interface 2 = Park with PCI mezzanine option 3 = Park with VME
sys_serial_num	The system serial number set by manufacturing.
tt_baud	This environment variable is used to change the baud rate of the UARTs; the console UART (channel A) and the auxiliary UART (channel B) are set to the same value. The baud rate field of the Console Terminal Block (CTB, a portion of the HWRPB) is updated, too. Once the baud rate is changed using the *set tt_baud* command, the console terminal's baud rate must then be updated to the new baud rate setting. Valid values for this environment variable are: 300, 600, 1200, 2400, 4800, 9600, and 19200.
version	Version of the console code firmware.
vme_config	VME setup mode. This environment variable is used by the operating systems (VxWorks and OSF) for storing VME configuration information for the initialization of the VME corner. Refer to the VxWorks and OSF technical documentation for more information.
vme_a32_base	Base address of VMEbus A32 space.
vme_a32_size	Size of A32 VMEbus address space.
vme_a24_base	Base address of VMEbus A24 space.
vme_a24_size	Size of A24 VMEbus address space.
vme_a16_base	Base address of VMEbus A16 space.
vx_bootline	File name used for VxWorks bootstrap.

5
Diagnostics

This chapter describes the built-in ROM-based diagnostics. These comprise both power-up self-test diagnostics and extended diagnostics.

5.1 Power-up Self-Test

When you turn on the power or press the reset switch, the AXPvme module runs its power-up self-test (POST). The module first runs a series of tests stored in the serial ROM (SROM). This code is loaded directly into the instruction cache on the processor chip over an internal serial line. The code is then executed from the instruction cache. The SROM tests display their test number on the LED display.

The module then runs a series of console code tests that are stored in flash ROM. The code for these tests is loaded into main memory and executed. These tests display their test names and results on the console terminal.

5.2 Miscellaneous Diagnostic Hooks

A test pattern of 4 bytes is present at the end of the console image stored in the flash devices. Table 5–1 details the address and corresponding byte contents. Accessing these test patterns verifies many hardware items; including PCI bus, Intel SIO setup, and the flash device.

Table 5–1 Test Patterns in Flash

PCI Memory Address	Byte Value
0x7FFDC	0xFF
0x7FFDD	0x00
0x7FFDE	0xAA
0x7FFDF	0x55

5.3 Diagnostic Test Descriptions

5.3.1 Available Console and SROM Diagnostics

Table 5–2 shows all the Console and SROM diagnostic tests and the commands used to invoke them. The majority of these tests can be user invoked at the console prompt.

Table 5–2 Console Diagnostic Tests

HW Under Test	Command
Flash ROMs	
- Flash ROM Test	flash_diag -g mfg -t 1
- Flash ROM Test	flash_diag -g mfg -t 2*
- Flash ROM Test	flash_diag -g mfg -t 3*
- Flash ROM Test	flash_diag -g mfg -t 4*
- Flash ROM Test	flash_diag -g mfg -t 5*
* : Erases and Restores Flash	
Memory and Cache	
- Bcache Diagnostic Exerciser	memecc_diag
- Memory ECC/DETECT Test	bcache_diag
- Memory Exerciser Test	memtest or mem_ex
- Memory POST Test	memdiag
Network Interface	
- DECchip 21040 NI Internal LPBCK	niil_diag -t 1
- DECchip 21040 NI External LPBCK	niil_diag -t 2
- DECchip 21040 NI CSR Test	nicsr_diag -t 1
- DECchip 21040 NI CSR Test	nicsr_diag -t 2
- DECchip 21040 NI CSR Test	nicsr_diag -t 3
NVRAM + TOY	
- NVRAM Testing	ds1386_diag -t 1
- NVRAM Testing	ds1386_diag -t 2
- NVRAM Testing	ds1386_diag -t 3
- TOY Register Testing	ds1386_diag -t 4
- TOY Register Testing	ds1386_diag -t 5
PCI Bus	
- Module CNFG Reg	modcnfg_diag -t 1
- Module CTRL Reg	modctrl_diag -t 1
- Module Display CTRL Reg	display_diag -t 1
- Module Display CTRL Reg	display_diag -t 2
- Module Reset Reason Reg	wdog_diag -t 1
- PCI Option Sizer (Mezzanine) Test	
SCSI	
- SCSI Device Test	ncr810 -t 1
- SCSI Device Test	ncr810 -t 2

(continued on next page)

Table 5–2 (Cont.) Console Diagnostic Tests

HW Under Test	Command
- SCSI Device Test	ncr810 -t 3
- SCSI Device Test	ncr810 -t 4
- SCSI Device Test	ncr810 -t 5
- SCSI Device Test	ncr810 -t 6
- SCSI Device Test	ncr810 -t 7
- SCSI Device Exer	exer dk
Timers	
- Heartbeat Timer	hbeat_diag -t 1
- Interval Timer	i8254 -t 1
- Interval Timer	i8254 -t 2
- Interval Timer	i8254 -t 3*
- Interval Timer	i8254 -t 4*
- Interval Timer	i8254 -t 5
- Interval Timer	i8254 -t 6
- Watchdog Timer	wdog_diag -t 1
* : Requires external loopback connector configured as shown in Figure 5–2.	
UART	
- Internal Loopback Test	z8530 -t 1
- External Loopback Test	z8530 -t 2*
- Channel B Interrupt Test	z8530 -t 3
* : Requires external loopback connector (spoon)	
VME Interface Tests	
- VIP PCI Configuration Register Test	vip_diag -t 1
- VIP Register Write/Read Test	vip_diag -t 2
- VIC Register Write/Read Test	vip_diag -t 3
- Scatter/Gather RAM Test	vip_diag -t 4
- VIP/VIC Local Interrupt Test #1	vip_diag -t 5
- VIP/VIC Local Interrupt Test #2	vip_diag -t 6
- Miscellaneous VIP BESR bits Test	vip_diag -t 10
SROM Tests	
- Console UART Test	
- Dynamic RAM Test	

(continued on next page)

Table 5–2 (Cont.) Console Diagnostic Tests

HW Under Test	Command
- External Backup Cache Test	
- Flash ROM Unload Test	
- Internal Data Cache Test	
- SIO I/O Bus Test	
MISC	
- Enet Hardware Addr Test	enet_diag -t 1
- Enet Hardware Addr Test	enet_diag -t 2

5.3.2 SROM Diagnostic Test Descriptions

This section details the tests that are executed by the SROM during the system initialization testing.

SROM System I/O Device Test

The Intel 82378 System I/O (SIO) device provides a bridge from the PCI bus to the 8-bit ISA I/O bus. This diagnostic first confirms the presence of the SIO device on the PCI bus. It initializes the SIO to allow PCI-to-I/O bus transactions.

Description

After the device's PCI identification (0x0484) has been verified by reading it from PCI configuration space, the I/O Clock Divisor register is initialized to produce an 8.33 MHz I/O clock from the PCI clock frequency. The I/O Controller Recovery Timer register is initialized to set the number of clock cycles between back-to-back I/O bus cycles.

LED Display Test Number: 8

SROM Console UART Test

This test checks the accessibility of the 8530 console Universal Asynchronous Receive/Transmit (UART) chip by writing and reading an internal register.

Description

LED Display Test Number: 7

Miscellaneous Notes

1. It is assumed that PALcode will properly initialize the UART before the console requires it.

SROM Internal Data Cache Test

This test verifies the 8 KB internal data cache on the CPU chip by "forcing hits" to the cache.

Description

The Data Cache Force Hit bit in the Abox_Ctl internal CPU register is set, forcing all data-stream references to hit in the data cache. Several data patterns are written, read, and verified from the 8 KB data cache. The Data Cache Force Hit bit is then disabled.

LED Display Test Number: 6

Miscellaneous Notes

All accesses to data cache are quadword operations.

SROM Dynamic RAM Test

This diagnostic checks the operation of the first 8 MB of Dynamic RAM (DRAM) memory into which the console code will be loaded.

Description

This test first writes the quadword address of each memory location to itself. The address-on-address pattern is then verified. A test of pattern of all A's is then written and verified; followed by a test pattern of all 5's. In all test cases, only the first 8 MB's of memory is tested.

This test covers stuck-at-bits, adjacent bit interactions, and address lines stuck at 0 or 1.

LED Display Test Number: 5

Miscellaneous Notes

1. All accesses to memory are quadword operations.
2. It is assumed that the internal data cache and the external backup cache have been disabled prior to this test executing at power-on or reset.
3. Since 8 MB of memory is first written, followed by reads, the memory refresh circuitry is implicitly tested.

SROM External Backup Cache Test

This test does minimal verification of the interaction of external backup cache and system DRAM. The console test provides additional diagnostics and exercisers.

Description

Due to code size constraints of the SROM, this test verifies the operation of the backup cache and system DRAM by doing only a minimum number of reads and writes to memory.

The test writes a data pattern to the first 512 KB of memory through the external backup cache. The same locations are then read and the data is compared.

Writing and reading 512 KB of memory provides some degree of confidence that Backup Cache is functioning properly (that is, cache block allocation, victim data writes to DRAM, and so forth are working).

LED Display Test Number: 3

Miscellaneous Notes

1. This test assumes that internal data cache has been enabled prior to invoking this test.
2. This test is invoked only when backup cache is present.

SROM Flash EPROM Unload Test

This diagnostic reads the console code from flash EPROMs and stores it in DRAM. A checksum of the data read from flash EPROMs is computed. If the computed checksum compares correctly, execution is transferred to the console code now loaded in DRAM.

Description

LED Display Test Number: 1

Miscellaneous Notes

Since the flash EPROMs are on the 8-bit I/O bus, flash EPROM reads are byte operations.

5.3.3 Console Power On Self Test Descriptions

This section details the power on self tests (POST) which are run during system initialization.

POST NVRAM Diagnostic

This test verifies the module's NVRAM. It performs a data integrity test, through power cycles, and a write/read/compare of specific NVRAM locations used for diagnostics. It also checks for uninitialized NVRAM by verifying the stored checksum with the calculated.

Description

This test is executed at the beginning of console boot before the console drivers and devices have been initialized.

Test Name: None; executes on powerup

POST Memory Diagnostic

This test verifies the 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

See also *memtest* in Chapter 3.

Note

IMPORTANT: This test is dependent upon the console "mode" flag. Setting mode to *fastboot* will evoke a quick verify test of the memory, and *nofastboot* will evoke a full test of memory.

This test is executed at the beginning of console boot before the console drivers and devices have been initialized.

The test also sets up a memory error bitmap in the Console area tested by SROM. Each bit in the bitmap represents an 8 KB page of memory. A "1" bit means that the page of memory has no errors. A "0" bit means that the page was not tested, or had one or more ECC correctable errors, or had one or more non-correctable errors.

This test provides the following coverage:

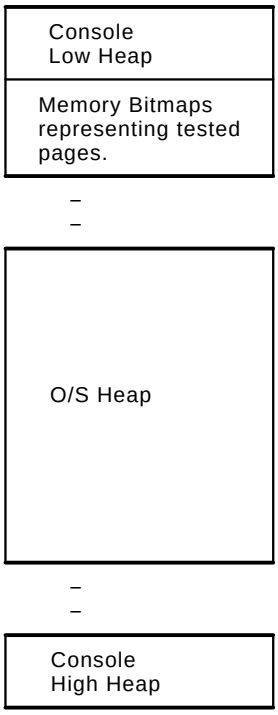
- Memory bits: Stuck bits, bit transition fault, or bit coupling fault
- Decoder logic: An address selects no memory, two or more addresses select the same memory cell, or one address selects more than one cell.
- Sense amplifier logic: Stuck fault or coupling fault.
- Component and path coverage: The 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. Note that the daughter card is assumed good since it is separately tested in manufacturing.

Test Name: None; executes on powerup.

Miscellaneous Notes

1. ECC is verified in hardware and correctable faults raise an exception but are dismissed by the PALcode ECC exception handler. The memory test will fail on data miscompare, non-existent memory, uncorrected ECC to memory, or backup cache.
2. Test prerequisite: The ECC must be properly initialized by the console.
3. Powerup test only. For additional memory tests see *memtest* in Chapter 3.

Figure 5–1 Memory Regions



5.3.4 Console Diagnostic Test Descriptions

This section details the tests that are available to the console which might be run during system initialization testing or could be run from the console.

Flash EPROM Tests

These diagnostics test the four flash EPROMs, decoders, the programmed console image, and printed circuit board module etch.

Flash Console Image Verification Test

This diagnostic verifies the console image programmed into the first two flash EPROMs. First, it verifies the flash *signature* in the second flash EPROM then it computes the checksum and compares it to the stored checksum. Isolation of the failing flash EPROM cannot be determined from this diagnostic.

Console Command: flash_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. The flash EPROM programming power does not have to be enabled for this test.
2. This test is intended for power-up self-test.

Individual Flash Device Tests

These diagnostics test all locations of a specified flash device (256 KB) for stuck bits including a simple address line test pattern.

Command	Flash Device Tested	PCI Memory Address Space
flash_diag -t 2	0	0x00000 - 0x3ffff
flash_diag -t 3	1	0x40000 - 0x7ffff
flash_diag -t 4	2	0x80000 - 0xcffff
flash_diag -t 5	3	0xc0000 - 0xfffff

Note

If power is lost during testing or the test aborted the original contents of the device will be lost. This may lead to loss of operable state of the module if the console image in flash is compromised.

Console Command: flash_diag -t 2,3,4,5

Command line parameters:

- -dd: print detailed test information on each pass.
- -qv: quick verify, only test pattern of all 1's is used on tests 2,3,4,5

Miscellaneous Notes

1. The flash EPROM programming power must be enabled for this test (DIP switch #2 = Closed)

2. The flash parts are spec'ed for 10,000 writes of the device. Each pass of this test writes the device 6 times, so extended use of this test may degrade the reliability of these parts.
3. This is intended to be an extended diagnostic.

Module Display Control Register/LED Tests

These diagnostics test the integrity of the Module Display Control Register, the LED display, decoders, and printed circuit board module etch. The Module Display register is read/write.

Module Display Register Data Line Test

This diagnostic tests the data lines between the Module Display Control Register and the LED. *This test is a visual test and requires operator interaction.* The test displays certain characters chosen to use each of the data lines and LED segments. The operator must type the character on the LED display. The test finishes by illuminating all LED segments in normal luminescence, then bright.

The characters chosen are as follows:

This Display . . .	Tests this Hardware . . .
U	Data lines 6, 4, 2, and 0 for stuck low, and 5, 3, and 1 for stuck high
*	Data lines 6, 4, 2, and 0 for stuck high, 5, 3, and 1 for stuck low
All segments normal, bright, then blank	Data line 7 and LED segments.

Console Command: display_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. This test is intended to be an extended test.
2. This test requires operator assistance.

Module Display Register Test

This diagnostic performs bit pattern tests on the Module Display Control register to look for stuck bits.

Console Command: display_diag -t 2

Command line parameters:

- -dd: print detailed test information on each pass.
- -qv: quick verify, only test pattern of all 1's is used on tests 2,3,4,5

Miscellaneous Notes

1. This test is intended to be a power-up self-test.

Module Control Register Test

This diagnostic tests the integrity of the Module Control Register, decoders and printed circuit board module etch. This register is read/writable.

Module Control Register Test

This diagnostic writes, reads and verifies several bit patterns to/from the Module Control Register in trying to determine stuck bits.

Console Command: modctrl_diag -t 1

Command line parameters:

- none

Miscellaneous Notes

1. This test is intended to be a power-up self-test.

Heartbeat Timer Test

This diagnostic verifies that a heartbeat interrupt is generated at the correct interval (1024 Hz) and is properly dismissed via the Module Clear Heartbeat Register.

This test checks the following logic:

- Heartbeat timer and interrupt delivery mechanism
- Module Clear Heartbeat Register

Heartbeat Timer Test

Console Command: hbeat_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. This is intended to be a power-up self-test diagnostic.
2. The test expects timer interrupts to be enabled. If they are not enabled an interrupt count of zero will result.
3. This test can not be run concurrently with other tests.

Interval Timer Tests

These diagnostics 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.

Since all three interval timers of the 8254 chip have different external configurations, several tests are required for complete test coverage.

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.

This test will require that *both* Timer 0 and 1 be properly programmed and externally connected for successful operation.

Timer 2 Terminal Count Test

This test exercises Timer 2 with the timer interrupts enabled. In the AXPvme design, 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, provided the Timer 2 Interrupt Enable bit is set.

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 Command: i8254_diag -t 1

Command line parameters:

- none

Miscellaneous Notes

1. 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. Bit 4 is toggled by writing to address 0x4010; bit 5 is toggled by writing to address 0x4014. In both cases, the data written is Don't Care.
2. A read of the Interrupt Status Register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
3. Due to hardware limitations on interrupt detection, the value programmed into Timer 2 must be greater than 2.
4. See the Intel 8254 Interval Timer sheet for more details.

Timer 2 Square Wave Test

This test exercises Timer 2. In the AXPvme design, 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 will produce 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 interrupt service routine (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

Command line parameters:

- none

Miscellaneous Notes

1. 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. Bit 4 is toggled by writing to address 0x4010; bit 5 is toggled by writing to address 0x4014. In both cases, the data written is Don't Care.
2. A read of the Interrupt Status Register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
3. Due to hardware limitations on interrupt detection, the value programmed into Timer 2 must be greater than 2.
4. See the Intel 8254 Interval Timer sheet for more details.

3 Timers Loopback Test

This test exercises Timer 2, Timer 1, and Timer 0. In the AXPvme design, 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 which the outputs of Timers 1 and 2 are tied to. Timer 2 is the Gate input and Timer 1 provides the Clock.

This test essentially emulates the OSF Realtime Time Provider and Slave scheme found in the Real Time Clock and Interval Device Driver functional specification.

Note

IMPORTANT: A VMEbus P2 loopback connector is required. See Figure 5–2, for a description of the loopback connections.

Note that using the *-lp* qualifier enables the timers indefinitely, making the module 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 the timers are initially programmed with the appropriate mode and then loaded with a count value, the OUT output will produce a continuous, square wave output whose period is 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 will control 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 interrupt service routine (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 that one occurs per Major Clock cycle.

Console Command: i8254_diag -t 3

Command line parameters:

- -np: no print qualifier; if specified no P2 connector message is printed
- -lp: prevents timers from being stopped at the end of the test; required before invoking Test #4.

Timer 0 Loopback Test

This test exercises only Timer 0. Timer 0 accepts its Clock and Gate input from the P2 loopback connector from Test 3 set up in a "Y" jumper configuration. Timer 2 and Timer 1 from the Master Timer Provider, or the module that is executing Test 3 with -lp specified 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 done in Test 3 but does not use Timer 1 or Timer 2. The Clock and Gate will both come from the Timers on the Master AXPvme module. Timer 0 will interrupt when the Gate is received and its count is decremented to 0.

Note

IMPORTANT: A VMEbus P2 loopback connector is required. See Figure 5–2, for a description of the loopback connections.

Console Command: i8254_diag -t 4

Command line parameters:

- -np: no print qualifier; if specified no P2 connector message is printed

Miscellaneous Notes

1. Test #3 must be invoked, with the -lp qualifier, on the *master* module prior to invoking this test.

Timer 2 Interrupt Test

This test exercises Timer 2 with the timer interrupt disabled. In the AXPvme design, 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 set the Timer 2 status bit and not generate a CPU interrupt.

The interrupt service routine global flag is checked verifying that the interrupt service routine was not invoked. The Timer 2 status bit is checked to indicate the interrupt took place.

Console Command: i8254_diag -t 5

Command line parameters:

- none

Miscellaneous Notes

1. 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. Bit 4 is toggled by writing to address 0x4010; bit 5 is toggled by writing to address 0x4014. In both cases, the data written is Don't Care.
2. A read of the Interrupt Status Register at address 0x4014 causes both interrupt status bits (bits 0 and 1) to be cleared.
3. Due to hardware limitations on interrupt detection, the value programmed into Timer 2 must be greater than 2.
4. See the Intel 8254 Interval Timer sheet for more details.

VIC Timer Interrupt Test

This test verifies the interrupt path of Timer 1 (Periodic RT Timer) through the VIC. First, the test verifies that Timer 1 will not interrupt through the VIC with the local interrupt disabled. Secondly, the test verifies that Timer 1 will successfully interrupt through the VIC with the local interrupt enabled.

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, 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 1 is reprogrammed.

A global interrupt count flag is checked verifying whether the interrupt service routine was invoked.

Console Command: i8254_diag -t 6

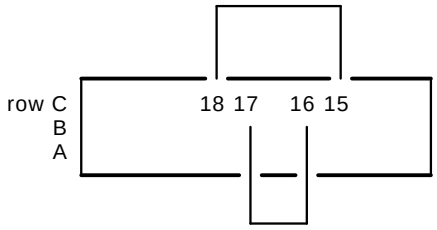
Command line parameters:

- none

Figure 5–2 Loopback Descriptions for Interval Timer Test 3 and 4

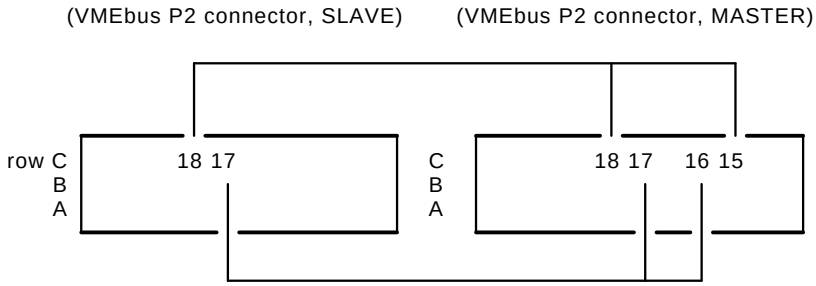
Configuration for Interval Timer test 3

To make a loopback for test 3 connect pin C15 to C18. With a second jumper, connect C16 to C17.



Configuration for Interval Timer test 4 (MASTER/SLAVE AXPvmes)

For test 4, the MASTER signals must be the input for the second AXPvme module. Connect pins C15 and C18 of the master to C18 of the SLAVE. With a second jumper, connect C16 and C17 of the MASTER to C17 of the SLAVE.



Miscellaneous Notes

1. The local interrupt at the VIC must be programmed to edge-sensitive.

DECchip 21040 Ethernet Controller Tests

These diagnostics verify that the internal and external loopback mechanisms are properly operating in the DECchip 21040 Ethernet controller chip as well as performing writes and reads to all configuration registers.

Ethernet Internal Loopback Test

The NI 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.

The NI external loopback test transmits Ethernet packets from the transmit ring in main memory and places them on the network medium (AUI cable). It concurrently listens to the line which 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

The external loopback test should not be run for long periods of time on an open network. It generates substantial amounts of traffic and can cause network performance degradation. The test will work just as well with a terminated cable or AUI termination.

These two tests check 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 AUI interface.

Console Command

- **For internal loopback:** niil_diag -t 1
- **For external loopback:** niil_diag -t 2

Command line parameters:

- -dd: print detailed test information on each pass.

DECchip 21040 PCI Configuration Register Dump

This 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

This test reads the CSRs of the DECchip 21040 and prints them to the standard output.

Console Command: nicsr_diag -t 2

DECchip 21040 Configuration Register Test

This test performs writes and reads to the chip's configurations registers with data patterns of all 1's, all 0's, and alternating 1's and 0's. Upon exiting, the test returns the configuration registers to their initial values.

Console Command: nicsr_diag -t 3

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. This test will be run only on power-up.

Memory ECC Detection Test

These tests verify the DRAMs used to store the ECC checkbits, and checks that the ECC detection logic is functional.

All drivers are stopped before executing any of these tests and restarted once the testing has completed.

These tests check the following logic:

- ECC memory bits
- ECC logic

Single Bit ECC Test

This test verifies that the ECC detection logic is functional. Single bit errors are forced on data writes. The corrupted locations are then read back and the ECC error register is checked for the correct error type.

Console Command: memecc_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Double Bit ECC Test

This test verifies that the ECC detection logic is functional. Double bit errors are forced on data writes. The corrupted locations are then read back and the ECC error register is checked for the correct error type.

Console Command: memecc_diag -t 2

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Miscellaneous Notes

1. Test prerequisite: The ECC must be properly initialized by SROM.

Backup Cache Tests

These tests verify the operation of the SRAMs. These tests are only executed if the optional Backup cache is present (determined by reading the Module Configuration Register).

These tests check the following logic:

- Backup cache data RAMs
- Backup cache tag RAMs
- Backup cache ECC RAMs

Bcache Victim Eject Test

This test verifies bcache data integrity, checks that "victim ejects" occur, and ensures that data is forced back on reads.

Console Command: bcache_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Bcache Tag Parity Test

Using the diagnostic features of the Alpha chip backup cache interface this test will force bad tag parity. A set pattern is written to and read from the backup cache data RAMs and the appropriate error settings are checked in the ESR or Error Status Register.

Console Command: bcache_diag -t 2

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Correctable Error Test

Using the diagnostic features of the Alpha chip backup cache interface this test will force correctable errors. A set pattern is written to and read from the backup cache data RAMs and the appropriate error settings are checked in the ESR or Error Status Register.

Console Command: bcache_diag -t 3

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Uncorrectable Error Test

Using the diagnostic features of the Alpha chip backup cache interface this test will force uncorrectable errors. A set pattern is written to and read from the backup cache data RAMs and the appropriate error settings are checked in the ESR or Error Status Register.

Console Command: bcache_diag -t 4

Command line parameters:

- -dd: print detailed test information on each pass.
- -np: suppress printing for options without external backup cache.

Miscellaneous Notes

1. Test prerequisite: Backup Cache (SRAM) must be enabled and initialized (that is, ECC, Valid Tag, and Tag parity) by the SROM. Memory ECC (DRAM) must also be initialized by the console.

8530 Serial Communication Controller Tests

These diagnostics test the functionality of the 8530 Serial Communication Controller chip and surrounding external circuitry, including decoders, line drivers/receivers, and printed circuit board module etch.

The 8530 has two serial communication channels, A and B. Channel A is dedicated to the console. This diagnostic does not test channel A, however, the Serial ROM performs a minimal channel A register access test. This diagnostic tests only channel B functionality.

8530 UAR/T Channel B Internal Loopback Test

This test exercises channel B of the 8530 serial communication controller in internal loopback mode. The test will exercise various serial port control bits and transmit and receive buffers.

First, the 8530 is configured to transmit and receive. Various data patterns are then transmitted. The data received is subsequently compared with the data that was sent.

Console Command: z8530_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.
- -lp: Loop on diagnostic until failure or ^C.

Miscellaneous Notes

1. Some write-only registers are modifed; consequently, the registers cannot be returned to their original state at the end of the diagnostic.
2. No loopback connector is required.

8530 UAR/T Channel B External Loopback Test

This test exercises channel B of the 8530 serial communication controller in external loopback mode. The test will exercise various serial port control bits, transmit and receive buffers, and external support circuitry.

First, the 8530 is configured to transmit and receive. The user is then asked to install an external loopback connector. Upon confirmation that the loopback connector is installed, various data patterns are transmitted. The data received is compared with the data that was sent.

Console Command: z8530_diag -t 2

Command line parameters:

- -dd: print detailed test information on each pass.
- -lp: Loop on diagnostic until failure or ^C.

Miscellaneous Notes

1. Some write-only registers are modifed; consequently, the registers cannot be returned to their original state at the end of the diagnostic.
2. This diagnostic is intended to be an extended test.

3. This diagnostic will fail if no loopback connector is installed.

8530 UAR/T Channel B Interrupt Test

This test exercises the ability of channel B to interrupt the CPU on every character it receives and when the transmit buffer is empty.

The 8530 is programmed to internal loopback mode. A character is loaded into the transmit buffer. When the character is transmitted, the transmit buffer becomes empty generating a transmit interrupt to the CPU. Additionally, upon receipt of the character, a receive interrupt is generated.

The interrupt service routine (ISR) invoked will increment an interrupt counter (separate counter for receive and transmit interrupts) and sets a global flag indicating the interrupt took place and that software was dispatched to the correct point. Lastly, the interrupt count is verified to have occurred once for each interrupt.

Console Command: z8530_diag -t 3

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. Some write-only registers are modifed; consequently, the registers cannot be returned to their original state at the end of the diagnostic.

DALLAS DS1386 RAMified Watchdog Timekeeper Tests

The DS1386 consists of 32 KB of NVRAM and a real time clock. This diagnostic tests each of these features on an individual basis. The diagnostic tests the DS1386, decoders, and printed circuit board module etch.

The functionality of the watchdog feature is to be tested in a separate diagnostic. No alarm features are tested, since the alarms are not used.

Tests 1 through 3 exercise the NVRAM. Tests 4 and 5 exercise the real time clock.

The NVRAM is be tested on a page basis; there are 128 pages each containing 256 bytes. The NVRAM, therefore, contains 128 pages. However, the first page has reserved addresses for the real time clock registers.

NVRAM March I Test

This test writes, reads, and compares all 32 KB of NVRAM with data patterns of all 1's, all 0's, alternating 1's and 0's, and shifting 1's and 0's. If the quick verify switch is set (default) only the first location of each page is tested. The no quick verify switch 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 line parameters:

- -dd: print detailed test information on each pass.
- -nqv: test every location in NVRAM, default is to test 1 location per 256 byte page.

Miscellaneous Notes

1. This diagnostic is considered to be an extended test.

NVRAM Address-On-Address Test

The NVRAM locations in the DS1386 are byte wide. Therefore, you don't have enough room to write the unique address into each corresponding location. However, this test will write the unique page offset to it's corresponding location in NVRAM.

This test writes, reads, and compares all 32 KB of NVRAM using this unique page offset for test data. If the quick verify switch is set (default) only the first location of each page is tested. The no quick verify switch 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 line parameters:

- -dd: print detailed test information on each pass.
- -nqv: test every location in NVRAM, default is to test 1 location per 256 byte page.

Miscellaneous Notes

1. This diagnostic is considered to be an extended test.

NVRAM March II Test

This test verifies NVRAM addressing by marching (write, read, and compare) a 0x00 byte value through a field of 0xFF. Each iteration will read the entire 32 Kbyte for background pattern of 0xFF. If the quick verify switch is set (default) only the first location of each page is tested. The no quick verify, -nqv, switch 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 3

Command line parameters:

- -dd: print detailed test information on each pass.
- -nqv: test every location in NVRAM, default is to test 1 location per 256 byte page.

Miscellaneous Notes

1. This diagnostic is considered to be an extended test.

TOY Bitwalk Test

This diagnostic does a walking 1, walking 0, and A5 register test on the Time of Year (TOY) 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 will rollover 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 rollover to the appropriate value.

The diagnostic cleans up by reenabling the Watchdog Reset bit in the Module Control register and restoring the original contents of the TOY Command register.

Note

The current date and time will have to be reset after invoking this diagnostic test since approximately 3 seconds of time will be lost for each pass.

Console Command: ds1386_diag -t 4

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. This diagnostic is considered to be an extended test.

TOY Time Advancement Test

This diagnostic is intended to be used as a power-up diagnostic. It verifies that the TOY 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 advanced from 59 to 0. If the rollover case is encountered, the test sleeps for another second and reads the register again. This is repeated for four times.

Console Command: ds1386_diag -t 5

Command line parameters:

- -dd: print detailed test information on each pass.

Miscellaneous Notes

1. This diagnostic is considered to be a power-up self-test diagnostic.

LAN Address ROM Test

This diagnostic tests 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

This diagnostic 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 line parameters:

- -dd: enables printing LAN ROM address to screen
- -np: no print, if specified, LAN ROM address is not printed to screen

Miscellaneous Notes

1. The LAN Address ROM octets must be read via longword aligned byte accesses.
2. This diagnostic is considered to be an extended test.

LAN Address ROM Verification Test

This 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 line parameters:

- -dd: enables printing LAN ROM address to screen

Miscellaneous Notes

1. The LAN Address ROM octets must be read via longword aligned byte accesses.
2. This test is considered a power-up self-test diagnostic.

Figure 5–3 LAN Address ROM Format

NCR 53C810 PCI-SCSI IO Processor Tests

These 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-up check of the NCR810's low-level modes through Programmed I/O issued from the CPU. There are no NCR810 SCRIPTS executing 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. (SW Fail point 1,2)

Important Note: If any of these tests fail the Console SCSI driver will not be restarted after the test. This will cause SCSI devices connected to the system to be removed from the device list (The command "show device" lists the currently installed devices) and any attempts to run the disk exerciser or boot from a disk will fail. The NCR810_diag routine also checks for the presence of the 53C810 option. A "no device present" message will be displayed to the screen. (SW Fail point 1)

NCR810 PCI COnfiguration Register Test

This test prints the current setting of the NCR810 PCI Configuration registers to the console screen using a formatted output.

Console Command: ncr810_diag -t 1

Command line parameters:

- Print the Config register if -np command qualifier is NOT specified.

NCR810 Command/Status Register Dump

This test displays the contents of all of the Command/Status registers on your screen. No test of the contents is performed.

Console Command: ncr810_diag -t 2

Command line parameters:

- Print the Config register if -np command qualifier is NOT specified.

NCR810 Command/Status Register Test

This test writes, reads, and compares all of the NCR810 Command/Status registers that are feasible to test. When the test finishes, it returns the registers to their initialized values.

Console Command: ncr810_diag -t 3

Command line parameters:

- -lp: loop on write/read if -lp qual present.

NCR810 Command/Status Register Reset Value Test

This test checks that a reset of the NCR810 sets the Command/Status registers to their default values as defined by the NCR810 53C810 chip specification.

Console Command: ncr810_diag -t 4

NCR810 Internal Loopback Test

This test performs a SCSI loopback internal to the NCR810 chip. The following data patterns are used: all 1's, all 0's, alternating 1's and 0's. 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

This test performs an internal SCSI loopback that also drives the signal lines on the SCSI bus.

All devices must be removed from the SCSI bus before running this test. Devices on the bus will interfere with the test and cause false error reports. Also, the test data may produce illegal device instructions and cause the devices to hang.

First the SCSI bus is placed in a high impedance state by loading a data pattern that causes the output drivers to draw no current. Then the output latches are checked 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 1's, all 0's, alternating 1's and 0's.

Console Command: ncr810_diag -t 6

NCR810 Interrupt Test

This test verifies the interrupt connection between the NCR810 and the SIO controller to the CPU. A general purpose timer is enabled which generates an interrupt that is dispatched to the CPU through the SIO controller. The Console PALcode will dispatch to the NCR810_diag interrupt service routine, which will clear the interrupt.

Console Command: ncr810_diag -t 7

Miscellaneous Notes

1. These tests will not run in parallel with the SCSI exerciser tests.
2. No external Loopback connectors are needed for the loopback tests.
3. References - NCR 53C810 PCI-SCSI I/O Processor specification rev 2.1

Watchdog Timer Interrupt Test

This test verifies the functionality of the watchdog timeout by its ability to handle a user programmed watchdog reset.

This test checks the following logic:

- Watchdog timer
- Some Reset logic
- DS1386 Time-of-Year device

Watchdog Timer Interrupt Test

The Diagnostic In Progress (DIP) bit is set and a watchdog timeout is invoked by loading a short time value into the watchdog timeout register. The user is queried to be sure the watchdog LED is off. Upon expiration of the watchdog, a HALT interrupt is expected. After the expected time, the Reset Reason Register is evaluated. If the HALT interrupt did not occur, or the watchdog reason was not set, an error callout is made. Also, the user is asked to verify the watchdog LED is now on. At the end of the test, the Watchdog Timer and DIP bit are disabled.

Console Command: wdog_diag -t 1

Command line parameters:

- -dd: print detailed test information on each pass.
- -nc: No Confirmation; user is not prompted to verify state of LED
- -np: No Print; overrides the -nc qualifier, no user prompts

Miscellaneous Notes

1. The purpose of setting the DIP bit is to avoid an actual system reset when the watchdog timer expires. The watchdog expiration first causes a HALT interrupt. Approximately 300ms later an actual system reset will occur, unless the DIP bit is set. The Reset Reason register will show a watchdog reset reason whether or not the DIP bit is set. We use the HALT interrupt and the reset reason for this diagnostic. User interaction can be suppressed with the

VME Interface Tests

These tests verify the VME Interface logic on the AXPvme module including the VME Interface Processor (VIP), the Cypress VIC064, the scatter/gather RAMs, and some of the interrupts paths from the VME corner to the Alpha processor. No VMEbus transactions are performed by these tests and therefore require no additional VMEbus modules.

VIP PCI Configuration Register Test

This test reads the first 8 longwords of VIP PCI Configuration space. Only the Device and Vendor ID, and Base Address 0, 1, 2, and 3 are compared to an expected value. The remaining longwords are always read and displayed only if the *-dd* command-line qualifier is present.

Console Command: vip_diag -t 1

Command line parameters:

- -dd: print detailed test information.

VIP Register Write/Read Test

This 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 line parameters:

- -dd: print detailed test information.

VIC Register Write/Read Test

This 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 line parameters:

- -dd: print detailed test information.

VME Scatter/Gather RAM Test

This test verifies the integrity of the Scatter/Gather RAM by performing write, read, and verify of various patterns to the entire Scatter/Gather RAM.

Console Command: vip_diag -t 4

Command line parameters:

- -dd: print detailed test information on each pass.

VIP/VIC Local Interrupt Test #1

This test uses the VIC Local Interrupt #2 source to generate interrupts in which the VIP detects and responds to. Since the VIC provides so much control of its interrupts, it is relatively easy to generate a local interrupt.

This test verifies the following:

1. The 3 IPL lines between the VIC and VIP
2. The ability of the VIP to generate a PCI interrupt
3. The Interrupt Acknowledge cycle to the VIP/VIC
4. The IPL update functionality in the VIP HWIPL register

Console Command: vip_diag -t 5

Command line parameters:

- -dd: print detailed test information.

VIP/VIC Local Interrupt Test #2

By invalidating a Scatter/Gather entry and then attempting to access VME through it, the following can be tested:

1. The VIP can detect an Outbound Error, indicated in the VIP_BESR
2. The VIP_ICR can properly enable/disable interrupt delivery

Console Command: vip_diag -t 6

Command line parameters:

- -dd: print detailed test information.

Miscellaneous VIP BESR Test

This test verifies bits in the VIP bus error/status register (BESR). The diagnostic is divided into the following parts:

1. Part 1 - Configuration Overlap Error
2. Part 2 - Local Bus Error

Console Command: vip_diag -t 10

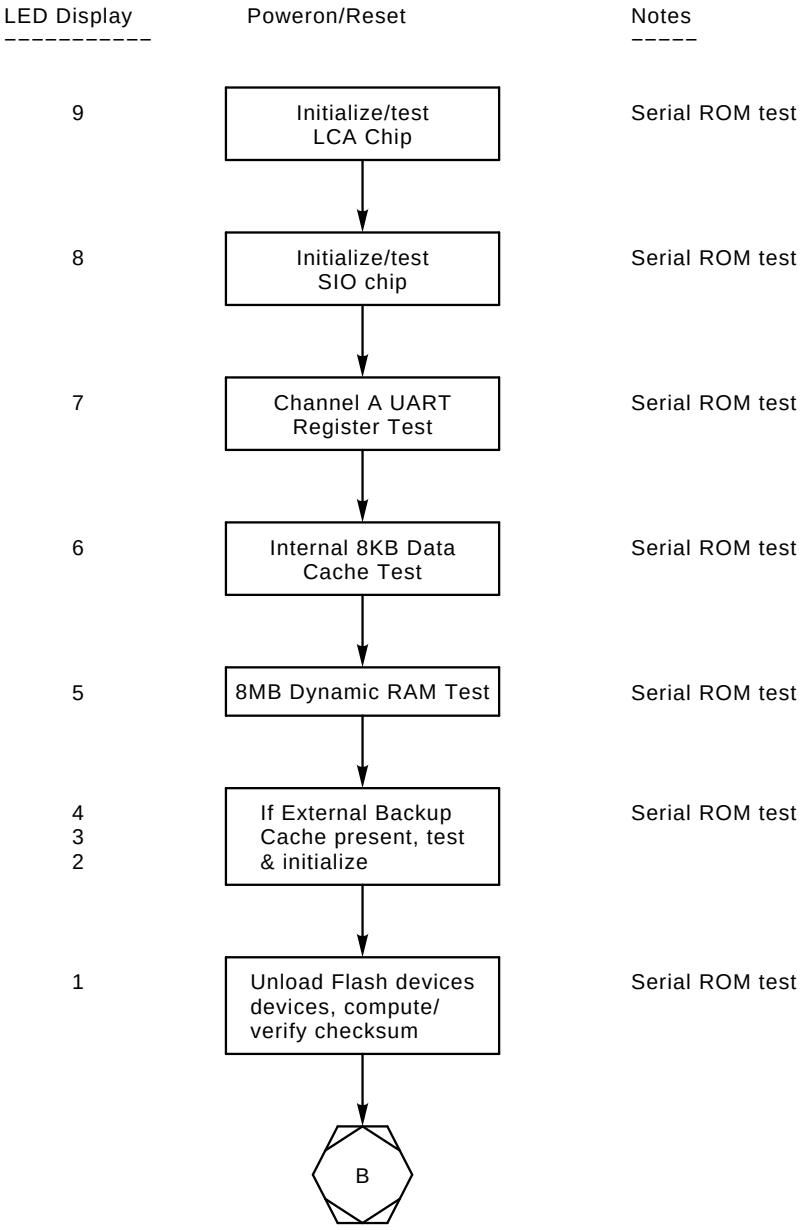
Command line parameters:

- -dd: print detailed test information.

5.4 Test Sequence

The diagnostic test sequence for a full power-up reset and initialization is shown in Figure 5–4.

Figure 5–4 SROM Test Flows



5.4 Test Sequence

Figure 5–5 Console Power-On/Self-Test Flows

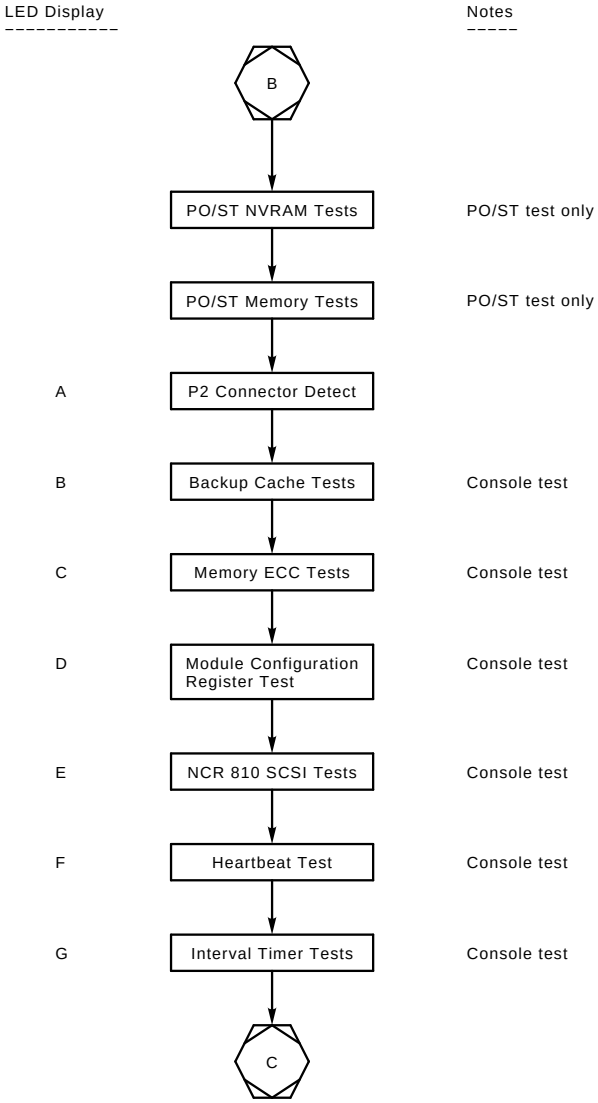
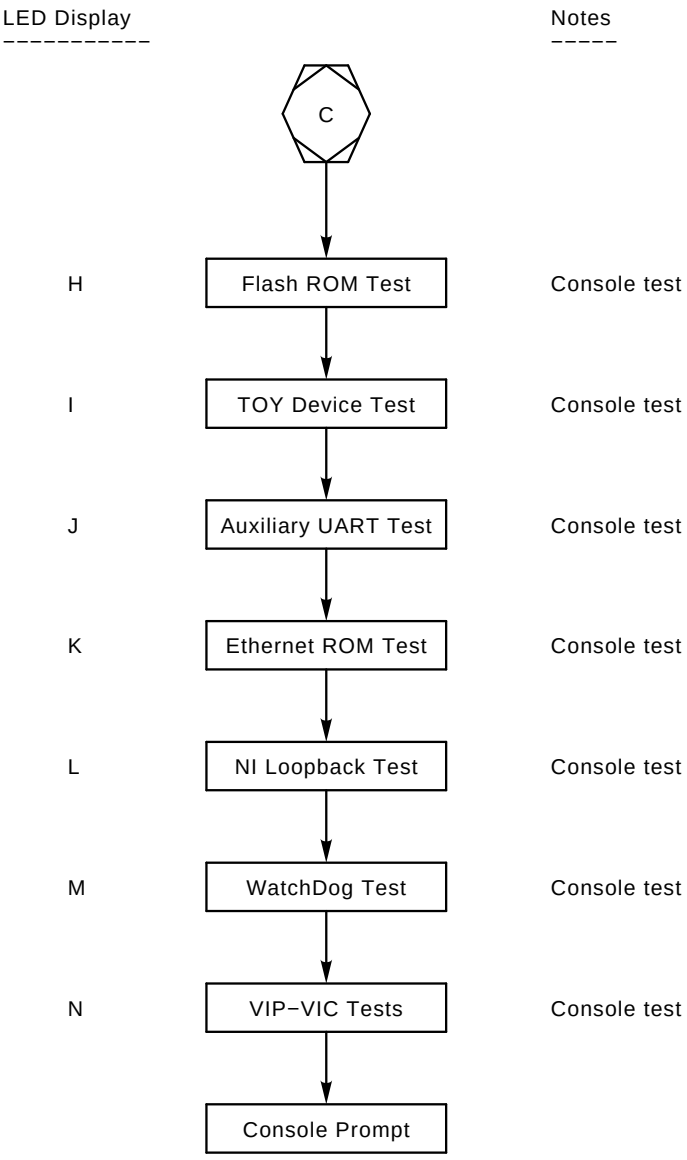


Figure 5–6 Console Power-On/Self-Test Flows



A

Specifications

Table A–1 shows the physical and environmental specifications for the AXPvme module. Table A–2 shows the power supply current and power for the AXPvme module. Stresses beyond those specified may cause permanent damage to the module.

Table A–1 Physical and Environmental Specifications

Characteristic	Specification
Industry standard	VME 6U module
Operating temperature	0°C to 50°C (32°F to 122°F)
Storage temperature	–40°C to 66°C (–40°F to 151°F)
Temperature change	20°C/hour (36°F/hour)
Relative humidity	10% to 95% (noncondensing)
Airflow	200 lfm minimum at 50°C abient inlet air temperature over the large square processor heatsink at the center of the AXPvme module
Vibration:	Operating in a suitable enclosure
	0.5g Pk 22.1–260 Hz
	0.25g Pk 200–500 Hz

Table A–2 Power Supply Current and Module Power Dissapation

CPU Modules w/Memory	Amps @ 5 V	Amps @ 12 V (note 1)	Amps @–12 V	Module Heat Dissapation
AXPvme 230	9.9 A	0.5 A	0.1 A	57 W
AXPvme 166	8.0 A	0.5 A	0.1 A	47 W
AXPVME 100	6.2 A	0.5 A	0.1 A	38 W
AXPVME 160	9.4 A	0.5 A	0.1 A	54 W
AXPVME 64	6.5 A	0.5 A	0.1 A	40 W
AXPVME 64LC	6.0 A	0.5 A	0.1 A	37 W

Options	Amps @ 5 V	Amps @ 12 V	Amps @–12 V	Power Dissapation
SCSI Termination	0.8 A Max	0.0 A	N/A	4 W Max
PMC Option Slot Budget (actual use)	2.0 A Max	N/A	N/A	10 W Max

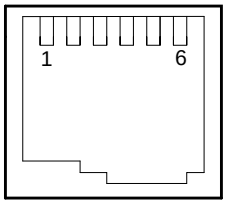
B

AXPvme Connectors

B.1 Serial Line Connectors

The console and auxiliary serial line interfaces support data leads only. They do not support modem control. Figure B–1 shows the pin numbers for the console and auxiliary serial line connectors and Table B–1 shows the signal descriptions.

Figure B–1 Serial Line Connectors



LJ-03756-TI0

Table B–1 Serial Line Connectors

Pin	Signal
1	Ready Out (always asserted)
2	Transmit Data +
3	Transmit Data –
4	Receive Data –
5	Receive Data +
6	Ready In (not used)

B.2 Ethernet AUI Connector

Figure B–2 shows the pin numbers for the Ethernet AUI connector and Table B–2 shows the signal descriptions.

Figure B–2 Ethernet AUI Connector

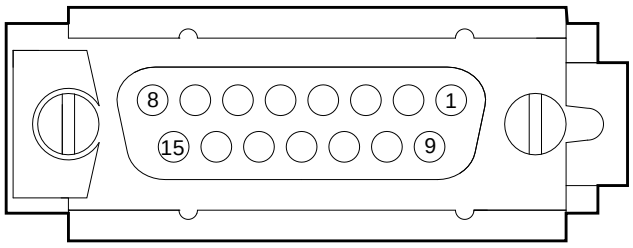


Table B–2 Ethernet AUI Connector

Pin	Signal	Description
1	GND	Ground
2	COLL+	Collision Detect +
3	XMIT+	Data Out +
4	GND	Ground
5	RCV+	Data In +
6	GND	Ground
7	CTRL OUT A	Not used, but it is terminated on the AXPvme module
8	GND	Ground
9	COLL–	Collision Detect –
10	XMIT–	Data Out –
11	GND	Ground
12	RCV–	Data In –
13	+12V	+12 V supply to transceiver
14	GND	Ground
15	CTRL OUT B	Not used, but it is terminated on the AXPvme module

B.3 SCSI Connector

Figure B–3 shows the pin numbers for the SCSI connector and Table B–3 shows the signal descriptions.

Figure B–3 SCSI Connector

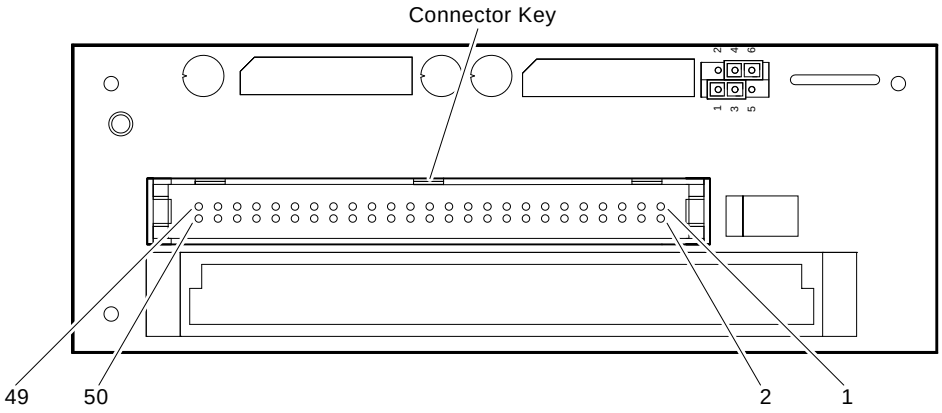


Table B–3 SCSI Connector

Pin	Signal	Pin	Signal
1	GND (Ground)	26	TERMPWR2 (+5V Termination power)
2	–DB0 (Data 0)	27	GND
3	GND	28	GND
4	–DB1 (Data 1)	29	GND
5	GND	30	GND
6	–DB2 (Data 2)	31	GND
7	GND	32	–ATN (Attention)
8	–DB3 (Data 3)	33	GND
9	GND	34	GND
10	–DB4 (Data 4)	35	GND
11	GND	36	–BSY (Busy)
12	–DB5 (Data 5)	37	GND
13	GND	38	–ACK (Acknowledge)
14	–DB6 (Data 6)	39	GND
15	GND	40	–RST (Reset)
16	–DB7 (Data 7)	41	GND
17	GND	42	–MSG (Message)
18	–DBP (Data parity)	43	GND
19	GND	44	–SEL (Select)
20	GND	45	GND
21	GND	46	–C/D (Cmd/Data)
22	GND	47	Not used
23	GND	48	–REQ (Request)
24	GND	49	GND
25	Not used	50	–I/O (Input/Output)

B.4 VMEbus P2 Connector

Figure B–4 shows the pin numbers for the VMEbus P2 connector on the breakout module and Table B–4 shows the signal descriptions.

Figure B–4 VMEbus P2 Connector

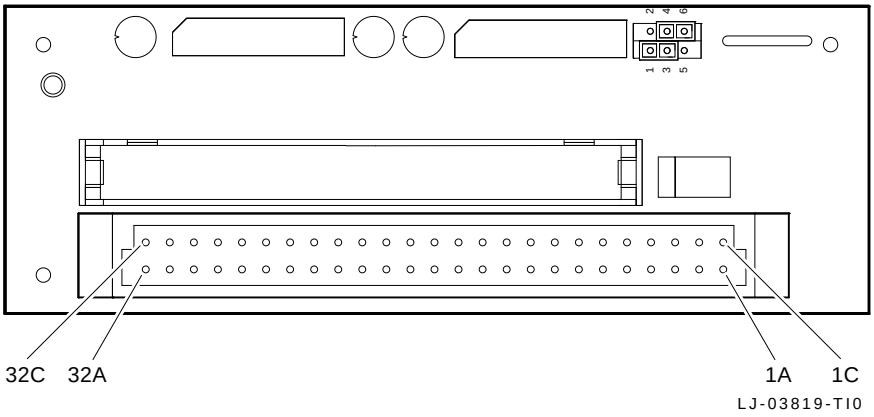


Table B–4 VMEbus P2 Connector

Pin	Row A	Row C
1	SCSI_DATA<0> L	+5V
2	SCSI_DATA<1> L	+5V
3	SCSI_DATA<2> L	GND
4	SCSI_DATA<3> L	GND
5	SCSI_DATA<4> L	+5V
6	SCSI_DATA<5> L	+5V
7	SCSI_DATA<6> L	GND
8	SCSI_DATA<7> L	GND
9	SCSI_DP L	WD_STATUS_OC H
10	SCSI_ATN L	GND
11	SCSI_BSY L	GND
12	SCSI_ACK L	+5V
13	SCSI_RST L	+5V
14	SCSI_MSG L	EXT_RESET L
15	SCSI_SEL L	TMR2_EXT_OP L
16	SCSI_CD L	TMR1_EXT_OPL
17	SCSI_REQ L	TMR_MINOR_IP L
18	SCSI_IO L	TMR_MAJOR_IP L
19	Not used	SCSI_TERMPWR H (AXPvme 64 only)
20	SROMD H	SROMOE L
21	SROMDIS H	SROMCLK H
22	GND	GND

(continued on next page)

Table B–4 (Cont.) VMEbus P2 Connector

Pin	Row A	Row C
23	Not used	SCSI_TERMPWR H (AXPvme 160 only)
24	+V5	GND
25	+V5	GND
26	+V5	GND
27	+V5	GND
28	GND	+5V
29	GND	+5V
30	GND	GND
31	+V5	+5V
32	+V5	+5V

B.5 PCI Option Connectors

Figure B–5 shows the pin numbers and Table B–5 and Table B–6 show the signal descriptions for the PCI option connectors.

Figure B–5 PCI Option Connectors

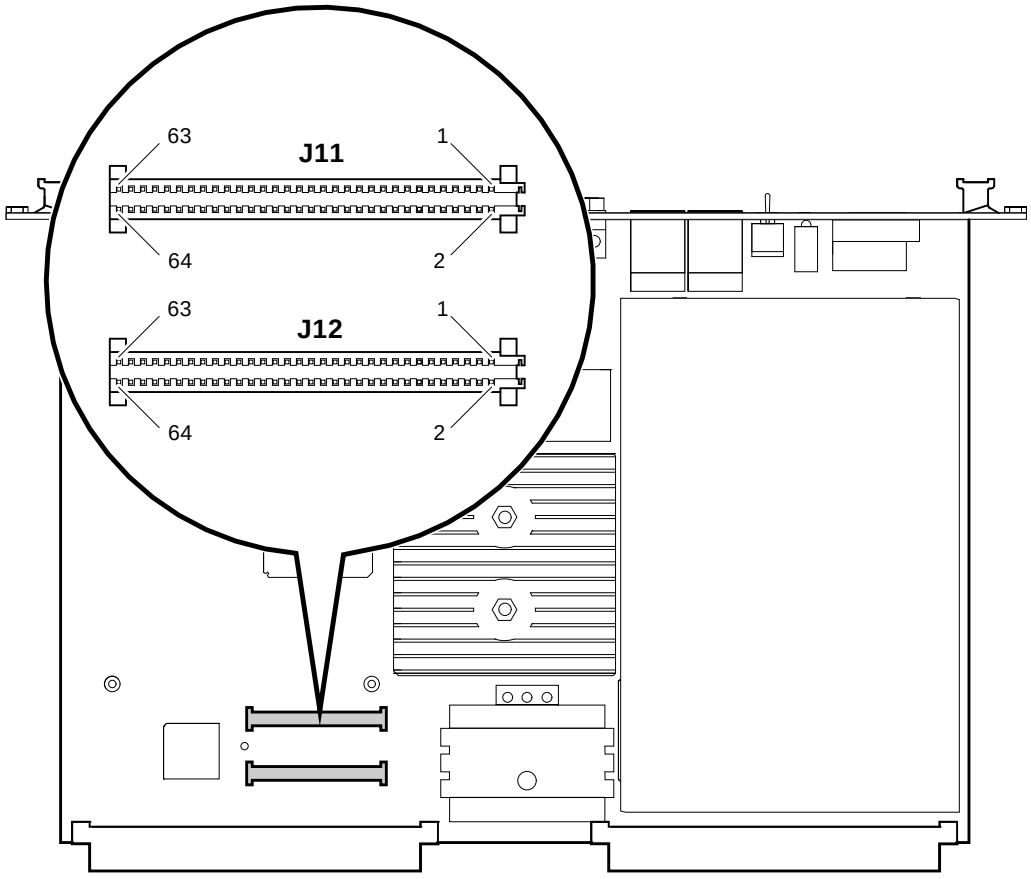


Table B–5 PCI Option J11 Connector

Pin	Signal	Pin	Signal
1	Not used	33	PCIFRAME L
2	–12V	34	GND
3	GND	35	GND
4	PCIOPT_IRQA L	36	PCIRDY L
5	PCIOPT_IRQB L	37	PCIDEVSEL L
6	PCIOPT_IRQC L	38	+5V
7	PCI_PRESENT1 L	39	GND
8	+5V	40	PCILOCK L

(continued on next page)

Table B–5 (Cont.) PCI Option J11 Connector

Pin	Signal	Pin	Signal
9	PCIOPT_IRQD L	41	Not used
10	Not used	42	Not used
11	GND	43	PCIPAR H
12	Not used	44	GND
13	PCICLK_OPT H	45	Not used
14	GND	46	PCIAD<15> H
15	GND	47	PCIAD<12> H
16	PCIGNT_OPT1 L	48	PCIAD<11> H
17	PCIREQ_OPT1 L	49	PCIAD<9> H
18	+5V	50	+5V
19	Not used	51	GND
20	PCIAD<31> H	52	PCICBE<0> L
21	PCIAD<28> H	53	PCIAD<6> H
22	PCIAD<27> H	54	PCIAD<5> H
23	PCIAD<25> H	55	PCIAD<4> H
24	GND	56	GND
25	GND	57	Not used
26	PCICBE<3> L	58	PCIAD<3> H
27	PCIAD<22> H	59	PCIAD<2> H
28	PCIAD<21> H	60	PCIAD<1> H
29	PCIAD<19> H	61	PCIAD<0> H
30	+5V	62	+5V
31	Not used	63	GND
32	PCIAD<17> H	64	Not used

Table B–6 PCI Option J12 Connector

Pin	Signal	Pin	Signal
1	+12V	33	GND
2	Not used	34	Not used
3	Not used	35	PCITRDY L
4	Not used	36	+3V
5	Not used	37	GND
6	GND	38	PCISTOP L
7	GND	39	PCIPERR L
8	Not used	40	GND
9	Not used	41	+3V
10	Not used	42	PCISERR L

(continued on next page)

Table B–6 (Cont.) PCI Option J12 Connector

Pin	Signal	Pin	Signal
11	PCI_PRESENT2	43	PCICBE<1> L
12	+3V	44	GND
13	PCIRST L	45	PCIAD<14> H
14	PCI_PRESENT3	46	PCIAD<13> H
15	+3V	47	GND
16	PCI_PRESENT4	48	PCIAD<10> H
17	Not used	49	PCIAD<8> H
18	GND	50	+3V
19	PCIAD<30> H	51	PCIAD<7> H
20	PCIAD<29> H	52	Not used
21	GND	53	+3V
22	PCIAD<26> H	54	Not used
23	PCIAD<24> H	55	Not used
24	+3V	56	GND
25	IDSEL L (PCIAD<15> H)	57	Not used
26	PCIAD<23> H	58	Not used
27	+3V	59	GND
28	PCIAD<20> H	60	Not used
29	PCIAD<18> H	61	Not used
30	GND	62	+3V
31	PCIAD<16> H	63	GND
32	PCICBE<2> L	64	Not used

Index

D

E

F

G

H

I

T

U

V

W

Reader's Comments

AXPvme
Single-Board Computer
Technical Description
EK-EBV1X-TD. B01

Your comments and suggestions help us improve the quality of our publications.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

For software manuals, please indicate which version of the software you are using: _____

_____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Shared Engineering Services
MLO5–5/E76
2 THOMPSON STREET
MAYNARD, MA 01754-1716

Do Not Tear – Fold Here