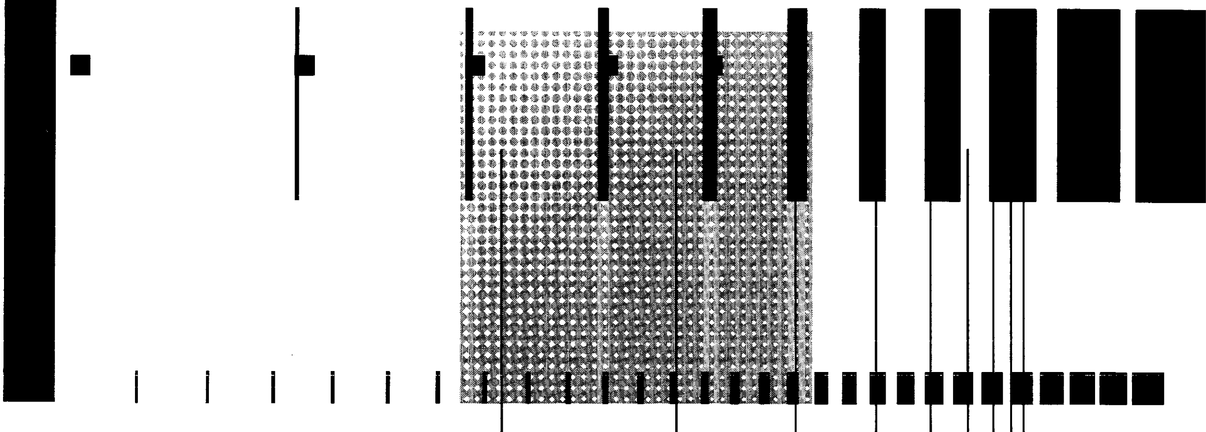
# *M/120 RISComputer Technical Reference*

*Order Number 3112DOC*



mips

The power of RISC is in the system.

# *M/120 RISComputer Technical Reference*

*Order Number 3112DOC*

**September 1988**

---

*Your comments on our products and publications are welcome. A postage-paid form is provided for this purpose on the last page of this manual.*

Mfg. Part Number 84-00058-B/02-00159B

© 1988 MIPS Computer Systems, Inc. All Rights Reserved.

RISCompiler and RISC/os are Trademarks of MIPS Computer Systems, Inc.
UNIX is a Trademark of AT&T Bell Laboratories.
Ethernet is a Trademark of XEROX.
Ada is a registered trademark of the U. S. Government (Ada Joint Program Office.)
VADS and Verdix are registered trademarks of the Verdix Corporation.
APSO and GVAS is a trademark of the Verdix Corporation

MIPS Computer Systems, Inc.
930 Arques Ave.
Sunnyvale, CA 94086.

---

Customer Service Telephone Numbers:

| | | |
|---|---|---|
| California: | (800) | 992–MIPS |
| All other states: | (800) | 443–MIPS |
| International: | (415) | 330–7966 |

Mfg. Part Number 84–00058B/02–00159B

# *About This Book*

This book provides information on the MIPS M/120 RISComputer System, a 32–bit, multi–user, UNIX system. The M/120 uses the R2000 processor and is based on the MIPS RISC (Reduced Instruction Set Computer) architecture.

## Audience

This book should be read by anyone who needs to unpack, setup, operate, write programs for, or interface devices to the M/120 RISComputer.

## Topics Covered

This book contains the following chapters:

- **Chapter 1, System Overview.** Gives an overview of capabilities and features of the system.
- **Chapter 2, Installation.** Describes the installation instructions for the M/120 RISComputer System.
- **Chapter 3, Programming Model.** Provides a system programmer's view of the M/120 System. Defines the system memory map, the interrupt system, and the purpose of the system's configuration and status registers. Also summarizes the addresses and functions of programmable registers provided by the devices in the M/120's I/O subsystem.
- **Chapter 4, Writing Device Drivers.** Describes the specific information needed by programmers writing and installing device drivers for the M/120 under the RISC/os™ (UMIPS) operating system.
- **Chapter 5, PROM Monitor.** Describes the PROM Monitor which provides the tools for examining and changing memory, downloading programs over serial lines (RS–232C), and booting programs from disk, tape, or Ethernet. The PROM Monitor also provides tools for altering power–up configuration options in non–volatile RAM.

- **Appendix A, AT Bus Compatibility Considerations.** Describes the system's implementation of the AT bus interface and includes details of the configuration options available.
- **Appendix B, Tape Drive Operation and Maintenance.** Describes the operating procedures for the standard cartridge tape drive and the maintenance considerations for the drive.
- **Appendix C, Installing Disk Drives in the Expansion Cabinet.** Describes how to install disk drives in the optional expansion cabinet that can be added to the M/120 system.
- **Appendix D, Power On Diagnostics.** Describes the built–in diagnostic program that is executed when the system is powered–up.
- **Appendix E, Sample Driver Listing.** Contains a listing for a sample device driver program.
- **Appendix F, Standalone Programs.** Describes the *format* and *sash* standalone programs.
- **Index.** Contains index entries for this publication.

# For More Information

The following publications contain additional information that you may need as you use the M/120 RISComputer:

- *MIPS RISC Architecture, Prentice–Hall ISBN0–13–584749–4*
- *MIPS Language Programmer's Guide 02–00035*
- *MIPS Assembly Language Programmer's Guide 02–00036*
- *RISC/os (UMIPS) System Administrator's Guide 02–00136*

# Contents

*Chapter 1*
***System Overview***

Introduction 1–1
System Description 1–1
- Motherboard 1–3
- Central Processing Unit 1–3
- System Memory 1–5
- AT Bus Slots 1–6
- Packaging 1–6
- Peripherals 1–6
- Controls, Switches, and Indicators 1–7
- Expansion Cabinet 1–8
- Specifications 1–9

*Chapter 2*
***Installation***

Site Selection 2–1
- Space Requirements 2–1
- Power Requirements 2–3
- Environmental Requirements 2–4

Select the Voltage 2–5
Install Additional or Optional PC Cards 2–7
- Removing the Side Panel 2–7
- Install Additional Memory Cards 2–8
- Install Optional AT Cards 2–9
- Reinstall the Side Panel 2–10

Install Serial I/O Devices 2–11
- General Considerations 2–11
- Terminal Connector Pinouts 2–12
- Connecting the Console 2–13
- Connecting Other Serial I/O Devices 2–13

Cable the System to an Ethernet Network 2–15
Power Up the M/120 System 2–16

*3*
*Programming Model*


Signal and Bit Naming Conventions 3–2
Data Formats and Addressing 3–2
System Memory Map 3–3
Interrupt System 3–6
  Interrupt Level–0 3–7
    Interrupt Status Register (ISR) 3–7
    Interrupt Mask Register 3–8
  Memory Fault Handling 3–8
    Fault ID Register (FID) 3–9
    Fault Address Register (FAR) 3–11
System Configuration Register 3–12
Direct Memory Access (DMA) 3–14
  DMA Controller Operating Modes 3–15
  DMA Controller Interface Registers 3–15
  DMA Software Control 3–16
LED Register 3–17
The ID PROM 3–17
I/O Subsystems 3–18
  Counter/Timer 3–18
    Counter/Timer Interrupt Acknowledge Registers 3–19
    Counter/Timer Register Summary 3–19
  Real–Time Clock & NVRAM 3–20
  Serial Ports 3–21
  SCSI Interface 3–23
    SCSI Controller Registers 3–24
    SCSI Operation Details 3–25
  Ethernet Interface 3–25
AT Bus Interface 3–26
  AT Bus Memory Access and Control 3–26
  AT Bus Memory Mapping 3–26
  AT Bus Byte Swapping 3–27
  AT Bus Control Registers 3–28
    AT Control Register 3–28
    AT DAckEn Register 3–30

*Chapter 4*
*Writing Device Drivers*


| | |
|---|---|
| Introduction | 4–1 |
| File Structure of the Kernel Subset | 4–1 |
| The io Directory | 4–2 |
| The master.d Directory | 4–2 |
| The bootarea Directory | 4–3 |
| AT&T and MIPS Reconfiguration Differences | 4–3 |
| AT&T's Reconfiguration Process | 4–3 |
| MIPS' Reconfiguration Process | 4–3 |
| Adding New Drivers | 4–4 |
| Set Your Environment Variable | 4–4 |
| Compile Your Driver | 4–5 |
| Create a Master File | 4–5 |
| Copy and Rename the Kernel and Sygen Files | 4–6 |
| Modify the New Kernel File | 4–6 |
| Modify the New Sysgen File | 4–6 |
| Build the Kernel | 4–7 |
| M/120 Machine Considerations | 4–7 |
| The AT Bus | 4–7 |
| AT Bus Address Space | 4–8 |
| Kernel Support Routines | 4–9 |
| Delay(n) Macro | 4–9 |
| Address Translation | 4–9 |
| Interrupt Priority Level Assignment | 4–10 |
| Changing Interrupt Levels | 4–11 |
| Kernel/PROM Interface | 4–11 |
| Memory Management | 4–12 |
| Volatile Memory and Optimizing Compilers | 4–12 |
| Write Buffer Considerations | 4–13 |
| SCSI Devices | 4–14 |
| Debugging Drivers | 4–15 |
| Halting the System | 4–15 |
| System Error Messages | 4–15 |

*Chapter 5*
*PROM Monitor*


Introduction 5–1
Description 5–1
Memory Usage 5–1
File Name Syntax 5–2
Environment Variables 5–3
Input Editing 5–4
Time of Day and Non–Volatile RAM 5–5
Using Breaks to Change Baud Rate 5–5
Extending the PROM Monitor 5–5
Command Set 5–6
auto 5–7
boot 5–8
cat 5–9
disable 5–10
dump 5–11
enable 5–13
fill 5–14
g 5–15
go 5–16
help 5–17
init 5–18
init_tod 5–19
load 5–20
p 5–22
printenv 5–22
pr_tod 5–23
setenv 5–24
sload 5–25
spin 5–26
unsetenv 5–27
warm 5–28

*Appendix A*
*AT Bus*
*Compatibility Considerations*


Memory Mapping Options A–1
AT Bus Memory Refresh A–3
Device Drivers A–3
Connectors A–3
Bus Timing A–3
DMA Operations A–3
DMA Request and Acknowledge Options A–4
DMA Terminal Count (TC) Signal A–5
AT Bus Interrupts A–6
Alternate Controllers A–7
Bus Access Control for Alternate Controllers A–7
AT Bus Option and Jumper Summary A–8
AT Bus Pin and Signal Assignments A–11


*Appendix B*
*Tape Drive Operation and Maintenance*


Operation B–1
Maintenance B–2


*Appendix C*
*Installing Disk Drives in the Expansion Cabinet*

*Appendix D*
*Power On Diagnostics*


Introduction D–1
Pon_Leds D–5
Pon_Duart D–5
Pon_Banner D–6
Pon_Cache1 D–6
Pon_Cache2 D–7
Pon_Cache3 D–8
Pon_Cache4 D–8
Pon_IdProm D–9
Pon_WB D–9
Pon_Memory D–10
Pon_Scr D–10
Pon_VM D–11
Pon_Allexc D–12
Pon_Parity D–13
Pon_NVram D–14
Pon_Timers D–15

Pon_Duarts D–16
Pon_Imr D–17
Pon_Fp1 and Fp2 D–17
Pon_UdcSlave D–18
Pon_Chain1 D–19
Pon_Chain2 D–20
Pon_ScsiSlave D–20
Pon_ScsiMaster D–21
Pon_EnetProm D–22
Pon_LanceSlave D–22
Pon_LanceMaster D–23
Pon_Atreg D–24

*Appendix E*
*Sample Driver Listing*

The C8.c Driver Program E–1
The Header file ss.h E–22
The SS.c Library E–24

*Appendix F*
*Standalone Programs*

Introduction F–1
Format F–3
Format Desription F–3
Additional Format Information F–8
Standalone Shell (sash) F–10
Extending the Standalone Shell F–10
Sash Commands F–10
cp (copy) F–12

# Chapter 1
# System Overview

## Introduction

This chapter describes the M/120 RISComputer System, which is a 32–bit, multi–user, UNIX system. The M/120 uses the R2000 processor and is based on the MIPS RISC (Reduced Instruction Set Computer) architecture. The M/120 is a high–performance UNIX system, which is suitable for use as a departmental minicomputer or as a compute server in a networked environment.

The M/120 is packaged in an upright enclosure that includes a power supply. A Motherboard supports card slots for the CPU and multiple memory cards that transfer data over a proprietary bus. In addition, there are four IBM PC/AT–compatible card slots for I/O expansion and additional connectivity. The system includes four RS–232C serial I/O ports, an Ethernet controller and port, and a Small Computer Systems Interface (SCSI) controller and port. The main enclosure houses a 5 1/4 inch Winchester disk drive and a cartridge tape drive. Up to five additional disk drives can be added to the system using the side–by–side Expansion Cabinet. Figure 1.1 on the following page shows the physical location of the system components.

The software for the M/120 includes a tailored UNIX V, Release 3 called RISC/os (also known as UMIPS). Included with RISC/os are the assembler, the C and Pascal optimizing compilers, symbolic debugger, linkage editor, and various profiling and development tools. The system tools include such items as an archiver, a build tool, a symbol table, and a disassembler.

RISC/os also includes Sun Microsystem's Network File System (NFS), which is a powerful network file interchange medium for the wide range of systems that support NFS. With NFS, you can integrate the M/120 System into heterogeneous computing environments.

## System Description

The minimum system configuration includes the subassemblies given in the following list. These subassemblies are shown in Figure 1.2 on the following page.

- Motherboard
- Central Processing Unit
- System Memory
- AT Bus Slots
- Packaging
- Peripherals
- Controls, Switches, and Indicators

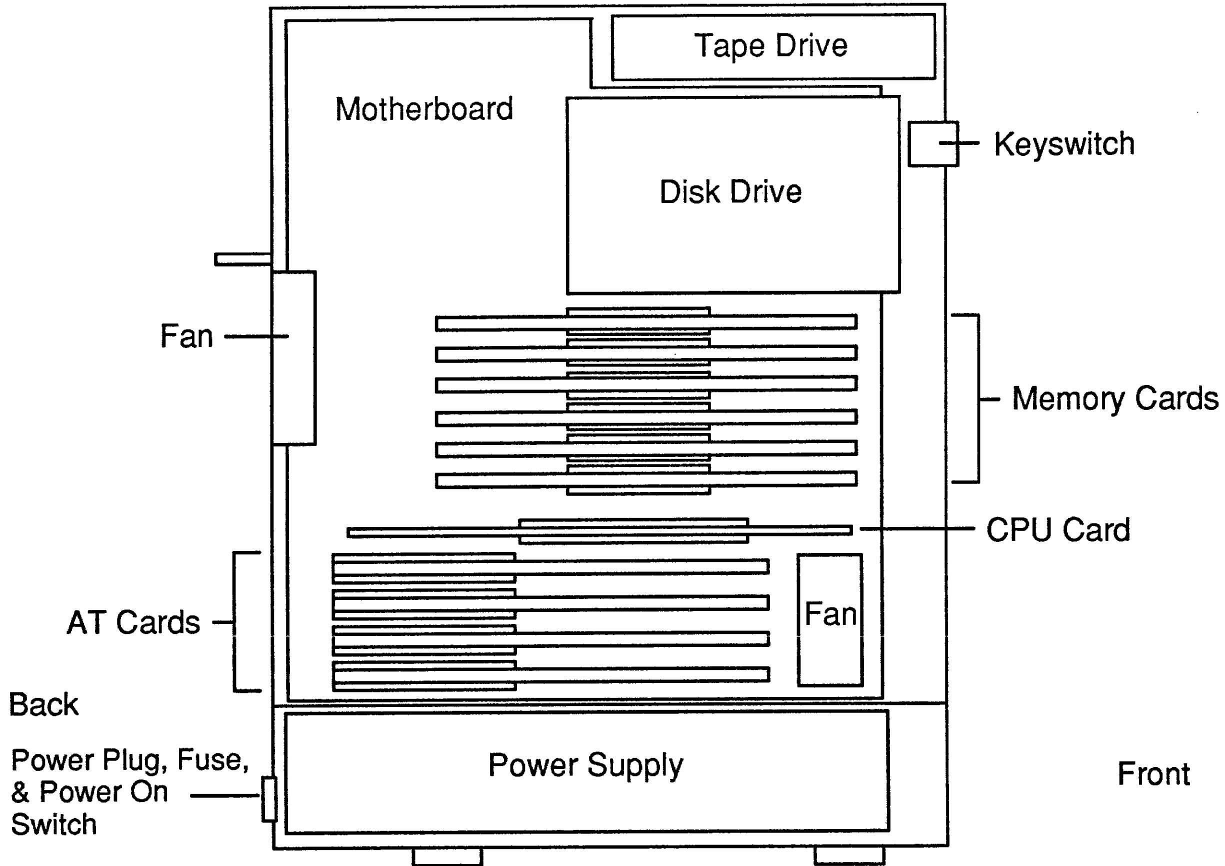



*Figure 1.1. M/120 System Components (Left Side View)*

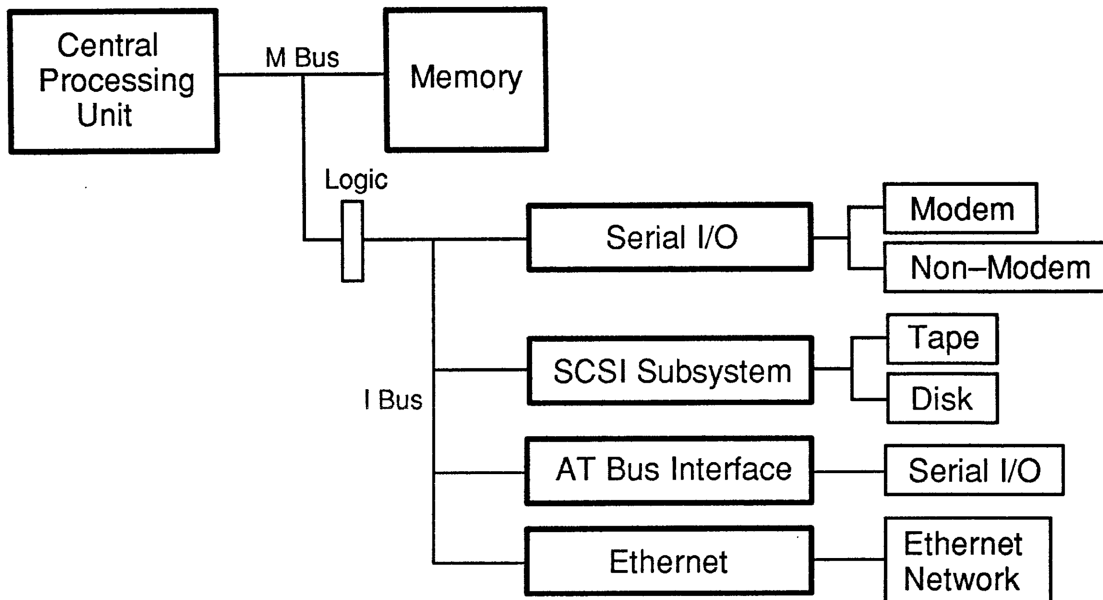



*Figure 1.2. M/120 System Block Diagram*

## Motherboard

The Motherboard contains a Small Computer Systems Interface (SCSI) as defined by the ANSI X3T9.2 committee. This interface supports up to seven target devices in asynchronous, synchronous, or mixed modes with DMA data transfers. The SCSI logic is brought out to an external connector that allows the integration of up to five additional SCSI devices in an Expansion Cabinet.

The Motherboard incorporates logic for the Lance Ethernet controller that supports the IEEE 802.3 Ethernet standard. The Lance Ethernet hardware implementation consists of the Am7990 Local Area Network controller and the Am7992A Serial Interface Adapter. The Ethernet controller functions in a memory–mapped I/O mode for efficient networking operation. This configuration supports full Ethernet only. It does not support Cheapernet.

The main memory controller on the Motherboard supports up to six, 8–megabyte memory array boards, which plug into the Motherboard memory slots.

The PC/AT bus interface on the Motherboard supports both slave and master type AT cards. The M/120 has four full size AT card slots that can be used for adding peripherals to the system. The Motherboard also contains EPROMs for boot code and power–up diagnostics, and system clocks.

The Motherboard contains two DUARTS that support four RS–232C ports. The serial I/O ports (ports 0 – 3) are located on the rear panel of the computer. Refer to Figure 1.3 on the following page. Ports 1 and 3 support full modem control for connecting modems and printers. These ports may be used as a download facility. Ports 0 (zero) and 2 are configured for terminal equipment and are suitable for a console. All four ports are Data Terminal Equipment (DTE) configured. Connection to terminal equipment requires a null modem connector.

## Central Processing Unit

The Central Processing Unit (CPU) is located on a separate printed circuit board module, which plugs into the Motherboard. There are two different CPU modules available: the one used in the M/120–3 has a 12.5 MHz CPU clock speed, and the version used in the M/120–5 has 16.7 MHz CPU clock speed. The CPU module is the core processor in the M/120 System. It includes the R2000 RISC Processor, the R2010 Floating Point coprocessor, a 64K byte Instruction Cache, a 64K byte Data Cache, and the R2020 Write Buffers. Figure 1.4 on the following page is a block diagram of the CPU module.

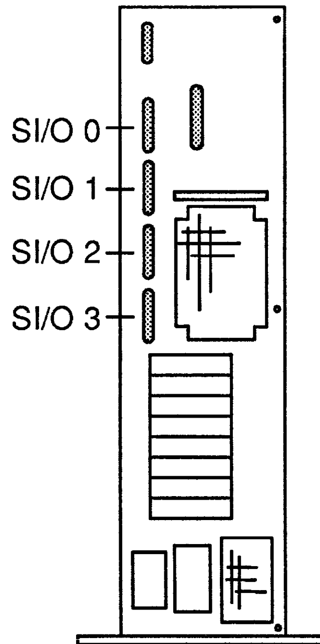



*Figure 1.3. Location of Serial Ports (M/120 Rear View)*

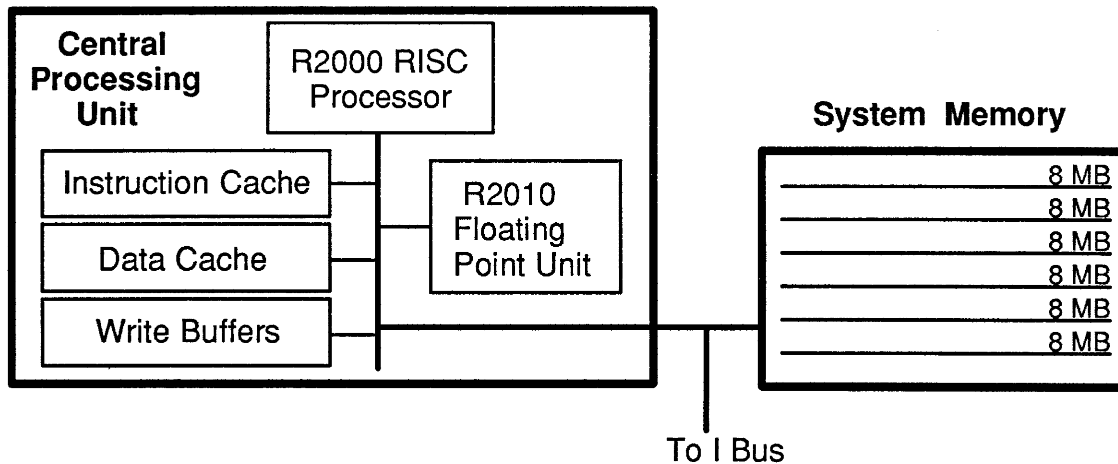



*Figure 1.4. Central Processing Unit*

The R2000 processor is a full–custom, 32–bit CMOS microprocessor based on the Reduced Instruction Set Computer (RISC) technology. The R2000 processor design uses only simple load/store operations for memory access and utilizes large caches for speed and efficiency. The custom CMOS processor combines two tightly coupled units on a single chip. The CPU executes instructions directly without microcode, and the memory management unit provides virtual memory and exception handling mechanisms needed for the efficient support of multi–user operating systems.

The R2010 Floating Point Unit (FPU) is an M/120 option. The CPU Module can be ordered without the Floating Point Unit, but unless specified, the Floating Point Unit is included. The R2010 FPU operates in conjunction with the R2000 processor and extends the R2000's instruction set to perform arithmetic operations on values in floating–point representations. The FPU executes instructions in parallel with the CPU, and most floating point instructions can execute or load/store during the same single cycle instruction executions as the CPU. The R2010 FPU contains sixteen 64–bit registers that each can hold data for single or double precision calculations. The R2010 FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, "IEEE Standard for Binary Floating–Point Arithmetic."

The M/120 CPU Instruction Cache and Data Cache each provide 64K bytes of high–speed memory that allow the processor to operate at maximum speed.

The CPU module incorporates four write buffer devices that enable the M/120 CPU to perform write operations at full speed, avoiding memory write time delays. The write buffers provide 4–deep buffering of 32–bit address and data words. Byte and half–word gathering of data is incorporated to reduce write accesses to main memory for full words where possible.

## System Memory

The main memory controller is contained on the Motherboard and integrates a synchronous high–speed main memory bus with the CPU Module. Peak memory bandwidth is 11 megabytes per second for reads and 16 megabytes per second for writes. This controller logic supports up to six R2450 Memory Cards, which each contain 8 megabytes of memory. Refer to Figure 1.4. The Memory Cards plug into the Motherboard as daughter boards for main memory expansion. The system can be configured to contain from 8 megabytes (minimum configuration) to 48 megabytes (maximum configuration) of system memory. Both the controller and the R2450 cards support byte parity.

Each R2450 Memory Card contains 8 megabytes of memory implemented with 1 Mbit DRAMs, plus the additional DRAMs required for parity memory. The first four R2450 card slots are bus–compatible, identical, and reserved for R2450 Memory Cards only. The last two slots can be used for either R2450 Memory Cards or for special function cards. A special bus master can be plugged into either of these two slots, but not both. The M–bus arbitor supports only one additional master.

The physical address space that is occupied by any given Memory Card is card–slot addressable on 8 megabyte boundaries. Each Memory Card slot is associated with a unique 8 megabyte address space. Therefore, the additional Memory Cards do not require jumper blocks for specifying the base memory address.

## AT Bus Slots

The M/120 features system expandability through the IBM PC/AT Bus Interface. This four slot bus is designed to support virtually any card that works in a PC/AT or compatible machine.

The AT Bus Interface supports both slaves and masters. A PC/AT slave can be accessed directly by the M/120 CPU as a memory mapped I/O device. A slave can transmit data to or receive data from the main memory via a single DMA channel. A PC/AT master can directly write to or read from a one megabyte address space of the M/120 main memory through mapping hardware. For additional information on the AT Bus compatibility considerations, see Appendix A.

The M/120 can access the full 16 megabyte AT address space as mapped into the M/120 main memory. Control registers set the mode of operation on the AT bus.

A large number of vendors offer AT bus controllers. These are normally supplied with MS–DOS, OS/2, or Xenix device driver software. When implementing a selected AT controller in the M/120, the device driver will usually have to be modified or completely rewritten to run on the M/120's R2000 processor with RISC/os, which is the MIPS port of the UNIX operating system. Refer to **Chapter 4, Writing Device Drivers**, for additional information.

## Packaging

The 400 watt power supply for the M/120 is mounted at the bottom of the system enclosure. It is switch selectable for worldwide AC power type compatibility. Air flow is from front to rear, with internal baffles to provide even cooling.

AT card cutouts are provided so that normal I/O connections can be made to them from the lower rear part of the system cabinet. Connectors for the four integral console and serial I/O ports, Ethernet, and the SCSI bus are also located on the rear panel.

One 5 1/4 inch SCSI disk is mounted in the main system cabinet. A 120 megabyte cartridge tape is also mounted in the cabinet at the top for operator convenience. The left side exterior panel is removable for access to all interior parts.

## Peripherals

There are two different Disk Drive capacities available for the M/120: a 328 megabyte Disk Drive and a 156 megabyte Disk Drive. The base configuration consists of a 328 megabyte formatted 5 1/4 inch, embedded SCSI Disk Drive. The 328 megabyte Disk Drive has an average access time of 16.5 ms and supports synchronous transfer rates to 4 megabytes per second. The 328 megabyte Disk Drive also incorporates a read look–ahead algorithm that maximizes sequential UNIX file system read performance. The drive consumes 27 watts of power (steady state) and its actuator and spindle motor design provide quiet operation.

For applications requiring a smaller capacity, the M/120 is also available with a 5 1/4 inch Disk Drive that provides 156 megabytes of formatted storage. This Disk Drive has an average access time of 16.5 ms and has an asynchronous SCSI bus transfer rate of 1.25 megabytes per second. The 156 megabyte Disk Drive consumes 27 watts of power.

A Quarter Inch Cartridge (QIC) Tape Drive for software distribution and backup purposes is also included in the base M/120 configuration. The standard Tape Drive for the M/120 is the high capacity 120 megabyte configuration (QIC–120). A 60 megabyte configuration supporting the QIC–24 format (QIC–11 read only) is also available. Both Tape Drives have an embedded SCSI interface that operates at 1.25 megabytes per second asynchronous. The Tape Drives are 5 1/4 inch half–height units that support 90 kilobytes per second sustained while streaming. Power consumption for both Tape Drives is 25 watts while in operation. For information on the Tape Drive operation and for preventative maintenance instructions, see **Appendix B.**

## Controls, Switches, and Indicators

The M/120 controls, switches, and indicators are listed below.

- Keyswitch
- Power On Switch
- Power On LED
- Disk Drive LED
- Tape Drive LED
- Head Loading Lever

The **Keyswitch** has three positions: lock, unlock, and reset. The key is removable in both the lock and the unlock positions. The three keyswitch positions are defined as follows.

Lock — When the key is in the bottom or lock position, the system cannot be booted if the power is on and the system is at the PROM Monitor. This position also prevents the system from being shutdown and places restraints on operating system Run state transitions. Refer to the Telinit man page, **telinit (1m)**, for additional information.

Unlock — When the key is in the middle or unlock position, the system can be booted when the power is on and the system is at the PROM Monitor prompt. The system can also be shut down when the key is in the unlock position and root or the superuser is the initiator of the shutdown.

Reset — The top position is a momentary switch and causes a system reset. The Power On Diagnostics are not run when this reset is used.

The **Power On Switch** is located on the rear panel of the computer adjacent to the power cord connection. When the Power On Switch is set to the ON position while the Keyswitch is in th Unlock position, then the Power On diagnostics are performed. (If the bootmode variable is

not set to *d*, then the Power On diagnostics are not performed. See **Chapter 5, PROM Monitor** for a discussion of the boodmode variable.)

There are three LEDs on the front panel of the computer: the Power On LED, the Disk Drive LED, and the Tape Drive LED. The Power On LED indicates that the power is on. The Disk Drive LED flickers when the Disk Drive is active. The Tape Drive LED comes on when a cartridge is inserted and the tape is not at the load point or end of tape. The Tape Drive LED stays on until the tape is rewound to the Load Point by a REWIND command. Normally, the cartridge should only be removed when the LED is off.

The Head Loading Lever locks the tape cartridge in place and loads the tape heads. This lever also moves the heads away from the tape and ejects the cartridge tape.

## Expansion Cabinet

Additional SCSI devices can be supported in an Expansion Cabinet that has the same physical dimensions and general appearance as the M/120 base system cabinet. The Expansion Cabinet is connected using the SCSI port and cabling. Up to five full height 5 1/4 inch disks or equivalent size SCSI devices may be configured in an Expansion Cabinet. Refer to Figure 1.5. A section of removable front trim on the Expansion Cabinet is provided so that a removable media device may be installed and accessed. The Expansion Cabinet contains a 400 watt power supply that supports a full complement of installed disks.

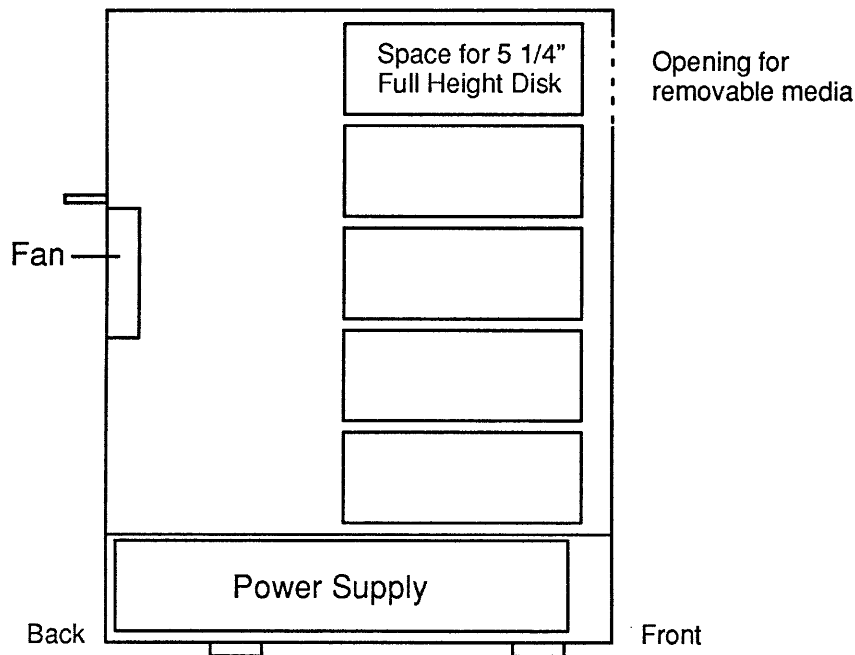



*Figure 1.5. Side View of Expansion Cabinet*

# Specifications

| General | |
|---|---|
| CPU Type | MIPS R2000 |
| Word length | 32 bits |
| FPU type | MIPS R2010 |
| Minimum Memory | 8 Megabytes |
| Maximum Memory | 48 Megabytes |
| Memory Configuration | 8 MB/slot, 6 slots |
| Virtual Address Space | 4GB, 2GB/process |

| **SCSI Bus** | ANSI X3.131–1986 |
|---|---|
| Max transfer rate | 4.0MB/second sync |
| Target devices | Up to 7, sync or async |

| **Ethernet Port** | IEEE 802.3, standard |
|---|---|
| Media type | Coaxial cable |
| Data rate | 10 Mbits per second |
| Access control | CSMA/CD protocol |

| **Serial I/O** | RS–232C, DB–25S |
|---|---|
| In base system | 4 ports (2 w/ modem control) |
| Using AT Bus | 8 lines per slot |
| AT Bus slots | 4 |
| Max baud rate | 19,200 |

| **Disk Drive Type** | 5.25" full height |
|---|---|
| Controller | SCSI, integral |
| Recording type | Winchester |
| Capacities | 328MB formatted<br>156MB formatted |
| Average seek time | 16.5 ms |
| Average latency | 8.3 ms |
| Average access time | 24.8 ms |
| Data command overhead | 0.75 ms for 328MB<br>2.0 ms for 156MB |
| Power consumed | 27 watts (92BTU/hr) |
| Disk drive MTBF | 40,000 hours |
| Configurations | 1 in base cabinet<br>up to 5 in Expan Cab |
| Weight | 3.7 kg (8 lbs) each |

| **Cartridge Tape Drive** | 5.25" half height |
|---|---|
| Controller | SCSI, integral |
| 1/4" tape capacity<br>optional | 120MB (QIC–120)<br>60MB (QIC–24) |
| Operating speed | 90 ips, streaming |
| Power consumed | 25 watts in use |
| Tape drive MTBF | 12,000 hrs @25% duty |
| Configuration | 1 in base cabinet |

**Dimensions, Weights, and Power**

| | Base System | Expansion Cabinet |
|---|---|---|
| Height | 58.5cm (23.0") | 58.5cm (23.0") |
| Width | 18.8 cm (7.0") | 18.8cm (7.0") |
| Depth | 45.7cm (18.0") | 45.7cm (18.0") |
| Weight | 25Kg (55 lbs) | 20 lbs (empty) |
| Shipping weight | 30kg (65 lbs) | 46kg (100 lbs) |
| Power supply | 400 watts | 400 watts |
| AC circuit rating | 1000 volt amps | 1000 volt amps |
| Heat in BTUs/hour | 2040 800 watts | 340 empty |

| Regulatory | |
|---|---|
| RFI emissions | FCC Class A, VDE Class A |
| Safety | UL, CSA, TUV, VDE, IEC |

| Environmental | |
|---|---|
| Ambient temp, op | 10° C to 40° C |
| Relative humidity | 10% to 80% non–c |
| Altitude | to 3000m (10,000 ft) |
| AC voltage | 90 to 130 vac or 180 to 264vac |
| AC frequency | 47 to 63 Hz |
| AC pwr cord length | 2m (6 ft) |

# Chapter 2
# Installation

This chapter contains the installation instructions for the M/120 RISComputer System. This chapter is divided into the following sections:

- Select the Site
- Select the Voltage
- Install Additional or Optional PC Cards
- Install Serial I/O Devices
- Cable the System to an Ethernet Network
- Power Up the M/120 System.

## Select the Site

When selecting a site for the M/120 computer, the important considerations are the space requirements, the power requirements, and the environmental requirements.

### Space Requirements

This section describes the physical dimensions of the M/120 and the minimum floor area required. The outside dimensions for the M/120 are shown in Figure 2.1 on the following page.

Since the M/120 does not have any swinging doors, the additional floor space required for an installed system is minimal compared to the actual physical dimensions. A small amount of space is required in back of the computer to allow for cable clearance. Table 2.1 on the following page summarizes the floor requirements needed for an installed M/120 System. The M/120 has a removable side panel that allows access to the inside of the computer system. Due to the compact design and portability features (53 lbs), space requirements for service and maintenance are not provided.

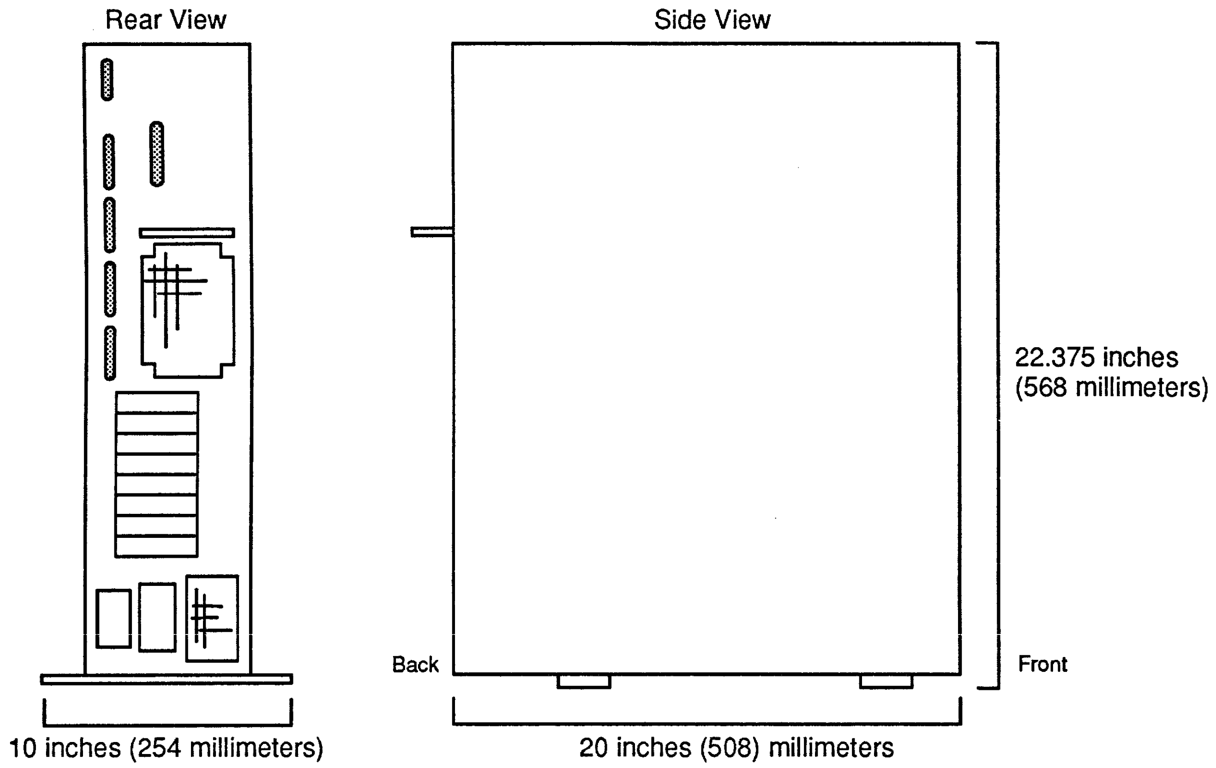



*Figure 2.1. M/120 Physical Dimensions*

**Table 2.1. Minimum Required Area**

| Dimension | Inches | Millimeters |
|---|---|---|
| Height | 22.375 | 568 |
| Width | 10 | 254 |
| Depth | 24 | 609 |

## Power Requirements

This section describes the voltage requirements, grounding, noise suppression, and types of power connectors.

The M/120 has a switchable power supply that requires 115 VAC or 230 VAC. The voltage is preset at the factory based upon the purchase order. If you need to change the preset factory setting, instructions are provided later in this chapter. Table 2.2 on the following page lists the voltage requirements.

**Table 2.2. Voltage Requirements**

| VAC (Volts AC) | HZ (hertz) | A (amps) | Range |
|---|---|---|---|
| 115 | 50/60 | 6 | 90 – 132 VAC, 50/60 Hz |
| 230 | 50/60 | 3 | 180 – 264 VAC, 50/60 Hz |

The AC power connector mounted on the rear panel of the computer system is a standard 3–conductor connector. The M/120 includes a 3–conductor power cord, which is terminated with a 3–conductor plug. For 230V (220–240) operation, the power cable should be terminated with the proper outlet connector for your local power source.

A green ground wire is connected to the metal frame of the system cabinet. This safety ground protects personnel against short circuits and other malfunctions. In order for this protective ground to work, the power cord must be plugged into an outlet that has a ground connection. The outlet ground connection must be connected to the distribution panel where the system's circuit breaker is installed.

The grounding wires for the outlets that are to be used by the M/120 and its peripheral equipment must be connected to the same ground wire (separate from neutral) at the distribution panel. A grounding wire should be installed from the distribution panel to earth ground. The earth ground could be the structural steel of the building, a ground rod, or a building entrance earth ground connection. All grounding wires should be insulated, and conduit must not be used as a ground path.

If specific protection against lightning is needed, consult Article 280 of the National Electrical Code. Article 280 describes the installation of lightning arrestors on power and communication lines.

Electromagnetic interference (noise), which can cause computer malfunctions, can be placed onto power distribution circuits by office equipment, janitorial equipment, electric motors, etc. To eliminate or to reduce the noise to an acceptable level, the computer and its peripherals must be provided with separate circuit breakers from those used by other electrical equipment.

## Environmental Requirements

An environment that meets the specifications given in Table 2.3 must be provided or created in order for the system to operate properly.

**Table 2.3. Environmental Specifications**

| Conditions | Temperature | Humidity |
|---|---|---|
| Maximum Operating | 50–104 degrees F (10–40 degrees C) | 20% – 80% non–condensing |
| Recommended Operating | 59–82 degrees F (15–28 degrees C) | 50% |
| Maximum Storage | –40 to 149 deg. F (–40 to 65 deg. C) | 95% non–condensing |

If the system is moved from one environment to another, then it is recommended that the equipment is not powered on until the system has had time to acclimatize to the new environment. Wait one hour for every 10 degree C increment of change that occurred before powering up the system.

Excessively high humidity levels can cause improper operation of disk drives and of paper–handling peripherals (printers). Excessively low humidity levels can increase problems with static electricity.

The discharge of static electricity from personnel can damage equipment, cause errors in system operation, and damage the contents of software media. To prevent damage from static electricity, use ground mats connected to earth ground around the computer. These mats dissipate accumulated static charge.

The M/120 is equipped with cooling fans to circulate environmental air throughout the cabinet. The total system heat dissipation is 800 watts (approximately .8 BTU per second).

If the system is to operate continuously, you must determine if the air conditioning is turned on or off during weekends and off–work hours. If the operation of the air conditioning is varied, then measurements should be taken during the down or off periods to verify that the temperature range of the system is not exceeded. If the operating temperature is exceeded, then additional air conditioning must be provided, or, as a last resort, the system must be shut down.

## Select the Voltage

The voltage is preset at the factory according to the purchase order. If the voltage selection needs to be changed, then use the following procedure. Changing the voltage selection is a two–step process. The two fuses must be changed and the voltage select switch must be toggled. Proceed as follows.

1. Turn the M/120 System power off.
2. Remove the power cord from the rear panel of the computer. Access to the fuses cannot be obtained without removing the power cord.
3. Snap the fuse cover out of the machine using a flat–head screwdriver. Refer to Figure 2.2.
4. Pull the fuse holder and cover out of the computer.
5. Slide the fuse holder out of the fuse cover, and remove the two fuses.

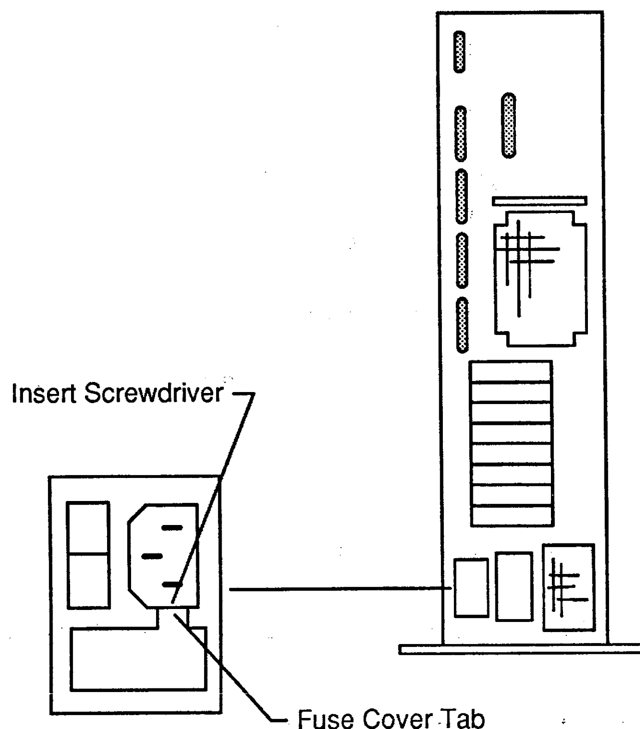



*Figure 2.2. Removing the Fuse Holder*

6. Position each fuse on the fuse holder, and press each fuse into place. For 115 VAC, install two 6 amp fuses (Little Fuse 312–006, 3AG). For 230 VAC, install two 3 amp fuses (Little Fuse 312–003, 3AG).
7. Slide the fuse holder back into the fuse cover.
8. Slide the fuse assembly back into the machine until it snaps into place.
9. Remove the two screws from the metal plate containing the CAUTION note to gain access to the voltage select switch. This cover plate is located next to the power socket on the lower edge of the rear panel. Refer to Figure 2.3.
10. Use Figure 2.3 below to locate the voltage select switch on the end of the power supply.

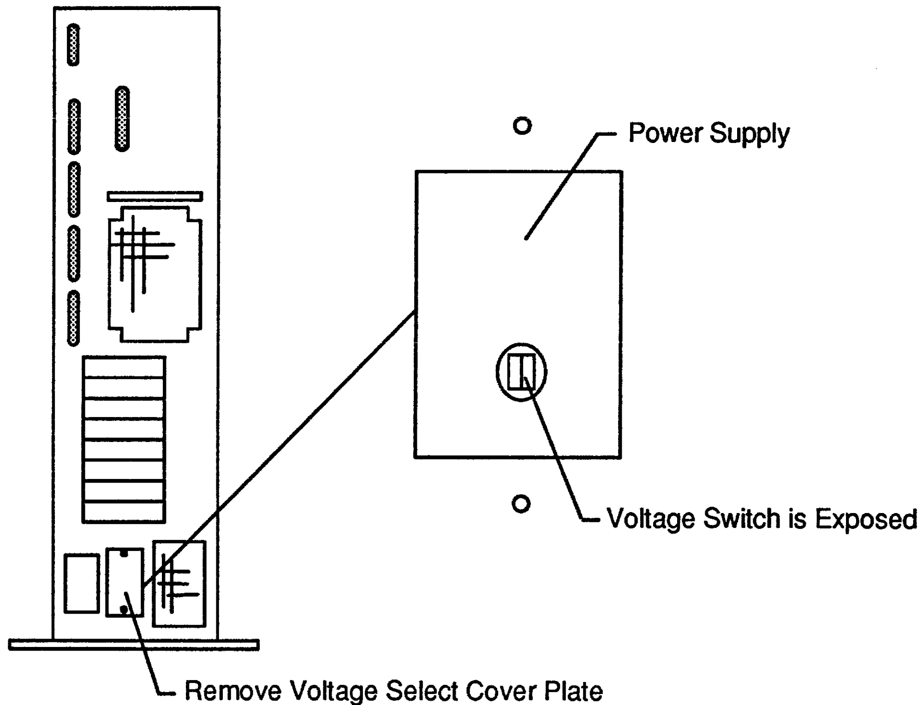



*Figure 2.3. Location of Voltage Select Cover Plate and Voltage Switch*

11. Toggle or slide the voltage switch to the other voltage selection using a flat–head screwdriver. The voltage switch is labeled with 115V and 230V. When selecting a voltage, toggle the switch so that the voltage you want to select is displayed on the switch.
12. Reinstall the voltage select cover plate by reinstalling the two screws.

Do not install the power cord at this time.

# Install Additional or Optional PC Cards

This section provides instructions for removing the side panel of the computer and gaining access to the inside of the computer. This section also includes information on installing additional memory cards in the computer and information on installing optional AT Cards.

## Removing the Side Panel

1. Turn the power switch off, and disconnect the power cord from the computer.
2. Remove the three screws along the right edge of the rear panel. Refer to Figure 2.4.

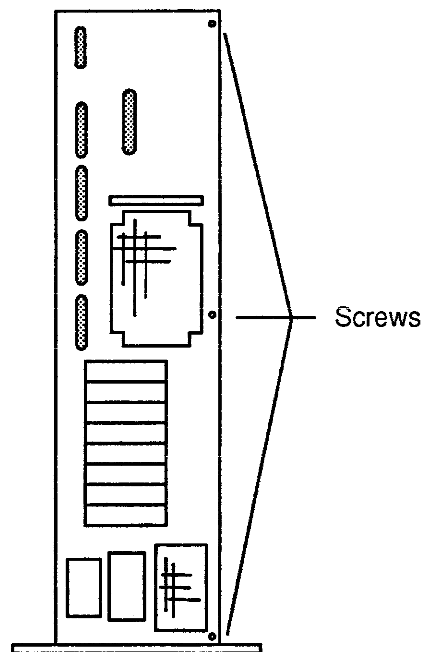



*Figure 2.4. Side Panel Screws*

3. Slide the side panel towards the back of the machine about one inch. Then, lift the side cover from the machine. The latching tabs located on the top and bottom, front edges of the side panel must clear the other half of the latching assemblies before the side panel can be lifted from the computer.

## Install Additional Memory Cards

Up to six R2450 Memory Cards can be installed in the M/120 RISComputer System. The basic system includes one Memory Card, which is installed in the top slot. Additional Memory Cards must be sequentially added from the top slot down. If you only install two Memory Cards, then the cards would be installed in the top two slots. Figure 2.5 shows the order that Memory Cards must be added to the system.

To install a Memory Card in the computer, position the card with the component side up, and insert the card in the card guide on the front side of the cabinet. Slide the card into the connector on the Motherboard until firmly seated.

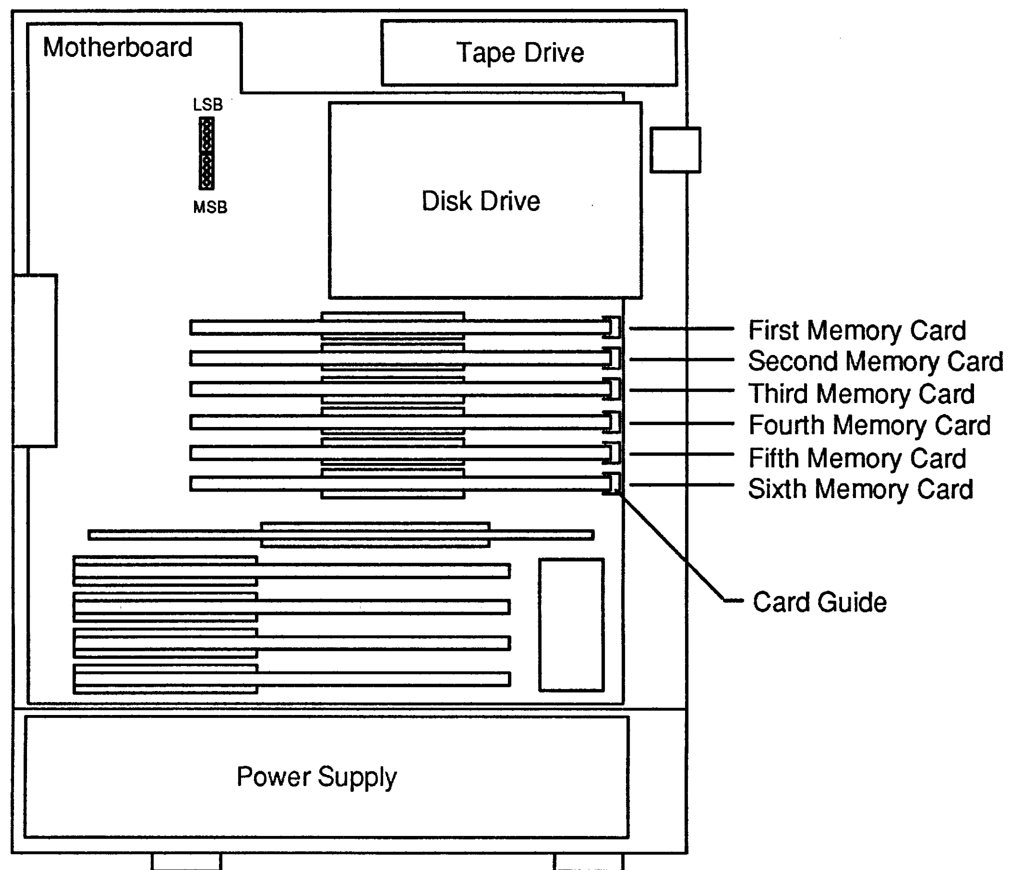



*Figure 2.5. Location of Memory Card Slots*

## Install Optional AT Cards

Up to four AT Cards can be installed in the M/120 to extend the capabilities of the system. Not all AT Cards are compatible, and therefore cannot be plugged into the AT slots. For detailed specifications on the AT Bus Compatibility Considerations, see **Appendix A**. A UNIX driver must be written for each AT Card that is installed in the M/120. Information on writing a UNIX device driver can be found in **Chapter 4, Writing Device Drivers**. Use the following procedure to install an AT Card.

1. Remove the Expansion Slot cover for the slot in which you want to install an AT Card by removing the screw. The four AT card slots are shown in Figure 2.6.
2. Position the AT card with the mounting bracket on the left side (towards the back of the computer), and carefully slide the card into the connector on the Motherboard.
3. Reinstall the screw removed in step one to secure the AT Card and mounting bracket .

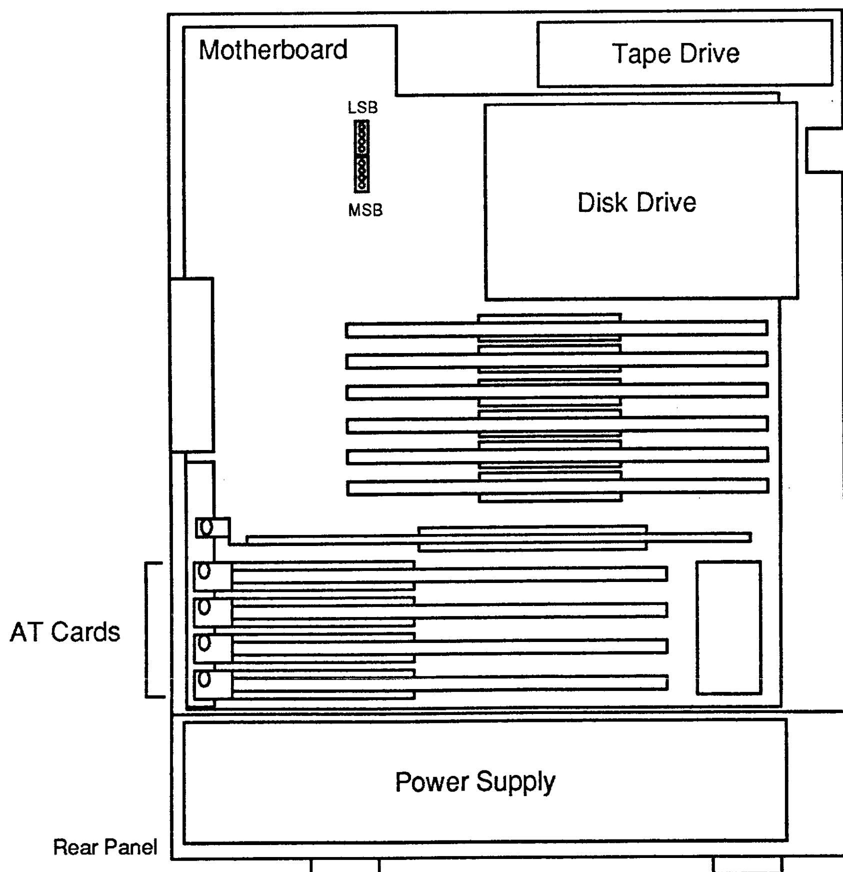



*Figure 2.6. Location of AT Card Slots*

## Reinstall the Side Panel

1. Align the latching tabs on the side panel with the slot at the top and the slot at the bottom of the computer chassis. See Figure 2.7.
2. Press the latching tabs into the two slots, and slide the side panel towards the front of the machine.
3. Reinstall the three screws on the rear panel that secure the side panel.

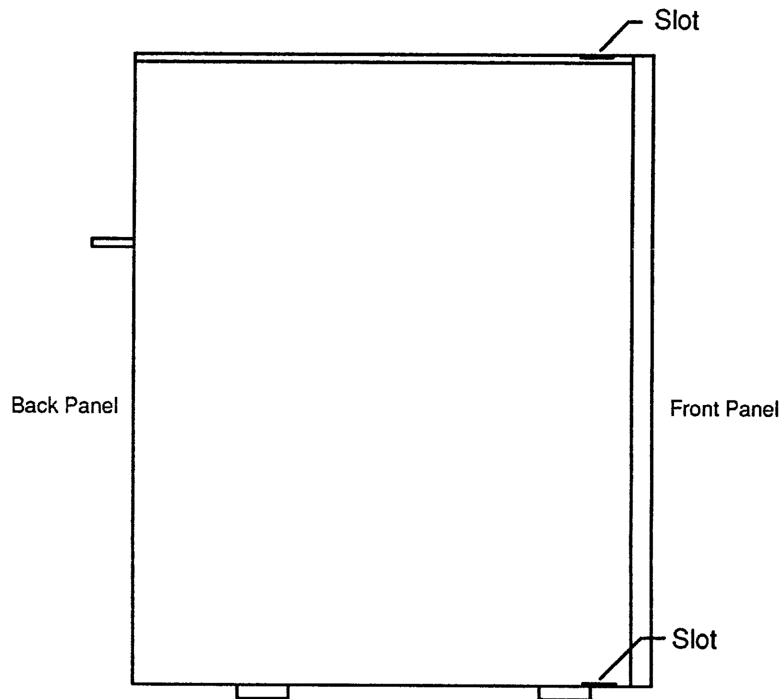



*Figure 2.7. Location of Latching Tab Slots*

# Connect Serial I/O Devices

This section provides the technical information needed in order to connect external equipment to the serial (RS–232C) I/O ports.

There are four serial ports located on the rear panel of the computer, which are shown in Figure 2.8. Serial I/O ports 1 and 3 support modem control for connecting modems and printers. Ports 0 (zero) and 2 are configured for terminal equipment. All ports are DTE (Data Terminal Equipment) configured. Connecting terminal equipment requires a null modem connector.

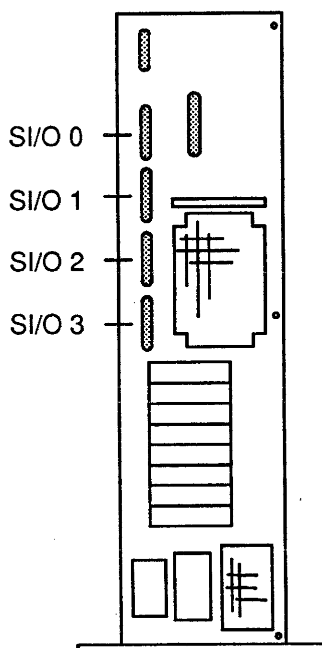



*Figure 2.8. Location of Serial Ports*

## General Considerations

The standard for RS–232C cables recommends that cables should not be longer that 50 feet. Longer cables may be used, but a longer cable may create line noise, which would affect the data and cause errors. If a cable longer than 50 feet is required, then an appropriate extender device should be used.

FCC regulations on EMI (Electromagnetic Interference) require the use of shielded cable. For best results, connect the I/O panel via a metallic connector hood and jackscrews. For terminal equipment, do not connect the shield to any other pins on the RS–232C connector.

If a cable must be disconnected from a peripheral device, then disconnect the cable from the rear panel of the computer. Improper termination of cables can reduce the system speed and throughput.

## Terminal Connector Pinouts

All four ports can be connected to terminals. However, ports 0 (zero) and 2 are specifically intended for this function.

The number of wires or pins used in building a cable depends on the application of the port being used. If the port is intended for a standard terminal, then use the connections shown in Figure 2.9. If the port is to be used for a modem adapter, then use the connections shown in Figure 2.10.

**NOTE**

The connections shown in Figures 2.9 and 2.10 are general connections. Your configuration and equipment may require different adapter connections. Refer to your peripheral manual for specific requirements.

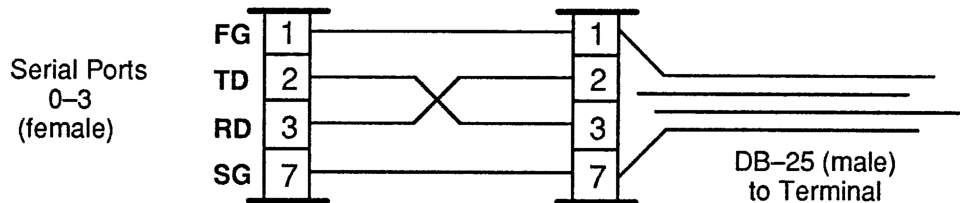



*Figure 2.9. Connections for a Terminal Adapter*

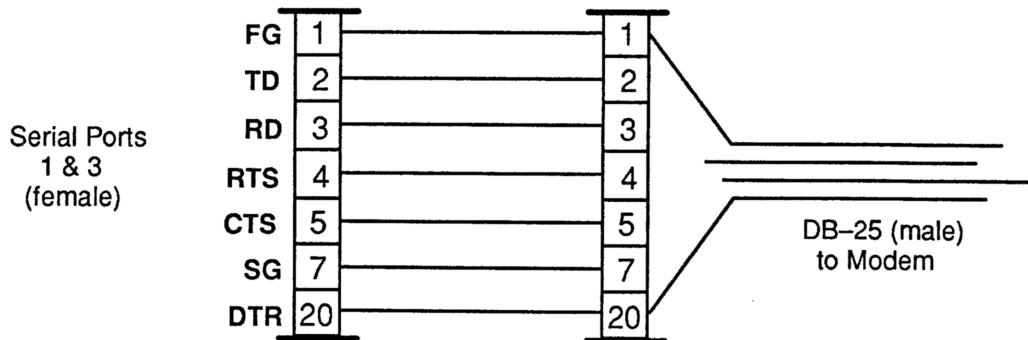



*Figure 2.10. Connections for a Modem Adapter*

## Connecting the Console

The cable for connecting the terminal you have designated as the system console may have been provided with the peripheral equipment. If not, then a cable may be available from the distributor from whom you purchased the peripheral equipment. Use the following procedure to connect the system console.

1. Connect the RS–232C cable to the connector labeled SI/O 0 (zero) on the rear panel of the M/120 computer. Tighten the connector screws to secure the cable connector.

2. Connect the other end of the console cable to the terminal you have designated as the system console.

3. Connect the system console power cord to a power source.

4. Turn on the system console, and set the console terminal parameters as specified below using the SETUP mode as described in the documentation for your terminal.
   - Baud rate = 9600 baud (unless the system default baud rate has been changed)
   - Bits per character = 8
   - Protocol = XON/XOFF
   - Parity = Disabled
   - Mode = Character (not block) and full duplex
   - Stop bits = 1

## Connecting Other Serial I/O Devices

Cables for connecting serial I/O devices may have been provided with the peripheral equipment. If not, then cables may be available from the distributor from whom you purchased the peripheral equipment. Standard RS–232 cables can also be obtained from almost any supplier of computer supplies.

Before cabling the system, verify that the system is alive and that all cards were installed correctly by powering up the computer system. Proceed as follows to power up the computer system and to cable the system.

1. Set the power switch on the rear panel of the computer in the off position, and then connect the computer power cord to a power source.

2. Turn the keyswitch to the Unlock position, and set the system power switch to the ON position. The Power On diagnostic and system information messages will be displayed on the system console. The PROM Monitor prompt (>>) is displayed on the system console after the Power On diagnostic and system information messages are displayed. (If the Boot-

mode variable described in **Chapter 5, PROM Monitor** is set to *d*, the system performs a different start up sequence. Refer to Chapter 5 for details.)

3. Power down the console and computer system, and unplug all power cords.
4. Connect the serial I/O device cables to the rear panel of the computer. Tighten the two screws on each cable connector.
5. Connect the other end of each cable to the correct peripheral, and tighten the connector screws. Refer to each peripheral device manual for additional installation instructions.

# Cable the System to an Ethernet Network

Use the following information and your Ethernet equipment manuals to install your Ethernet network.

The Ethernet port is located on the top, left side of the rear panel as shown in Figure 2.11. The pinouts for the Ethernet port are given in Table 2.4.

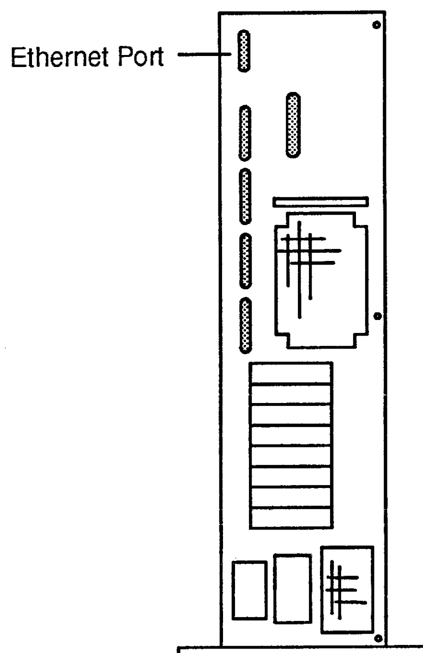



*Figure 2.11. Location of Ethernet Port*

**Table 2.4. Rear Panel Connector to Transceiver Cable Pin Signals**

| Pin | Signal Name |
|---|---|
| 1 | Shield |
| 2 | Collision Presence + |
| 3 | Transmit + |
| 4 | Ground |
| 5 | Receive + |
| 6 | Power Return (Ground) |
| 7 | Reserved |
| 8 | Reserved |
| 9 | Collision Presence – |
| 10 | Transmit – |
| 11 | Reserved |
| 12 | Receive – |
| 13 | Power (+ 12v fused) |
| 14 | Reserved |
| 15 | Reserved |

# Power Up the M/120 System

**Note**

If you have an Expansion Cabinet that needs Disk Drives or other peripheral devices installed, then turn to **Appendix C** and complete the instructions given before continuing.

1. Connect any disconnected interface cables to the associated peripheral equipment.

2. Connect any power cords for attached peripherals including the console to a power source.

3. Verify that the power switch on the rear panel is in the OFF position, and connect the power cord for the computer to a power source. It is recommended that a single power line be dedicated to the computer.

   The AC power connector is a standard, 3–prong AC power receptacle. Use an AC power cable that is rated at 10 Amperes at 250 volts AC. MIPS supplies power cables for most North American users. For users who need power cables for other types of outlets, contact your local dealer or distributor.

4. Turn on the system console, and wait for the cursor to appear.

5. Turn the keyswitch to the Unlock position.

6. Press the Power On switch located on rear panel of the M/120 RISComputer. The Power On diagnostics will be performed. For additional information on the Power On diagnostics, refer to **Appendix D**. The Power On diagnostic messages and the system information messages are displayed on the system console. The PROM Monitor prompt shown below is displayed on the system console after these messages:

   **>>**

   (If the Bootmode variable described in **Chapter 5, PROM Monitor** is set to *d*, the system performs a different start up sequence. Refer to Chapter 5 for details.)

7. Enter "auto" at the PROM Monitor prompt as shown below if you want to boot the Operating System. If you want to use the PROM Monitor, then refer to **Chapter 5, PROM Monitor** for a description of each PROM Monitor command.

   **>>auto <Enter>**

   Turn to the *System Administration Guide* for additional information and instructions on booting the Operating System.

# Chapter 3
# Programming Model

This chapter provides a programmer's view of the M/120 System. It defines the system memory map supported by the M/120, describes the interrupt system supported by the system, and specifies the purpose of the system's configuration and status registers. The chapter also summarizes the addresses and functions of programmable registers provided by the devices used to implement the M/120's I/O subsystem. Figure 3.1 shows the organization of the M/120 and the major programmable elements within the system.

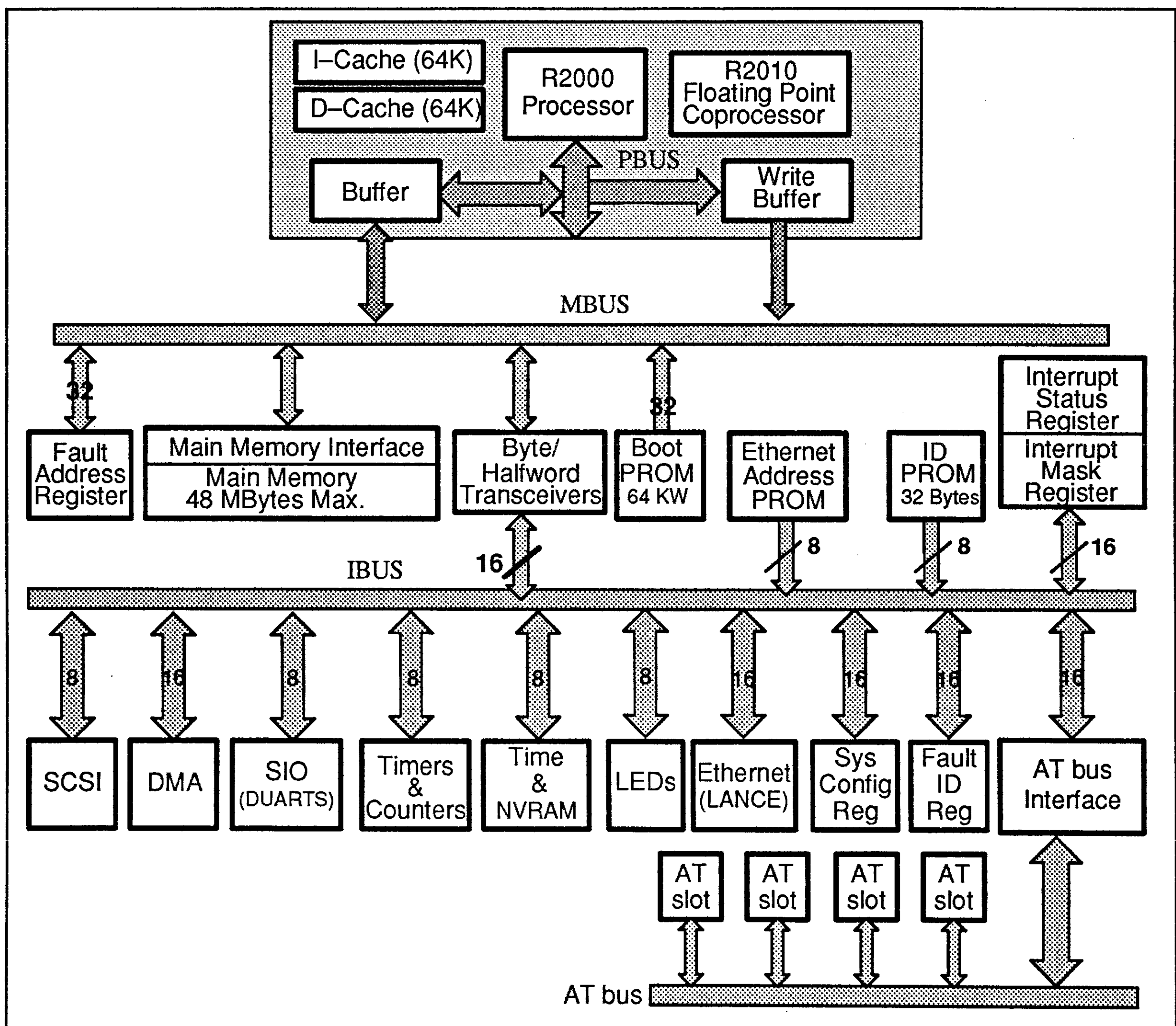



*Figure 3.1 M/120 System Block Diagram*

# Signal and Bit Naming Conventions

Throughout this manual, the names of bits in control and status registers or signal names follow the following rules:

- Register bits whose names end with the letter *B* or with an asterisk (*) are true or asserted when they are set to "0". All other bits are true or asserted when they are set to "1". For example, the bit named TimeOut is true (indicating a timeout condition) when it is set to "1" , and the bit named DAckEnB is true (indicating that the DAck signal is enabled) when it is set to "0".
- Signals whose names end with the letter *B* or with an asterisk (*) are true or asserted when they are at a logic "0" or low voltage level. All other signals are true or asserted when they are at a logic "1" or high voltage level. For example, the signal named Intr4* is true (indicating an interrupt condition) when it is at a logic "0" level.

# Data Formats and Addressing

The M/120 uses an R2000 processor as its central processing unit. The R2000 defines a 32–bit word, a 16–bit half–word, and an 8–bit byte. The system is configured as a **big–endian** system: byte 0 is always the most significant (leftmost) byte, thereby providing compatibility with MC 68000® and IBM 370® conventions.

Figure 3.2 shows the ordering of bytes within words and the ordering of words within multiple–word structures for the M/120.

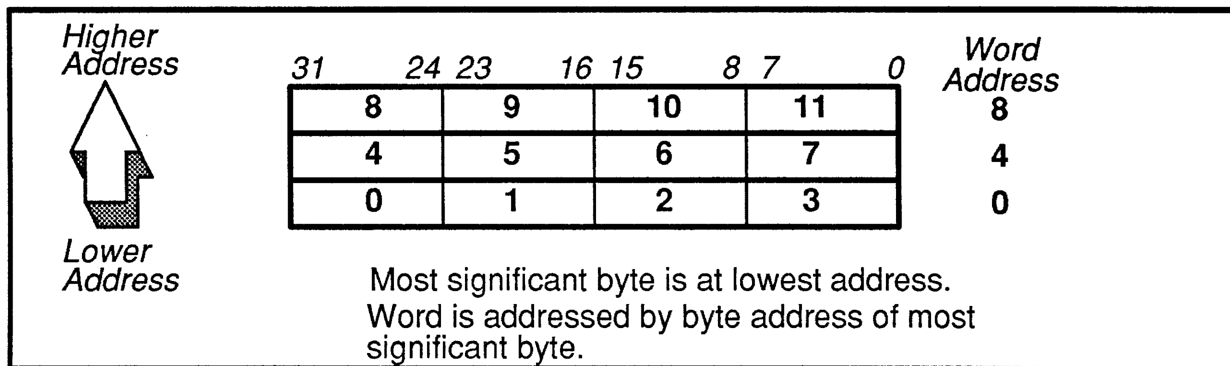



*Figure 3.2 Addresses of Bytes within Words*

The R2000 uses byte addressing, with alignment constraints, for half–word and word accesses. half–word accesses must be aligned on an even byte boundary, and word accesses must be aligned on a byte boundary divisible by four.

As shown in Figure 3.2, the address of a multiple–byte data item is the address of the most–significant byte.

Special instructions are provided for addressing words that are not aligned on 4–byte (word) boundaries. These instructions are Load Word Left/Right (LWL, LWR) and Store Word Left/Right (SWL, SWR). These instructions are used in pairs to provide addressing of mis–aligned words with one additional instruction cycle over that required for aligned words. Figure 3.3 shows the bytes accessed when addressing a mis–aligned word with a byte address of 3 for each of the two conventions.

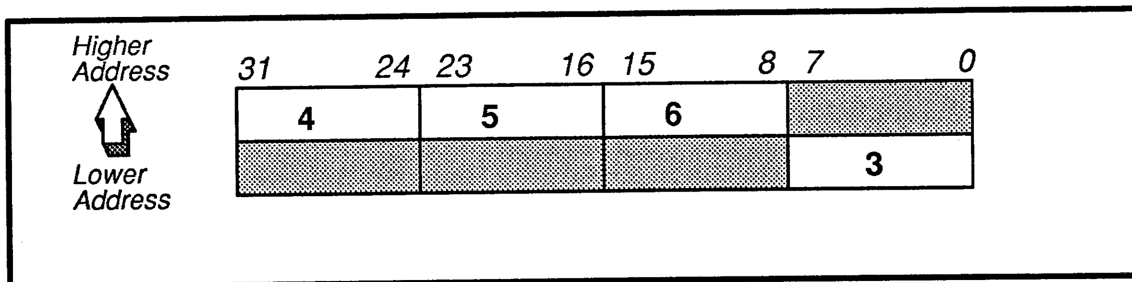



*Figure 3.3 Mis–aligned Word: Byte Addresses*

# System Memory Map

Figure 3.4 shows the physical memory map that is defined by the M/120, and Figure 3.5 provides more detail on the address assignments within the address space assigned to local I/O. The four GBytes of address space is allocated into blocks devoted to main memory, AT bus memory and I/O devices, local I/O devices (timers, time–of–day clock, serial port DUART, and so on), PROMS, and AT bus interrupt cycle vectors. Additional details on address assignments within each of these blocks is provided later in this chapter.

The M/120 memory map provides for up to 128 Mbytes of contiguous main memory in the physical address space from 0x0000_0000 through 0x07ff_ffff. However, the maximum specified "real" main memory that can be installed in the six available slots is 48 Mbytes and occupies a contiguous physical address space from 0x0000_0000 through 0x02ff_ffff. The main memory address space is partitioned into six equal sized 8 MByte address spaces. The physical address space occupied by a memory board is slot specific on 8 MByte boundaries; each unique memory card slot is responsible for a unique 8 MByte address space. This approach eliminates the need for any "address bank" select jumpers or similar mechanisms.

The first four memory board slots (0–3) are bus–compatible, identical, and reserved for R2450 Memory Boards ONLY. The last two slots (4–5) can be used for either R2450 Memory Boards or for special function cards than may require the capability of operating as bus masters. Note that only one of these two slots can contain a card acting as bus master.

Parity errors generated by an R2450 Memory Board can be "bypassed" by software (via the *System Configuration Register*) to facilitate debug and diagnostics.

| Address Range | Assigned to | Number of bytes |
|---|---|---|
| 0x1fff ffff<br>0x1f00 0000 | Boot PROM | 16 Megabytes |
| 0x1eff ffff<br>0x1e00 0000 | ID PROM | 16 Megabytes |
| 0x1dff ffff<br>0x1d00 0000 | Ethernet PROM | 16 Megabytes |
| 0x1cff ffff<br>0x1800 0000 | Local I/O<br>(see Figure 3.5) | 96 Megabytes |
| 0x17ff ffff<br>0x1000 0000 | PC AT I/O<br>&<br>PC/AT Memory<br>(see Figure 3.21) | 128 Megabytes |
| 0x0fff ffff<br>0x0800 0000 | unused | 128 Megabytes |
| 0x07ff ffff<br>0x0300 0000 | reserved | 80 Megabytes |
| 0x02ff ffff<br>0x0280 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #6) |
| 0x027f ffff<br>0x0200 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #5) |
| 0x01ff ffff<br>0x0180 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #4) |
| 0x017f ffff<br>0x0100 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #3) |
| 0x00ff ffff<br>0x0080 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #2) |
| 0x007f ffff<br>0x0000 0000 | R2450<br>Main Memory | 8 Megabytes<br>(slot #1) |

*Figure 3.4 M/120 System Physical Memory Map*

| Address Range | Assigned to |
|---|---|
| 0x1b00 0002 | AT DAck Enable Register |
| 0x180f 0006<br>0x180f 0002 | 7990 Lance Controller (Ethernet) |
| 0x180e 000a<br>0x180e 0002 | 9516 DMA Controller (UDC) |
| 0x180d 00f3<br>0x180d 0003 | MB87030CR SCSI Controller |
| 0x180c 00ff<br>0x180c 0003 | 8254 Interval Timers |
| 0x180b 1fff<br>0x180b 0003 | MK48T02 Calendar Clock & 2Kbyte NVRAM |
| 0x180a 003f<br>0x180a 0003 | 2681 DUART1 |
| 0x1809 003f<br>0x1809 0003 | 2681 DUART0 |
| 0x1808 0003 | LED Register |
| 0x1807 0002 | AT Control Register |
| 0x1806 0003 | Timer1 Acknowledge |
| 0x1805 0003 | Timer0 Acknowledge |
| 0x1804 0002 | Fault ID Register |
| 0x1803 0000 | Fault Address Register |
| 0x1802 0002 | Interrupt Mask Register (IMR) |
| 0x1801 0002 | Interrupt Status Register (ISR) |
| 0x1800 0002 | System Configuration Register |

*Figure 3.5 M/120 Local I/O Map*

Each of the devices and registers in the local I/O address space are described in detail later in this chapter.

# Interrupt System

The R2000 CPU supports six level–triggered interrupt inputs (**Intr0*** through **Intr5***). Figure 3.6 illustrates the assignment of these interrupts in the M/120 system.

Each of the R2000 interrupts can be individually enabled/disabled by setting/clearing an appropriate bit in the processor's internal Status Register. *All* interrupt inputs to the processor can also be disabled via a single bit in this register. (Refer to the *MIPS RISC Architecture* book for a complete description of the processor's Status Register.) Additionally, the Level–0 interrupts (Intr0*) from the AT–bus and other I/O system controllers can be individually enabled/disabled via the *Interrupt Mask Register.*

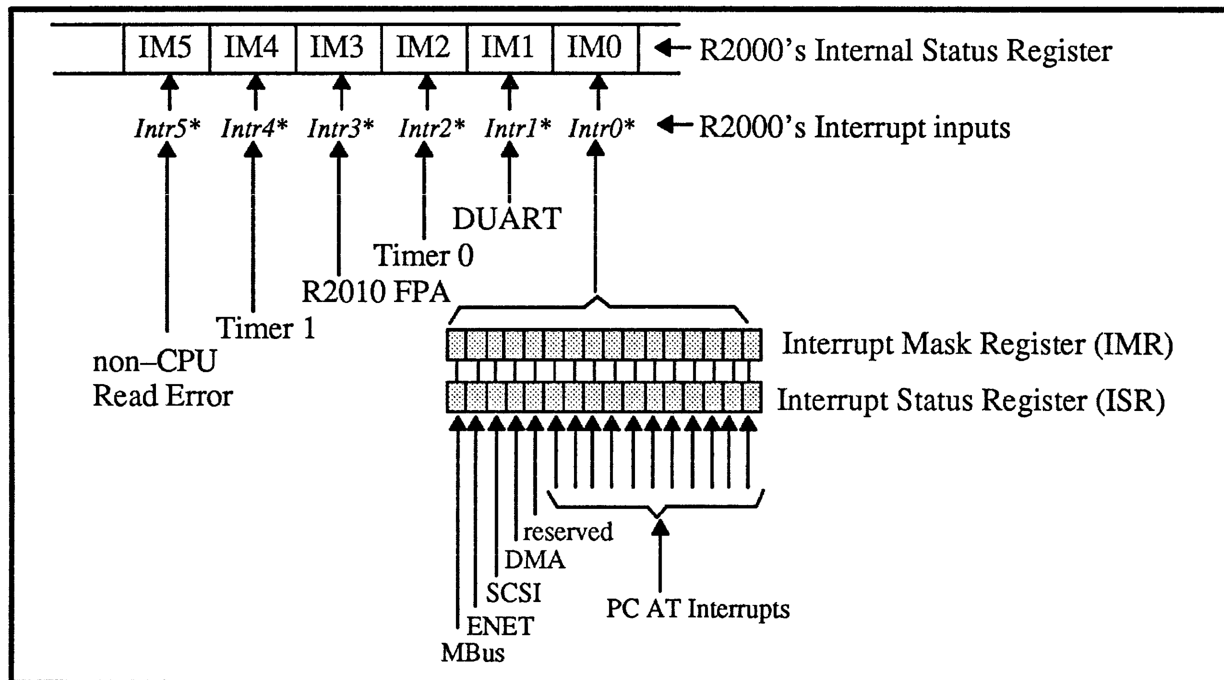



*Figure 3.6 M/120 Interrupt Structure*

Each of the interrupt sources, their initiation, and termination is described later in this chapter when the corresponding device is discussed. The following is a summary of the interrupt sources.

- **Level 0** interrupt (Intr0*) is connected to the 11 PC/AT interrupt sources as well as local interrupts from devices such as the SCSI, DMA or Ethernet controllers. The *Interrupt Mask Register* is used to enable/disable the generation of the Level 0 interrupt from these sources.
- **Level 1** interrupt (Intr1*) is derived from a signal which logically ORs the interrupt output from DUART0 (console and remote serial ports) and DUART1 . Level 1 interrupts are acknowledged (and terminated) by reading or writing the DUARTs' Interrupt Status/Mask Register.

- **Level 2 and Level 4** interrupts (Intr2* and Intr4*) are assigned to the 8254 programmable timers, Timer0 and Timer1 respectively. Two physical addresses in the Local I/O space are designated as timer/counter acknowledge registers. Reading these registers clears the corresponding timer interrupt.
- **Level 3** interrupt (Intr3*) is used by the R2010 FPA coprocessor interface.
- **Level 5** interrupt (Intr5*) is used to signal bus errors (except those occurring during CPU reads) reported by the M/120 main memory bus or the PC/AT bus. Software can read the *Fault ID Register* to determine the cause of the interrupt. This interrupt level is reset when the interrupt handler reads the contents of the *Fault Address Register,* which contains the 32–bit address of the physical location which caused the error. Reading the *Fault Address Register* clears the *Fault ID Register* as well. Refer to the section *Memory Fault Handling* later in this chapter for additional details.

## Interrupt Level–0

There are 16 possible sources that can cause the system to assert the level 0 interrupt input (Intr0*) to the R2000 microprocessor. These interrupt sources are grouped together and routed through a single 16–bit register referred to as the *Interrupt Status Register* (ISR), and are selectively maskable through a separate 16–bit *Interrupt Mask Register* (IMR). Any interrupt reporting to the ISR register may cause a level 0 interrupt assertion if its corresponding "enable" bit at the IMR is enabled. Figure 3.7 shows the bit assignments for the ISR and IMR.

To acknowledge and cause the remission of a level 0 interrupt asserted by a device, the interrupt handler routine must access the interrupting device's control or status registers to ascertain and service the cause of the interrupt, This causes the interrupting device to "de–assert" the interrupt at its input to the ISR within two clock cycles.

### Interrupt Status Register (ISR)

The *Interrupt Status Register* (ISR) located at half–word physical address 0x1801_0002 can be read to determine the source of the Intr0* interrupt level. The presence of a logic "1" value in an ISR bit position indicates that there is an interrupt pending for that source. All bits within the ISR default to undefined state at power–up or manual system reset. However, no interrupt is generated at Intr0* because the *Interrupt Mask Register* (IMR) defaults to all interrupts disabled at reset.

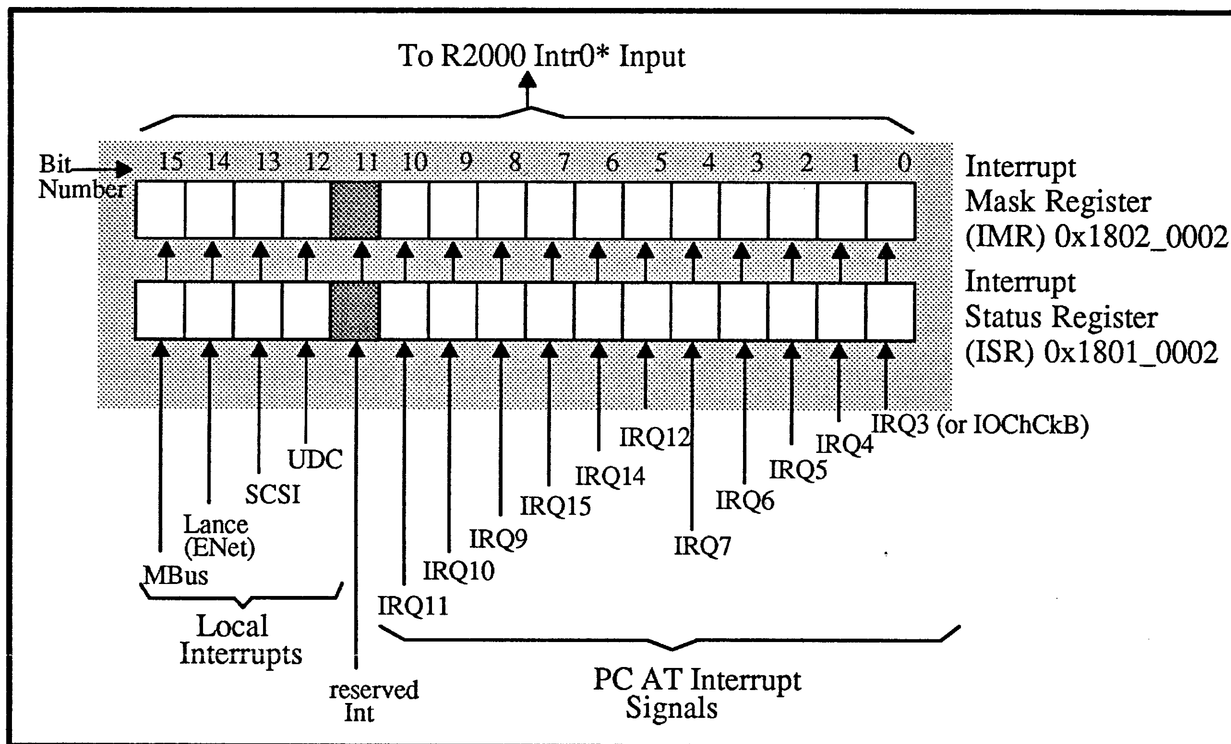



*Figure 3.7. M/120 Interrupt Status and Interrupt Mask Registers*

### Interrupt Mask Register

The 16–bit *Interrupt Mask Register* (IMR) is a read/write addressable register located at 0x1802_0002. All local interrupt sources can be masked by writing the appropriate mask bit(s) to the IMR. Refer to Figure 3.6 for IMR bit assignments. All bit positions within the IMR default to logic "0" (interrupts disabled) at power–up or manual system reset. Writing a logic "1" to an IMR bit enables the associated interrupt.

Bits 10 through 0 of the register are assigned to the AT bus interrupt requests. Note that the hardware imposes no prioritization of the AT bus interrupts.

Software can enable/disable multiple bits of the IMR register simultaneously. If an interrupt enable bit is previously set (enabled) and an interrupt occurs before or at the same time that interrupt enable bit is reset in the IMR, the interrupt will be allowed.

## Memory Fault Handling

The M/120 System provides two registers to facilitate handling of faults that occur during memory transactions. The R2000 processor's level 5 interrupt (Intr5*) is asserted as a result of a Bus Error or Time Out. An interrupt handler can determine the cause of the fault by examining the contents of the *Fault ID Register* (FID) and can determine the source of the fault by reading the *Fault Address Register* (FAR).

## Fault ID Register (FID)

The *Fault ID Register* (FID) is a 16–bit register that helps system software recover from a memory fault by logging the exact nature or cause of the fault. (The device address responsible for the fault is captured in the *Fault Address Register*). This information is preserved in the FID register until the FAR is read. Typically, the interrupt exception handler would read the contents of the FID register to determine the cause of the interrupt, and then read the contents of the FAR to ascertain the address where the fault occurred.

Figure 3.8 illustrates the bit assignments within the FID register and the paragraphs that follow describe the function of each bit.

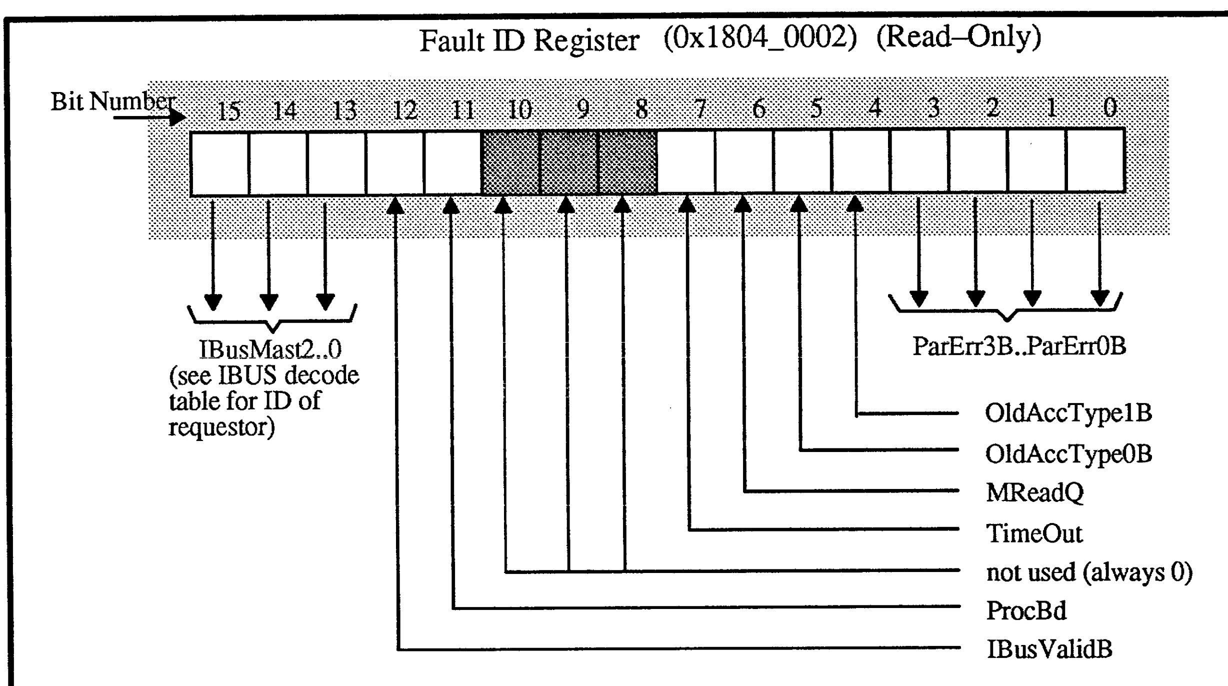



*Figure 3.8. Fault ID Register*

**ParErr3B..ParErr0B** (bit3..bit0): These bits indicate which of the four bytes comprising a 32–bit word contain a parity error: if the parity error is in the low–order byte (bits 0–7) then bit3 would be set to "0", and so on. This field is undefined and should be ignored if the *TimeOut* bit is set.

**OldAccType1B OldAccType0B** (bit5..bit4): This field is the ones–complement of the AccType field which existed on the M–Bus at the time of the fault. The AccType field normally indicates a "data–type" or size of transfer (byte, half–word, tri–byte, word) as shown in the following table.

| OldAccTypeB 1 0 | Data Size |
|---|---|
| 1 1 | Byte (8 bits) |
| 1 0 | Half–word (16 bits) |
| 0 1 | Three bytes (24 bits) |
| 0 0 | Word (32 bits) |

**MReadQ** (bit6): This bit indicates whether the fault occurred during a read operation or write operation: the bit is set to "1" if a read operation was in process and set to "0" if a write operation was being performed. A read indication could mean, for example, the R2412 CPU reading from the I–Bus or the Am9516 UDC reading from M–Bus memory.

**TimeOut** (bit7): This bit indicates that the source of the fault monopolized the M–Bus for an excessive amount of time (approximately16 microseconds) and was disengaged from the M–Bus in order to allow a refresh cycle to execute. This could happen if, for example, an I–Bus device were to read from a non–existent memory location or if the processor were to attempt a write to any PROM.

**Reserved** (bit10..bit8): These bits are not used and will always return "0" when read.

**ProcBd** (bit11): This bit indicates that the processor board (either the CPU or the write buffer) was responsible for the fault. This bit, in conjunction with MReadQ, indicates whether the fault was due to the write–buffer address (MReadQ = "0") or the CPU read (MReadQ = "1").

**IBusMast2..IBusMast0** (bit13..bit12): These three bits encode one of eight possible I–Bus masters as follows:

| *FaultID Reg Bits* 15 M2 | 14 M1 | 13 M0 | *Unit Requesting IBus Access* |
|---|---|---|---|
| 0 | 0 | 0 | PC AT Level 4 |
| 0 | 0 | 1 | PC AT Level 3 |
| 0 | 1 | 0 | PC AT Level 2 |
| 0 | 1 | 1 | PC AT Level 1 |
| 1 | 0 | 0 | 9516 DMA for Chaining |
| 1 | 0 | 1 | 9516 DMA for PC AT |
| 1 | 1 | 0 | 9515 DMA for SCSI |
| 1 | 1 | 1 | Lance |

**IBusValidB** (bit15): This bit, when set to "0" indicates that the encoded field *IBusMast(2:0)* is valid. If the *IBusValidB* bit is set to "1" the data in the *IBusMast(2:0)* field should be ignored.

## Fault Address Register (FAR)

The *Fault Address Register* (FAR) is a 32–bit, read–only *word–addressable* register at memory mapped I/O address 0x1803_0000. The FAR is always synchronously latching the physical addresses used by the system's memory controller logic in anticipation of a memory fault (parity error or bus timeout).

When the system's memory controller logic detects a memory fault, it asserts the level–5 interrupt (Intr5*) and disables the FAR from latching additional memory addresses. The captured fault address is held until software reads the *Interrupt Status Register* (ISR). Reading the ISR causes the Intr5* signal to be de–asserted and also allows the FAR to resume latching physical addresses.

# System Configuration Register

The 16–bit *System Configuration Register* located at address 0x1800_0002 provides information about the configuration of various system elements and lets software control the operation of some system devices and activities. The low–order eight bits of this register are read–only and should not be written into. Figure 3.9 shows how the bits in the register are interpreted. The paragraphs that follow describe each of the bits in detail.

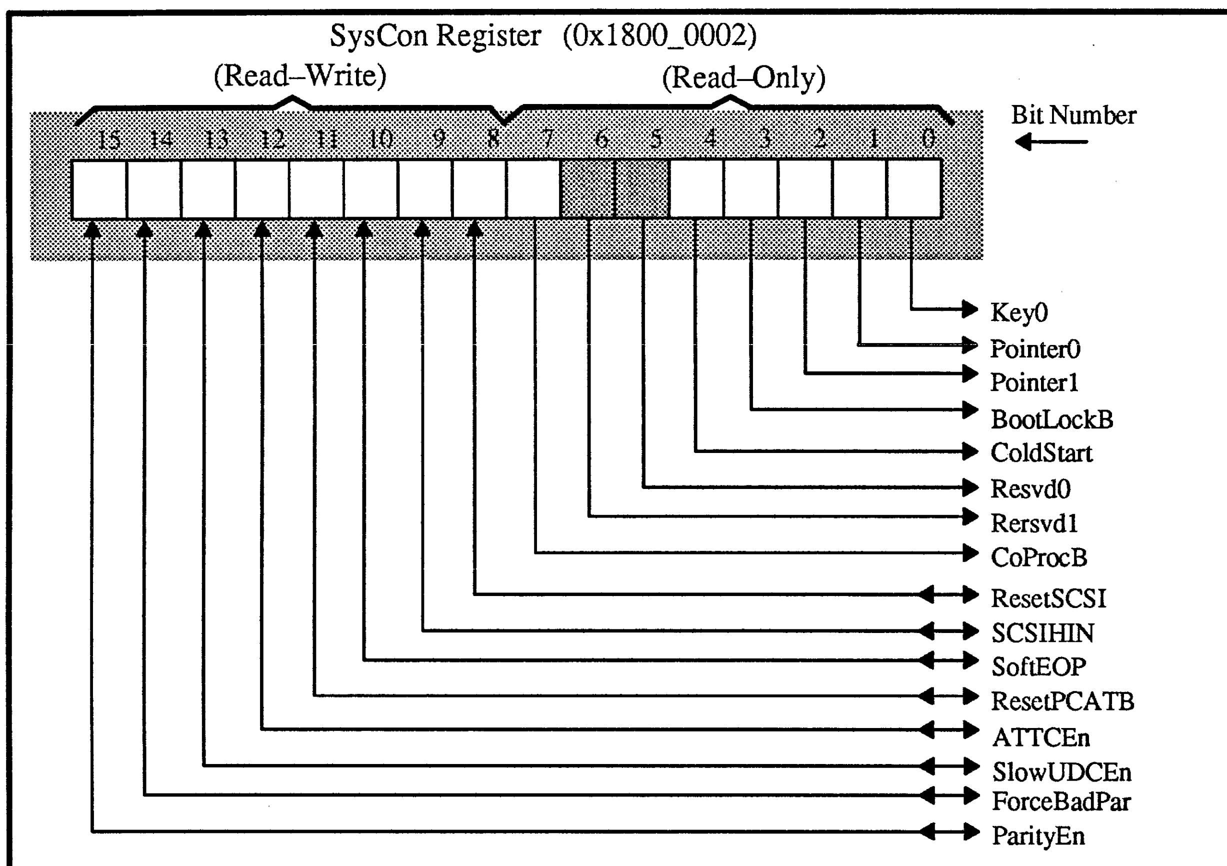



*Figure 3.9 M/120 System Configuration Register Bit Assignments*

**Key0** (Bit 0): Indicates the type of CPU board currently in use on the R2400. In current versions of the system, this bit is set to “1”.

**Pointer[1:0]** (Bits 2:1): These bits are intended for diagnostic purposes and indicate the current position of the SCSI byte transfer counter in the system’s SCSI/DMA logic. Diagnostic software can determine a residual byte–count which exists between the SCSI Protocol Controller and the main memory as shown in the following table.

| Pointer 1 0 | Residual byte count |
|---|---|
| 0 0 | none |
| 0 1 | 3 bytes remaining |
| 1 0 | 2 bytes remaining |
| 1 1 | 1 byte remaining |

**BootLockB** (Bit 3): This bit is set to "0" if the front–panel keyswitch is in the *Lock* position and indicates to system software that the operating system should not be booted after reset. This is a security feature based on the position of the front–panel keyswitch. This bit is set to "1" if the keyswitch is in the *Unlock* position and tells system software that it should allow a manual or auto–boot sequence to continue from power–up all the way to multi–user UMIPS.

**ColdStart** (Bit 4): This bit is set to "1" if a "power–up" system reset has just been executed and it indicates to system software that main memory initialization is required. This bit is set to "0", if the reset was executed via the front–panel keyswitch and indicates to system software that the contents of main memory may contain relevant information which the operating systems software may wish to examine.

**Rsvd1,Rsvd0** (Bita 6:5): These two bits are reserved.

**CoProcB** (Bit 7): When set to "0", indicates the presence of the R2010 FPA coprocessor.

**ResetSCSI** (Bit 8): Set by software to initialize the SCSI/UDC DMA state machine. Software must first set this bit to "1" and then to set the bit back to "0" to initiate and then terminate the reset operation.

**SCSIHIN** (Bit 9): Set by software to indicate the direction of a SCSI DMA data transfer to the SCSI/UDC DMA state machine and the SCSI Protocol Controller. When set to "1" it indicates that data is to be transferred from a SCSI device (tape or disk) into main memory, when set to "0" the direction of data transfer is from main memory out to a SCSI device.

**SoftEOP** (Bit 10): Set by software to send an EOP (end–of–process) signal to the 9516 UDC. Refer to the *Direct Memory Access (DMA)* section later in this chapter for details.

**ResetPCATB** (Bit 11): Generates the signals to reset the AT bus. Software must first set this bit to "0" and then set the bit back to "1" to assert and then release the bus reset signal.

**ATTCEn** (Bit 12): Enables the transfer complete (TC) signal required by some devices to terminate a DMA cycle on the AT bus. Refer to the *Direct Memory Access (DMA)* section later in this chapter for details.

**SlowUDCEn** (Bit 13): Used by software to signal when "slow–readable" UDC registers (Am9516 DMA device) will be accessed by the CPU (programmed I/O mode). Refer to the *Direct Memory Access (DMA)* section later in this chapter for details.

**ForceBadPar** (Bit 14): Used by software to cause "bad–parity" to be forced into the main memory parity checker logic. This bit is primarily intended for diagnostic software use.

**ParityEn** (Bit 15): Used by software to enable or disable the occurrence of a level–5 interrupt resulting from a parity error. Software can set the bit to a logic "1" value to enable a parity error to cause a level–5 interrupt. This bit is primarily intended for diagnostic software use.

# Direct Memory Access (DMA)

The M/120 uses an Am9516 Universal DMA Controller (UDC) to support direct memory access operations in the system. One channel (*CH#2*) of the UDC is dedicated to supporting transactions to and from the AT bus and the other channel (*CH#1*) is dedicated to supporting the SCSI interface. Figure 3.10 illustrates the role of the controller in the system and shows the registers associated with the UDC interface. The paragraphs that follow briefly describe how the UDC is utilized in the M/120 system. Refer to the *Am9516 Universal DMA Controller Technical Manual* for a complete description of the device and its capabilities.

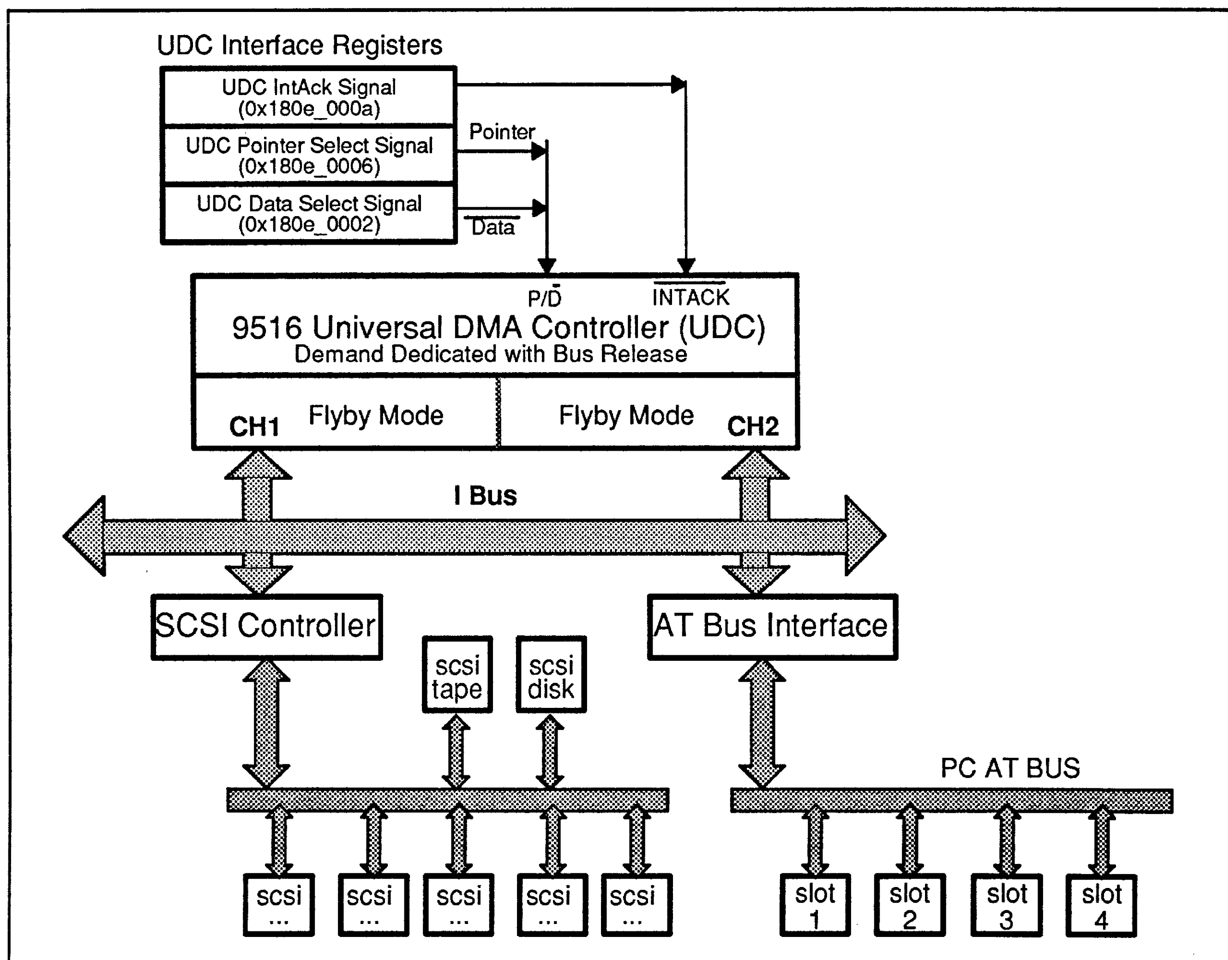



*Figure 3.10 Direct Memory Access (DMA)*

## DMA Controller Operating Modes

Although the 9516 supports several different transfer types and transaction types, the M/120 hardware and RISC/os software place constraints on the controller modes that can be used in the system. The general constraints are:

- **Transfer Type**: The system requires that the controller operate in the *Demand Dedicated with Bus Release* mode. The *CPU Interleave* mode is specifically not supported by the current version of software since this mode of operation can interfere with SCSI operation.
- **Transaction Type**: The system requires that both channels operate only in the *Flyby* mode. The SCSI interface requires that channel 1 always operate in this mode. Some AT bus devices can (or sometime, require) that DMA operations be performed in the *Flowthru* mode, but the current version of RISC/os does not support the *Flowthru* mode for channel 2 (the AT bus channel).
- **Operand Size**: The 9516 and system hardware support byte/half–word packing/unpacking through a byte/half–word funneling register which transfers data between half–word wide main memory and byte–sized peripherals.

## DMA Controller Interface Registers

Three registers located in the M/120's local I/O address space provide the software interface to the DMA controller. The address and function of these registers is as follows:

| I/O Address | Read (R) Write (W) | Register Name/Function |
|---|---|---|
| 0x180e_0006 | R/W | *Pointer Address.* Accessing this address with a read or write operation causes the UDC's P/D (Pointer/Data) input signal to be asserted high. A write to this location causes the write data to be loaded into the Pointer Register where it is used to "point to" (address) one of the UDC's internal registers. The register that is pointed to can then be accessed by directing subsequent read/write operations to the *Data address* (described below). |
| 0x180e_0002 | R/W | *Data Address.* Read and write operations to this address access the UDC's internal register specified by the current contents of the Pointer Register. |
| 0x180e_000a | R/W | *Interrupt Acknowledge Address.* Writing to this address pulses the UDC's INTACK input signal. This informs the controller that its request for an interrupt has been granted. |

*Figure 3.11 9516 DMA Controller Register Summary*

## DMA Software Control

Three bits in the *System Configuration Register* are associated with the interface to the 9516 DMA controller:

- **SoftEOP** (Bit 10): Software can set this bit to assert the EOP (end–of–process) signal to the 9516 UDC. Asserting the EOP signal is one method of terminating a DMA transfer. In the M/120, this termination method is intended primarily for diagnostic purposes since it aborts any DMA operation in progress.
- **ATTCEn** (Bit 12): Setting this bit enables sending of the DMA transfer complete (TC) signal required by some devices to terminate a DMA cycle on the AT bus. Refer to the discussion in Appendix A for details on the use of this bit.
- **SlowUDCEn** (Bit 13): Used by software to signal when "slow–readable" UDC registers must be accessed by the CPU. Setting this bit causes the DMA interface logic to insert the required number of wait–states in a UDC read–access. Refer to the manufacturer's data sheet for a discussion of "slow–readable" UDC registers.

There are also four bits in the *AT Bus Control Register* that affect the operation of channel 2 (CH#2) in the DMA controller. These bits are listed below and described in more detail in the *AT Bus Interface* section later in this chapter.

- **FlowToMbus** (Bit15) This bit specifies the direction of the FlowThru mode of operation for channel 2 of the DMA controller.
- **FlowThruMode** (Bit14) This bit specifies whether channel 2 of the DMA controller is operating in the *FlowThru* mode or the *Flyby* mode.

**NOTE**

Flowthru cycles to the AT bus require that the 9516 be put in "CPU Interleave" mode. This mode is not compatible with the M/120 SCSI state machine, which requires two "atomic" bus cycles. THerefore, 9516 Ch2 AT Flowthru cannot be done concurrently with 9516 Ch1 servicing SCSI. Therefore, this bit must be set to "0" to specify the Flyby mode when channel 2 of the DMA controller is being used for the AT bus.

- **DAckEnB** (Bit13) This bit specifies whether channel 2 of the DMA controller will generate the DAck signal during *Flyby* operations. Setting the bit to "0" enables the DAck signal.
- **ATReqEn** (Bit12) This bit is enables/disables requests for service (DReq) to the DMA controller from devices on the AT bus. Setting the bit to "1" enables the DReq signals.

# LED Register

The M/120 system Motherboard provides eight green light–emitting–diodes (LEDs) that software can turn on and off as visual indicators of the board's operation. (For example, the diagnostic program described in **Appendix B, Power On Diagnostics** uses these LEDs.) The LEDs are turned on and off by writing to the LED Register located at address 0x1808_0003. The bit assignments for this register are shown in Figure 3.12.

At "power–up" or manual system reset the entire contents of the LED register default to a logic "0" state (all LEDs ON). Bits 0 through 7 represent the code for LEDs. This 8–bit register is "byte–addressable" only and is read/write addressable. Writing a "1" to a bit position turns an LED off and writing a "0" turns the corresponding LED on.

LED Register (0x1808 0003)

Bit Number → 7 6 5 4 3 2 1 0

MSB LSB

0 = turn LED ON
1 = turn LED OFF

*Figure 3.12. LED Register Bit Assignments*

# THE ID PROM

A 32–byte PROM is provided in the M/120 system and contains identifying information about the system that can be read by software. The ID PROM is positioned at address 0x1e00_0000 and its contents are as follows:

| Address | Contents |
|---|---|
| 0x1e00_0000 | *Board Type (4)* |
| 0x1e00_0007 | *Revision level (0x10 = M/120–5, 0x20 = M/120–3)* |
| 0x1e00_000b | *Five–digit serial number* |
| 0x1e00_000f | |
| 0x1e00_0013 | |
| 0x1e00_0017 | |
| 0x1e00_001b | |

The remaining bytes of the ID PROM are currently unused and will return all "1's" (ff) when read. The last byte of the PROM contains a checksum.

# I/O Subsystems

The M/120 I/O subsystem includes the following:

- Real–Time Clock & Interval Timers
- Battery Back–Up Calendar Clock
- Serial Ports
- SCSI Interface
  ± SCSI Rigid Disk Drive
  ± SCSI Tape Drive
- Ethernet Interface

Each of these I/O elements is described in the pages that follow. The AT bus is considered an extension of the I/O subsystem and is treated separately after this section describing the I/O subsystem.

## Counter/Timer

The M/120 provides an Intel 8254 (or equivalent) programmable counter/timer that includes 3 separate counters. The outputs of counters 0 and 1 (OUT0 and OUT1) from the chip are connected to two of the interrupt inputs of the R2000 processor (Intr2* and Intr4*, respectively). Counter 2 is driven by a 3.6864 MHz clock and it in turn drives the other two counters. The connections for the counter/timer are illustrated in Figure 3.13.

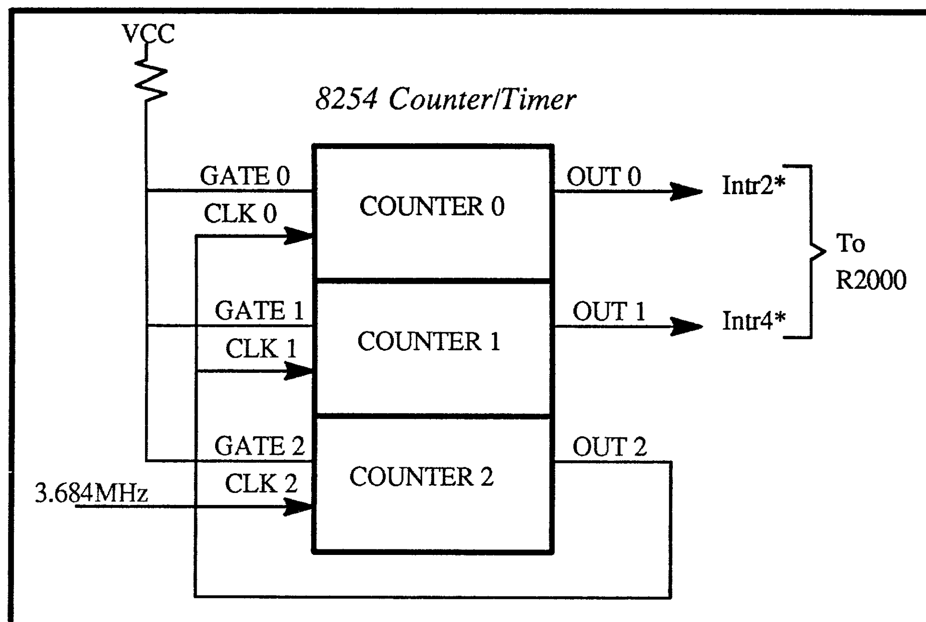



*Figure 3.13. Counter/Timer Connections*

This configuration lets counters 0, 1, and 2 be programmed so that counters 0 and 1 can generate interrupts at intervals ranging from 1 millisecond to 100 milliseconds. The operating sys-

tem uses the counter 0 output (Intr2*) as a scheduling clock and the counter 1 output (Intr4*) as a profiling clock.

Figure 3.13 also shows a GATE signal input to each counter that is connected to a logic "1". This connection places each counter in a continuous countdown mode: after an initial count is loaded into a counter, that count is decremented on each positive–to–negative clock transition.

### Counter/Timer Interrupt Acknowledge Registers

Since the counter outputs consist of pulses, the interrupt signals derived from these outputs are latched on the M/120 to ensure that the processor receives the interrupt. In order to clear the timer interrupts, the interrupt service routine must read the contents of the appropriate interrupt acknowledge register located in the Local I/O address space as shown in Figure 3.13.

| Address | Register Name |
|---|---|
| 0x1805_0003 | Timer 0 Interrupt Acknowledge |
| 0x1806_0003 | Timer 1 Interrupt Acknowledge |

*Figure 3.14 Counter/Timer Interrupt Acknowledge Register Addresses*

Note that both of these are 8–bit read–only registers. The read operation returns no meaningful data and writing to these registers has no effect.

Because of the way in which the counters are interconnected, there is no way to individually disable the counters or prevent them from generating their OUT signals (and related interrupt signals). Therefore, if you want to disable the interrupts produced by counters 0 and 1, you must disable them via the R2000 Processor's Status Register.

### Counter/Timer Register Summary

The program accessible registers of the counter/timer device are 8–bits wide. (However, you can load a 16–bit value into a counter by writing two successive bytes to a counter.) Figure 3.15 lists the I/O addresses used by the processor to communicate with the counter/timer registers. Refer to the 8254 data sheets for details on the operation and use of these registers.

| Address | Register Name |
|---|---|
| 0x180c_0003 | Counter 0 initial count register |
| 0x180c_0007 | Counter 1 initial count register |
| 0x180c_000b | Counter 2 initial count register |
| 0x180c_000f | Counter/Timer control word register |

*Figure 3.15 Counter/Timer Register Addresses*

Although the 8254 has six different operation modes, the M/120 uses only Mode 2 — the Rate Generator mode. In this mode the counters function like a divide–by–N counter. When an initial value is loaded into a counter, the counter immediately begins decrementing the count. When the count reaches 1, the counter's OUT signal goes low for one CLK pulse and then returns high. Referring back to Figure 3.13, you can see that when OUT2 (from counter 2) makes a transition, it clocks both counters 0 and 1, thus decrementing the count held in these two counters. When OUT0 makes a transition, it sends the Intr2* signal to the R2000, and when OUT1 makes a transition it sends the Intr4* signal to the R2000. When any counter reaches a count of zero, it is automatically reloaded with the initial count and the same sequence is repeated indefinitely.

## Real–Time Clock & NVRAM

The Motherboard provides a battery backed–up real time calendar/clock function for the CPU. A Mostek Mk48T02 Timekeeper RAM (or equivalent) device containing 2048 bytes of non–volatile static RAM (NVRAM) is used. The system uses the NVRAM to store items such as network address, baud rate of the console ports, bootfile, console location, and "time–of–day" valid bit. This information is used by both the boot PROMS and the RISC/os operating system as configuration information required for booting.

### Real–Time Clock Register Summary

Figure 3.16 summarizes the registers and general purpose RAM bytes provided by the Real–Time Clock. All registers and RAM locations are 8–bit bytes and are located in the Local I/O address space.

| Address | Usage |
|---|---|
| 0x180b_0003 | RAM location 0 |
| 0x180b_0007 | RAM location 1 |
| ... | ... |
| ... | ... |
| ... | ... |
| 0x180b_1fdf | RAM location 2040 |
| 0x180b_1fe3 | Control Register |
| 0x180b_1fe7 | Seconds |
| 0x180b_1feb | Minutes |
| 0x180b_1fef | Hour |
| 0x180b_1ff3 | Day of week |
| 0x180b_1ff7 | Day of month |
| 0x180b_1ffb | Month |
| 0x180b_1fff | Year |

*Figure 3.16. Real–Time Clock Register and RAM Addresses*

For a complete description of this chip, refer to the MK48T02 data sheet from *Mostek Corporation.*

## Serial Ports

The M/120 provides four RS–232C interfaces that are supported by two 2–channel DUART (Dual Universal Asynchronous Receiver/Transmitter) devices (SCN2681 or equivalent). One channel of each DUART (Channel B) can be programmed to provide full modem control and may be used as a download facility. The other channel (Channel A) is suitable for a console.

The DUARTs can be programmed to interrupt the processor when they receive a character from an attached terminal. The interrupt outputs (INTRN) from both of the DUARTs are tied to the Intr1* pin of the R2000 processor. The devices can be programmed to interrupt the CPU when they receive a character from a terminal.

### DUART Programmable Registers

DUART operations are controlled by programming the devices' internal registers. The 16 internal registers of each DUART are located in the I/O address space and are addressed as shown in Figure 3.17. The function of each of these registers is described briefly in the paragraphs that follow. (For a complete description of the DUART, refer to the SCN2681 Series DUART data sheet from *Signetics Microprocessor Division.*)

All of the DUART registers are eight bits wide and are located at odd addresses.

| Address | | Mnemonic | Description (Read/Write) |
|---|---|---|---|
| DUART 0 | DUART 1 | | |
| 0x1809_0003 | 0x180a_0003 | MR1A/MR2A | Channel A Mode Registers |
| 0x1809_0007 | 0x180a_0007 | SRA/CSRA | Channel A Status/Clock Registers |
| 0x1809_000b | 0x180a_000b | —/CRA | Channel A Command Register |
| 0x1809_000f | 0x180a_000f | RHRA/THRA | Channel A Receive/Transmit Holding Register |
| 0x1809_0013 | 0x180a_0013 | IPCR/ACR | Input Port Change/Auxiliary Control Registers |
| 0x1809_0017 | 0x180a_0017 | ISR/IMR | Interrupt Status/Interrupt Mask Registers |
| 0x1809_001b | 0x180a_001b | CTU/CTUR | Counter/Timer Upper Register |
| 0x1809_001f | 0x180a_001f | CTL/CTLR | Counter/Timer Lower Register |
| 0x1809_0023 | 0x180a_0023 | MR1B/MR2B | Channel B Mode Registers |
| 0x1809_0027 | 0x180a_0027 | SRB/CSRB | Channel B Status/Clock Select Registers |
| 0x1809_002b | 0x180a_002b | —/CRB | Channel B Command Register |
| 0x1809_002f | 0x180a_002f | RHRB/THRB | Channel B Receiver/Transmit Holding Register |
| 0x1809_0033 | 0x180a_0033 | —/— | Reserved |
| 0x1809_0037 | 0x180a_0037 | INPT/OPCR | Input Port/Output Port Control Register |
| 0x1809_003b | 0x180a_003b | START CNTR | Start Counter/Set Output Port Bits |
| 0x1809_003f | 0x180a_003f | STOP CNTR | Stop Counter/Reset Output Port Bits |

*Figure 3.17. DUART Register Summary*

## Serial I/O Connectors

Separate 25–pin connectors (SIO0, SIO1, SIO2, and SIO3 ) are provided on back of the M/120 for the RS–232C connections as shown in Figure 3.18.

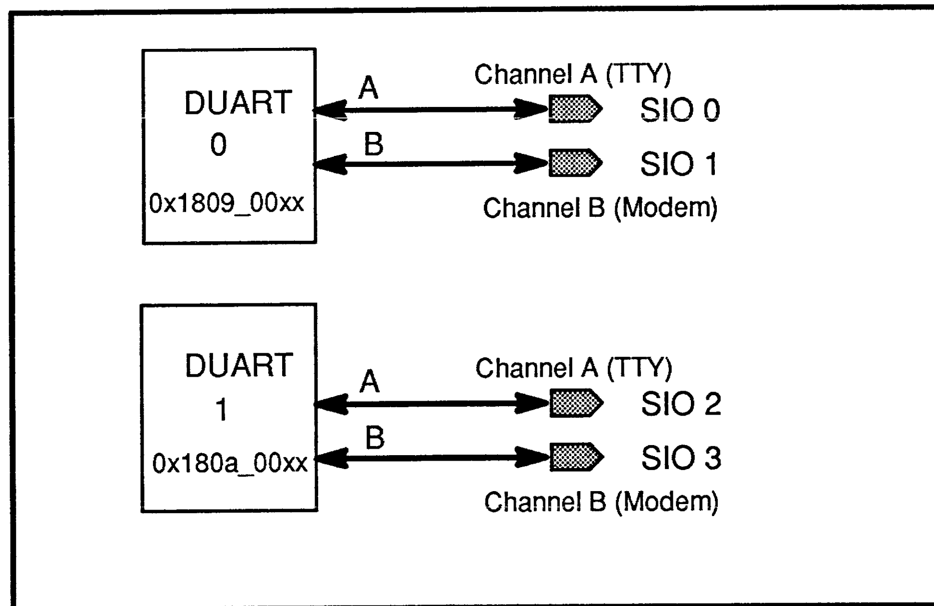



*Figure 3.18 DUART Connections*

Serial channel A of the DUARTs has only the receive and transmit data lines connected, but channel B of each DUART has complete modem control. The following table shows the functions available on channel B via the DUART's Input Port/Output Port Control register.

| Pin | Function | I/O | Function Description |
|---|---|---|---|
| IP2 | CTSB | Input | Clear to send |
| OP0 | DTRB | Output | Data Terminal Ready |
| OP1 | RTSB | Output | Request to Send |

## SCSI Interface

The Motherboard supports a synchronous Small Computer Systems Interface (SCSI) as defined by ANSI X3.131–1986. M/120 supports one internally mounted 5–1/4 inch winchester disk drive with embedded SCSI. A cartridge tape drive (also with embedded SCSI) is also supported. The disk and/or tape may both support a synchronous or asynchronous SCSI interface or, alternatively, a "mixed" sync/async SCSI interface. In addition, the M/120 SCSI interface is brought out to an external connector to facilitate the integration of up to five additional SCSI based peripherals external to the enclosure.

The SCSI interface is implemented using the Fujitsu MB87030CR–8 SCSI Protocol Controller (SPC) or equivalent, which supports both synchronous and asynchronous SCSI peripherals. The SPC device resides on the M/120 I–Bus and is memory mapped I/O addressable over the IDATA<07:00>data path. In this M/120 application the MB87030 supports the SCSI bus in an *Initiator Only* role.

The SPC provides the following features:

- Full SCSI control (both synchronous and asynchronous)
- Serves as INITIATOR on SCSI
- Synchronous mode transfer with programmable offset (up to 8 bytes)
- Synchronous mode transfer programmable at four rates.
- Maximum data transfer (synchronous) at 4 Mbytes/second.
- Eight byte FIFO data buffering
- 24–bit Transfer Byte counter

Refer to the Fujitsu MB87030 SCSI Protocol Controller Users Manual for additional details on the SPC.

### SCSI Controller Registers

Figure 3.19 illustrates the MB87030 SPC address map and register assignments.

| I/O Address | Read / Write | Register Name |
|---|---|---|
| 0x180d_0003 | Read/Write | Bus Device ID |
| 0x180d_0007 | Read/Write | SPC Control |
| 0x180d_000b | Read/Write | Command |
| 0x180d_000f | Read/Write | Transfer Mode |
| 0x180d_0013 | Read/Write | Interrupt (read) / Reset (write) |
| 0x180d_0017 | Read/Write | Phase sense (read) / Diagnostic (write) |
| 0x180d_001b | Read only | SPC Status |
| 0x180d_001f | Read only | SPC Error Status |
| 0x180d_0023 | Read/Write | Phase Control |
| 0x180d_0027 | Read/Write | Modified Byte Counter |
| 0x180d_002b | Read/Write | Data Register |
| 0x180d_002f | Read/Write | Temporary Register |
| 0x180d_0033 | Read/Write | Transfer Counter High |
| 0x180d_0037 | Read/Write | Transfer Counter Middle |
| 0x180d_003b | Read/Write | Transfer Counter Low |
| 0x180d_003f | Read/Write | External Buffer |

*Figure 3.19. SCSI Protocol Controller Register Summary*

### SCSI Operation Details

All transfers to and from SCSI devices are performed using channel 1 of the 9516 UDC in the flyby mode to move data between main memory and the SCSI SPC. The SPC is the initiator of the transactions, and the data transfer direction is determined by the setting of the SCSIHIN bit at the *System Configuration Register.* A single data request from the SPC starts a state sequencer running. The sequencer automatically continues until a minimum of 4 (store) or 8 (load) bytes are transferred between SCSI and main memory.

Data transfers between the UDC and memory for SCSI are all 32–bit word transfers even though the 9516 is a 16–bit device: half–words are unpacked from memory words for SCSI store transactions (memory → SCSI), and half–words are packed into words for all SCSI load transactions (SCSI → memory). The hardware always forces address bit 1 generated by the UDC to zero for data transfers involving channel 1 (SCSI). SCSI/UDC driver software must ensure that the minimum SCSI data transfer is always a multiple of four bytes.

## Ethernet Interface

The R2400 Motherboard includes the Lance Ethernet controller logic to support the 802.3 Ethernet standard. The Lance Ethernet hardware implementation consists of the Am7990 Local Area Network Controller for Ethernet (or equivalent) which interfaces to the M/120 CPU as memory mapped I/O, and the Am7992A Serial Interface Adapter (SIA). This configuration supports full Ethernet (not *Cheapernet*). The two Lance registers that are directly accessible are listed in Figure 3.20.

| Address | Register Name |
|---|---|
| 0x180f_0002 | Lance Register Address Port |
| 0x180f_0006 | Lance Data Port |

*Figure 3.20 Lance Register Addresses*

# AT Bus Interface

The M/120 system supports expansion through an implementation of IBM's PC/AT bus. This four–slot bus is designed to support virtually any card that works in a PC, PC/AT or equivalent machine. This section provides an overview of the M/120 system's AT bus and the software mechanisms available for controlling the AT bus interface. Appendix A provides a discussion of compatibility considerations to assist in evaluating and integrating AT–type cards into the system.

The description of the bus found in the *IBM PC/RT Technical Reference* has been used as a guide to the bus implementation. The AT bus interface performs all functions necessary to enable the MIPS processor to function as a master or a slave on the AT bus.

## AT Bus Memory Access and Control

The M/120 processor can directly access 16 MBytes of memory and I/O space that are assigned to the AT bus and can perform 8–bit and 16–bit transfers within this space. One DMA channel is dedicated to the AT bus. This channel can be programmed to directly control DMA devices on the bus or to perform memory–style operations. AT bus master cards (*Alternate Controllers*) can access other AT bus 16–bit cards and the M/120's main memory. A one MByte section of the M/120's main memory can be mapped into the AT bus address space. The AT bus address at which the mapping occurs is set with jumpers on the motherboard. (For a description of the motherboard jumpers used to specify the AT bus address for mapping, refer to **Appendix A, AT Bus Compatibility Considerations**.) The address in M/120 main memory which is mapped is set via the AT Control Register described later in this chapter. The base of the 1MB mapped section can be set at any one MByte boundary in AT bus and M/120 space.

## AT Bus Memory Mapping

The 16 MByte AT bus address space is mapped into the M/120's address space eight times as shown in Figure 3.21. The multiple copies of the AT space allow the system's CPU or DMA controller to completely select the type of access. The eight–fold duplication reflects all combinations of three access characteristics: CPU timing versus DMA timing; memory access versus I/O access; and byte swapping versus no byte swapping.

Typical AT implementations allow bus access by either the CPU or by the DMA controller, with slightly different timing for each master. The M/120 allows emulation of the two timing patterns through the choice of address.

The AT bus also allows CPU access to both AT bus memory and AT bus I/O spaces. Again, the choice of M/120 address is used to distinguish the two spaces. The byte swapping capabil-

ity is described in the section that follows. Figure 3.21 defines the eight different access types that can be performed and the address space that corresponds to each access type.

| Main Memory Address | Access Characteristics | | |
|---|---|---|---|
| | CPU or DMA cycle | Mem or I/O cycle | Swap/No Swap |
| 0x1000_0000– 0x10ff_ffff | CPU | Mem | Swap |
| 0x1100_0000– 0x11ff_ffff | CPU | Mem | No Swap |
| 0x1200_0000– 0x12ff_ffff | CPU | I/O | Swap |
| 0x1300_0000– 0x13ff_ffff | CPU | I/O | No Swap |
| 0x1400_0000– 0x14ff_ffff | DMA | Mem | Swap |
| 0x1500_0000– 0x15ff_ffff | DMA | Mem | No Swap |
| 0x1600_0000– 0x16ff_ffff | DMA | I/O | Swap |
| 0x1700_0000– 0x17ff_ffff | DMA | I/O | No Swap |

*Figure 3.21. PC/AT Bus Address Mapping Characteristics*

## AT Bus Byte Swapping

The AT Bus space can be accessed via byte and half–word reads and writes, but not with tri–byte or full word reads and writes. Additionally, if writes are attempted to adjacent half–words, the system's write buffers may merge these writes to an illegal tri–byte or full word transaction. Therefore, the programmer must prevent this merging by flusghing the write buffer between writes to adjacent half–word addresses.

Byte swapping on access is available for programmer convenience. The M/120 uses the convention of "big–endian" addresses (described at the beginning of this chapter) while PCs use Intel's "little–endian" convention. In some programs it may be convenient to swap the position of the bytes within a half–word during reads or writes to AT bus addresses. Swapping is performed only on accesses through a swapped copy of the address space. For a half–word (16–bit) transfer in unswapped space the upper and lower bytes on the AT bus are the same as the upper and lower within the half–word seen by the CPU or DMA controller. In swapped space the upper and lower bytes are reversed on both reads and writes. The addresses are defined in Figure 3.21. All 16–bit bus cycles can be run "byte swapped" or "not swapped". Note that although the I–Bus is only an 8/16 bit data path, insofar as DMA channels are concerned, byte/half–word data transfers can take place on any M–Bus byte or half–word lane, respectively (read or write). This is also true of selecting any byte–lane on the PC/AT bus. This is accomplished with the byte/half–word swap transceivers, which reside between the I–Bus and the M–Bus as illustrated in Figure 3.22.

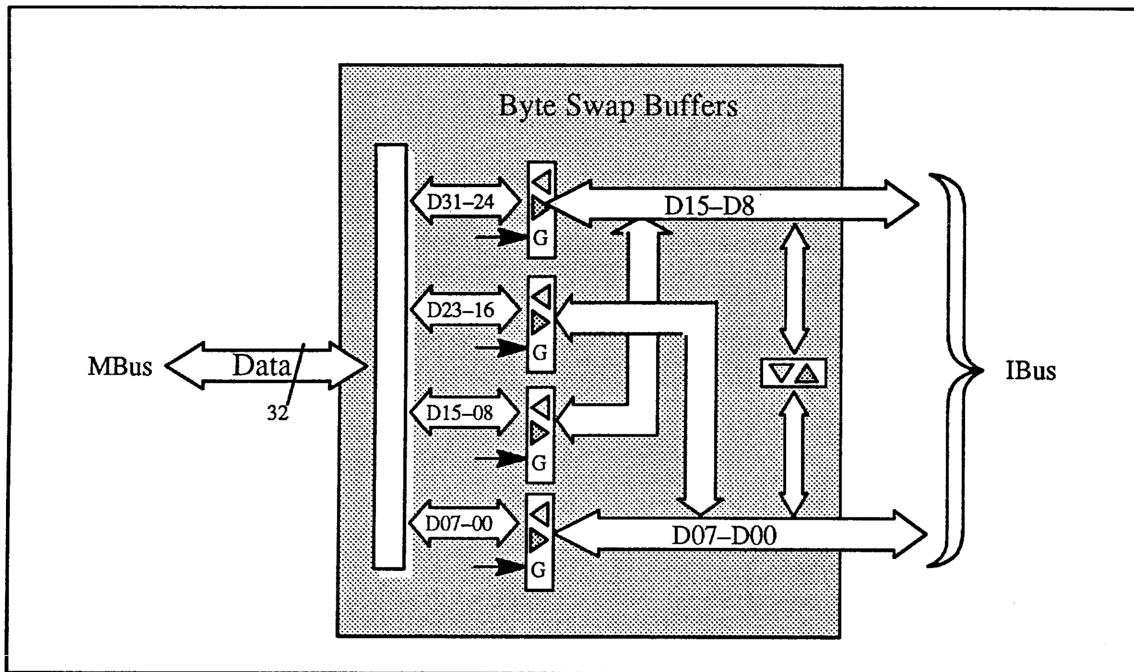



*Figure 3.22. M/120 Byte/half–word Swap Transceiver*

## AT Bus Control Registers

Two registers in the M/120's address space control some operational characteristics of the AT bus. The *AT Control Register* contains read/write control bits to set the mode of operations on the AT bus by the 9516 DMA controller and an enable mask to allow AT bus masters to request the AT and I/O buses. The *AT DAckEn Register* is 16 bits wide and contains mask bits that allow the acknowledge from the 9516 DMA controller to be steered to specific AT cards under program control. Both of these registers are described in detail in the sections that follow.

### AT Control Register

This 16–bit read/write register is located at address 0x1807_0002 and provides bits that control operating modes of the DMA controller, enable/disable bus requests from Alternate Controllers, and specify address mapping for accesses to main memory by alternate controllers. The bit assignments for the register are illustrated in Figure 3.23 and the function of each bit is described in the paragraphs that follow.

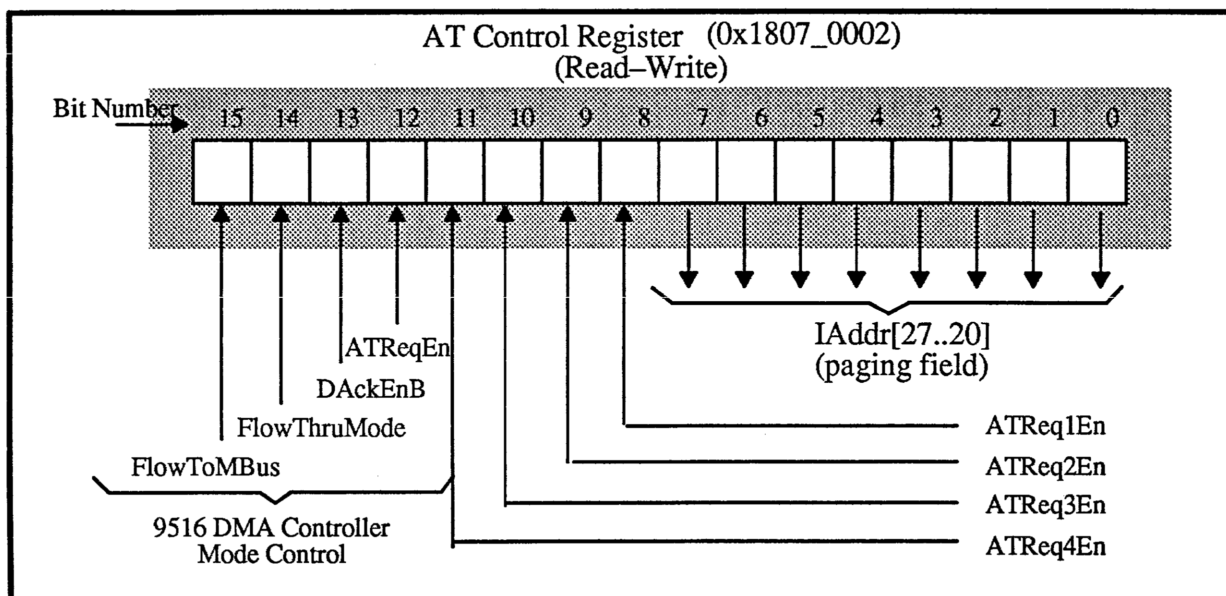



*Figure 3.23. PC/AT Control Register*

**IAddr27..20** (Bit7..Bit0) The system lets *Alternate Controllers* directly access a one MByte address space of the M/120's main memory. The AT Controller provides the low–order address bits (bit19..bit0). The eight bits in this register (IAddr27..20) are used to provide the high–order address bits and thus let software specify the one MByte of main memory that is to be made accessible to the AT bus alternate controller. For a detailed discussion of this address mapping, refer to **Appendix A, AT Bus Compatibility Considerations.**

**ATReq[3..1]En** (Bit11..Bit8) These bits specify whether *Alternate Controllers* located in AT–bus slots 4 through 1 can request access to the system's IBus. For example, when bit 10 is set to 1, an Alternate Controller located in slot 3 can request access to the IBus. (This example assumes that the option jumpers on the motherboard, which let you reassign the requests coming from the AT bus to other slots, have not been changed. Refer to Appendix A for a description of of jumper–selectable options for these signals.) Note that these bits should be set to "0" if no alternate controllers are being used in the system.

**ATReqEn** (Bit12) This bit enables/disables requests for service to the DMA controller from devices on the AT bus. When this bit is set to "1" it specifies that DReqs are enabled to Ch2 of the DMA controller. When the bit is set to "0" it disables requests to the DMA controller.

**DAckEnB** (Bit13) This bit specifies whether CH2 of the DMA controller will generate the DAck signal during *Flyby* operations. Set this bit to "0" if an AT device needs the DMA controller to generate DAckB. Set this bit to "1" to prevent DAckB on the AT bus. Note that this bit globally enables/disables the DAckEnB signals: the DAckEn Register described in the next section lets you individually enable/disable DAckEnB signals.

**FlowThruMode** (Bit14) When this bit is set to "1" it specifies that Ch2 of the DMA controller is operating in the *FlowThru* mode, when the bit is set to "0" it specifies the *Flyby* mode of operation for Ch2 of the DMA controller.

**FlowToMbus** (Bit15) This bit specifies the direction of the FlowThru mode of operation for the DMA controller. When set to "1", the data is being passed from the AT–bus towards main memory and the CPU, when set to "0" the data is flowing from the CPU/main memory to the AT–bus.

## AT DAckEn Register

This 16–bit read/write register is located at address 0x1b00_0002 and provides bits that enable/disable the acknowledge signal coming from the DMA controller to a specific AT bus DAckB signal. The AT bus defines seven separate DAckB signals, DAck0..DAck3 and DAck5..DAck7. Jumpers are provided on the motherboard so that the four bits in the DAckEn register can be used to steer the DMA acknowledge signal onto any of the seven DAck signals. Refer to Appendix A for a description of the jumpers. Figure 3.24 shows the register's bit locations used to control the DAckEn signals.

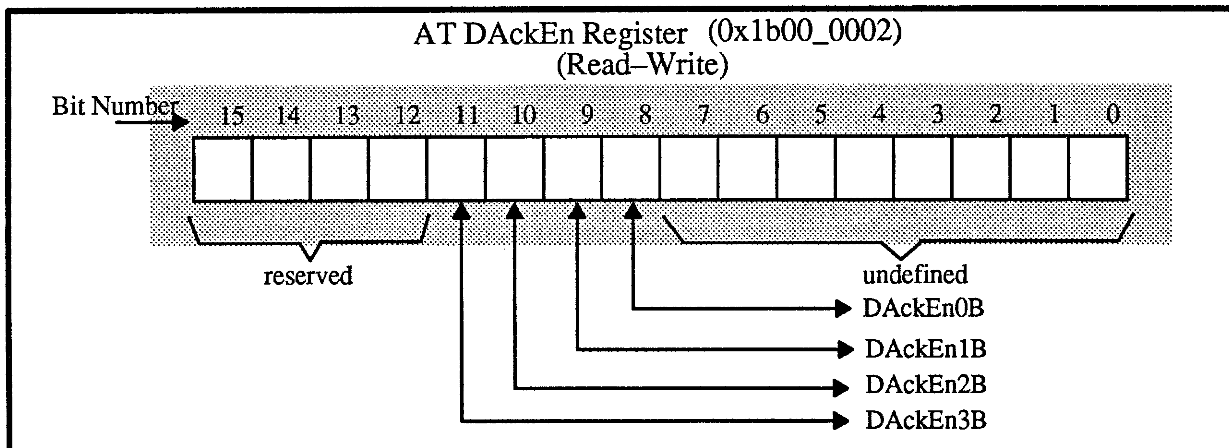



*Figure 3.24. AT DAckEn Register*

For a description of the motherboard jumpers used to steer these bits to the AT bus DAckB signals, refer to **Appendix A, AT Bus Compatibility Considerations.**

# Chapter 4
# Writing Device Drivers

## Introduction

This chapter contains information for programmers writing and installing device drivers under the RISC/os™ (UMIPS) operating system. It does not explain how to write device drivers; rather, it gives specific information on the M/120 that you need to know in order to write them. This chapter describes:

- The file structure of the kernel source subset shipped with each operating system. You must be aware of this structure when creating and modifying the files required by a new device driver.
- The differences between the procedures used on an AT&T operating system and on the operating system supplied by MIPS Computer Systems in reconfiguring the kernel for a new device driver.
- A step–by–step procedure for adding a driver to your operating system.
- M/120 hardware and firmware facilities that influence the logic and implementation of a device driver.
- Procedures to follow in debugging and testing a new device driver.

This chapter assumes you know the following: how the RISC/os (UMIPS) operating system works; advanced C programming; how to write device drivers for System V or Berkeley Software Distribution (BSD) kernel; and how to write the master and system files (*sysgen* and *kernel*) for the driver. For information about how to write these files, see **master(4)** and **system(4)**.

**Appendix E** contains a serial device driver program for the M/120 that you can use as a model when you write a device driver. A version of this program is also located on–line in */usr/src/uts/mips/io/c8.c*.

## File Structure of the Kernel Subset

The operating system is available in both binary code and source code. A subset of the kernel source code, however, accompanies each binary version of the operating system. This section describes the file structure as it applies to the source code delivered with a binary version— not the structure as it applies to the entire operating system.

All pathnames mentioned in this chapter are relative to a shell variable that is called *$ROOT*. This variable usually means '/' on most installed systems. The name *$ROOT* is meant to sig-

nify the root at which the operating system is located. Because the source for the operating system does not have to be installed at '/', *$ROOT* was created to point to the location of the system. This can be useful when more than one person is programming the kernel.

An installed system typically has the reconfiguring kernel subtree installed at */usr/src/uts/mips*. Because this subtree may be installed anywhere, pathnames to the tree are typically expressed as *$ROOT/usr/src/uts/mips*. *$ROOT* in most cases is '/'.

A small set of directories under *$ROOT/usr/src/uts/mips* are required when reconfiguring a kernel to add a device driver. They are *io*, *master.d*, and *bootarea*.

## The io Directory

The *io* directory contains the source to the drivers that may be linked to the kernel. The *io* directory not only contains drivers, but it also contains most of the reconfigurable code in the kernel. Although you may not consider some of the code in this directory 'driver' code, nonetheless the code is a reconfigurable part of the kernel. To add a driver to the system, you place the source in this directory and add the driver name to *Makefile.drv*. The file *Makefile.drv* is a normal makefile that is invoked by the **make** command to build the drivers.

## The master.d Directory

The *master.d* directory contains all of the driver configuration files that are used to reconfigure your system. Every driver has a configuration file, which usually has the same name as the driver, minus any suffix that the driver might have. For example, the driver *dkip.c* has a configuration file called *dkip* in the *master.d* directory.

**Note:** The suffix must not contain a '.' because only the rightmost '.' is noticed. The make files and reconfiguration tools use the suffix to maintain separate configuration files and kernels. For example, you could use *unix.r2300_std* because the suffix is *r2300_std*, but you could not use *unix.r2300.std* because the operating system would recognize *.std* as the suffix.

The configuration files contain information about the driver and structures that may change based on configuration. Refer to **master(4)** for more information about configuration files.

Two special configuration files are also in *master.d* directory, *kernel.suffix* and *sysgen.suffix*. where *suffix* specifies a version of the file. The file *kernel.suffix* is the *master.d* file for the kernel. It contains reconfigurable options for the kernel, the number of buffers in the buffer cache, and the number of procedure (proc) table entries. This information makes it possible, for example, to make a new kernel with a bigger proc table without rebuilding the whole kernel.

The *sysgen.suffix* file is the system configuration file. It lets the reconfiguration tools know what exists in the system, and what drivers to link into the kernel. It also tells the reconfiguration tools which drivers should not be included. Both files are described in more detail later in this chapter.

### The bootarea Directory

The *bootarea* directory is the directory where the reconfiguration Makefile looks to ensure that all of the files in the bootarea are current, before it links them. Once drivers are made, instructions in the makefile link them into the *bootarea* directory.

## AT&T and MIPS Reconfiguration Differences

You add a device driver by reconfiguring the kernel. The way kernels are configured differs between AT&T and MIPS. These differences are described below. For more information about AT&T's process, see the *AT&T 3b2 and 3b5 Driver Design Guide.*

### AT&T's Reconfiguration Process

AT&T's UNIX System V.3 on 3b class computers has an auto configuration boot. This process is divided into three steps. The first step modifies the appropriate files, typically */etc/system* and */etc/master*. These files correspond almost exactly to RISC/os *master.d/master.suffix* and *master.d/sysgen.suffix*. The second step shuts off the system. The third step re–boots what AT&T calls **lboot(1M)** to construct a kernel in memory based on the configuration information in the */etc/system* and the */etc/master* files.

### MIPS' Reconfiguration Process

Unlike AT&T, the RISC/os configuration is not done at boot time. Instead, a version of **lboot(1M)** called **mboot** (not to be confused with AT&T's mboot) performs all **lboot(1M)** functions except the final link. See **lboot(1M)** in the *System Administrator's Reference Manual* for details on the operation of **mboot.**

First the **mboot** program reads the system configuration file *system.suffix* and all of the master files specified by the system configuration files, to generate a file called *master.suffix.c*. This file contains the unresolved externals referenced elsewhere in the kernel. Second, **mboot** generates an *objlist.suffix* file in the *bootarea* directory that contains the complete set of driver objects to link in with *kernel.o*.

The makefile can then produce a complete kernel by compiling *master.suffix.c* into master.*suffix.o* and linking it with *kernel.o* and all of the files in *objlist.suffix*.

# Adding New Drivers

This section explains the five–step procedure for adding drivers to full–source or binary kernels. The steps are:

- Set your environment variable
- Compile your driver
- Create the master file
- Create the configuration file
- Build the kernel

Before adding a new driver, you should be familiar with UNIX System V, Release 3 drivers and their interaction with the kernel.

**Note:** If you purchased the source code and the kernel has not been built, then refer to *MIPS Software Source Release Notes* for information about how to build full–source kernels.

## Set Your Environment Variable

Set the *BUILDTYPE* environment variable to *reconfig*. The following command, assuming that you are in /bin/sh, sets this for you:

```
% BUILDTYPE=reconfig; export BUILDTYPE
```

## Compile Your Driver

Place your driver into the *io* directory, add the name of your driver to *Makefile.drv* so you can compile your object file. Compile your driver by typing make after the prompt as shown:

```
% make
```

This command starts the makefile program that builds your driver and links the object file into the *bootarea* directory.

## Create a Master File

Create a master file for your driver. The name of the master file should be the same as your driver, excluding the suffix. For example, if your driver is named *c8.c,* then name the master *c8*. See **master(4)** for more information on the syntax of *master.d* files.

If your driver requires a major number, you can typically use any number up to the number 255 except the following which are already in use, as indicated below.

| Major Number | Device Driver |
|---|---|
| 0 | Duart |
| 1 | Mem |
| 2 | Gentty |
| 3 | Ram Disk |
| 4 | reserved |
| 5 | Qic tape |
| 7 | Profiler |
| 10 | Streams Clone Device |
| 11 | Ingres |
| 16 | reserved |
| 17–24 | SCSI |
| 32 | Digiboard |
| 48–56 | reserved |
| 64–72 | reserved |

To avoid potential conflicts with future MIPS references, you should begin at 255 and work downwards in assigning major numbers.

## Copy and Rename the Kernel and Sygen Files

Make copies of the two configuration files *kernel.suffix* and *sysgen.suffix*. The default files for the M/120 are *kernel.r2400_std*, and *sysgen.r2400_std*. Do not change the default files, but instead make copies of them with a new suffix appended to them.

For example, to recopy and rename the default files, type:

```
# cp kernel.r2400_std kernel.r2400_new
# cp sysgen.r2400_std sysgen.r2400_new
```

In Makefiles, the *suffix* after the last '.' is an identification used to prevent making multiple kernels which overwrite each other. For example, to build a kernel with the default files listed above, the kernel would call *unix.r2400_std.* The suffix is tacked onto the end of the kernel.

### Modify the New Kernel File

For most situations, you won't need to modify the *kernel.suffix* file. This file is the kernel's master file. It contains all tunable kernel parameters. You would use this file, for example, to change the number of process table entries. For more information, see **master(4).**

### Modify the New Sysgen File

You will, however, have to modify the *sysgen.suffix* file. This file is used by the **mboot** program to obtain configuration information. This file generally contains information used to determine if specified hardware exists, a list of software drivers to include in the load, the assignment of system devices such as pipedev and swapdev, and instructions for manually overriding the drivers selected by the self–configuring boot process. For more information, see the **system(4)** manual page.

**Including a Driver.** There are three directives that you use in your *sysgen* file to include a driver into the kernel, VECTOR, INCLUDE, and ATBUS.

- The VECTOR command describes whether the device is to be an AT device or hardware that exists on the main CPU board.
- The INCLUDE command specifies which software drivers to include. These are software drivers that have no hardware interrupts associated with them. A good example of this is the shared memory (shm) subsystem in the kernel.
- ATBUS is a new directive that includes a specific AT bus device into the kernel. It works the same as the VECTOR directive but contains slightly different information.

**Specifying Address Space.** Both the VECTOR and ATBUS commands have a base specifier that specifies the base address for the device. For ATBUS, refer to the AT bus table for an unused address range. For VECTOR specifications, there is a vector specifier for each device. Since UMIPS does not autovector, you must choose this yourself. Make sure that it does not conflict with vectors specified in other VECTOR lines in the *sysgen* file.

### Build the Kernel

Once you have completed your work on the *sysgen* file and the *kernel* file, build the kernel. Go to the *$ROOT/usr/src/mips* directory and type:

```
make unix.suffix
```

The *suffix* name is the same name that you applied to your *kernel.suffix* and your *sysgen.suffix* files.

Once the make has completed, a kernel named *unix.suffix* is made and placed in the current directory.

## M/120 Machine Considerations

This section describes specific information for the M/120 that you need to know when you are writing device drivers.

### The AT Bus

The AT bus is an industry standard device interfacing bus. AT bus boards are usually shipped with an interrupt request signal already assigned. Use this assignment if it does not conflict with existing assignments.

The kernel interrupt handler can accept multiple interrupts per interrupt request level (*irq*). For AT bus specifications, an *irq* (interrupt request) specifies an AT bus interrupt request level for the driver. For example, every time an *irq3* (interrupt request, level 3) arrives, all *irq3* boards' interrupt routines are called.

AT bus *irq*s are *or*ed together to generate a hardware level 0 interrupt. Then, the level 0 interrupt handler determines AT pending bus interrupts by reading the ISR (Interrupt Status Register) and services them by priority, based on *irq*. Refer to **Chapter 3**, for a description of the ISR.

For example, if an irq5 comes in while an irq3 is being serviced, it must wait for the next interrupt. This is because the interrupt handler takes a snapshot of the pending interrupts, and services them. Because all AT bus, Ethernet, SCSI, and UDC interrupts come in on the same hardware level, they cannot preempt each other. The details of the AT bus operation are described in **Appendix A** and in **Chapter 3**.

## AT bus Address Space

Figure 4.1 illustrates the address space for the AT bus devices. Note that the only preallocated address space is for the Digiboard.

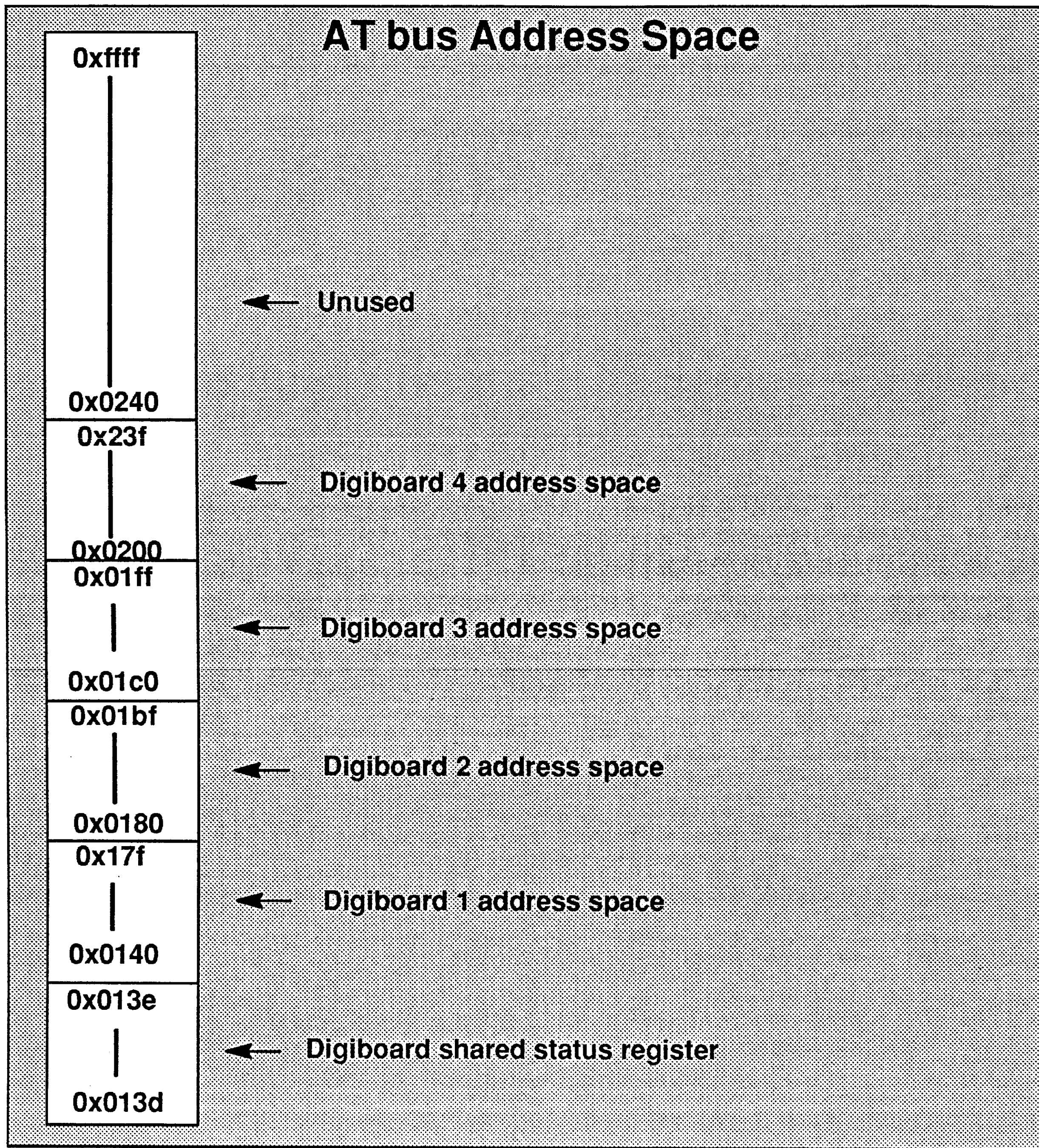



*Figure 4.1. Address space for AT bus devices.*

## Kernel Support Routines

There are two kernel support routines:

- a *DELAY(n)* macro routine
- address translation routines

### Delay(n) Macro

Because MIPS RISComputers are faster than a device that is attached to the system, you need to cause a delay in order to do successive writes to the device. The DELAY(n) macro provides this function. When a system boots, the *DELAY(n)* macro computes a delay factor based on the machine's speed. This macro uses this factor to automatically adjust for system speed when the system boots. This delay factor makes it easy to move the same binary kernel from a slower to a faster system. The *DELAY(n)* macro resides in */usr/src/sys/param.h.*

### Address Translation

Often, you must include code in your driver that translates a virtual address to a physical address; the driver requires this translation to pass the address to a device. Translation is a two step procedure.

In the first step, the driver determines which segment it is translating using the macros *IS_KSEG0()*, *IS_KSEG1()*, *IS_KSEG2()*, and *IS_KUSEG()*. These macros return a 1 if the address is within the segment, or a 0 if the adress is external to a segment. For details about these macros, see */usr/src/uts/mips/sys/sbd.h* and */usr/src/uts/mips/immu.h.*

In the second step, the driver translates addresses according to the segment containing the address. If the address is in K0 (known as kseg0) or in K1 (known as kseg1), then you can use the macros *K0_TO_PHYS()* and *K1_TO_PHYS()* to translate the address to physical space.

```
physaddr = k0_TO_PHYS(addr); /*For K0 addr */
physaddr = K1_TO_PHYS(addr); /*For K1 addr*/
```

However, if the address you want to translate is in K2 (known as kseg2) or KUSEG, then the translation is more difficult.

To convert a K2 address to a physical address use the following:

```
physaddr = ctob(kvtokptbl(addr) ->pgm.pg_pfn) | (addr & PGOFSET);
```

To convert a KUSEG address to a physical address use the following:

```
physaddr = ctob(vtop((unsigned)addr, u. u_procp) ->pgm.pg_pfn) |
                                           (addr PGOFSET);
```

## Interrupt Priority Level Assignment

The kernel assigns all interrupts in the kernel *master.d/sysgen* file. The operating system does not check to see which devices are connected for autoconfiguration. The *master.d/sysgen* file describes the hardware configuration. The makefiles massage the *master.d/sysgen* file into *master.d/system*. For more information about this file, see **Adding New Drivers** in this chapter and the manual page **system(4)**.

The MIPS R2000 processor has eight interrupt levels: six hardware levels and two software–defined levels. The interrupt levels 8 through 3 map to the hardware interrupt levels 5 through 0 respectively. In the M/120 system, these interrupts are used as follows:

| Hardware Interrupt Level | Description |
|---|---|
| level 8 | write bus error from memory |
| level 7 | profiling clock, if enabled |
| level 6 | floating point |
| level 5 | scheduling clock |
| level 4 | duart |
| level 3 | vectored AT bus and on–board devices |

| Software Interrupt Level | Description |
|---|---|
| level 2 | software network |
| level 1 | software clock |

### Changing Interrupt Levels

A driver uses the following routines to change interrupt levels:

| Routine | Description |
|---|---|
| *spl0()* | block no interrupts |
| *splsoftclock()* | block software clock interrupts |
| *splnet()* | block software network interrupts |
| *splimp()* | block network device and duart interrupts |
| *splbio()* | block VMEBus and AT bus device interrupts, SPC, and UDC |
| *spltty()* | block tty device interrupts (duart, VMEBus, and AT bus) |
| *splclock()* | block scheduling clock and floating point interrupts |
| *splhigh()* | block all interrupts |

Each routine returns the old interrupt priority level. The driver should save this value and use it as a parameter to the *splx()* routine. This routine restores the previously saved interrupt priority level.

## Kernel/PROM Interface

When writing a device driver, you must be aware of the interface between the kernel and the PROM monitor. The entry points to the PROM monitor are in the file *$ROOT/usr/src/uts/mips/sys/firmware.h.* See **Chapter 5** for information on the facilities of the PROM Monitor and the PROM monitor commands.

# Memory Management

When writing device drivers, you need to understand how the following software topics are handled in M/120 machines:

- the memory management system and cache control, as implemented in the architecture and described in the *R2000 RISC Architecture* book.
- the effects of the optimizing compilers and specifying volatile memory
- the M/120 FIFO write buffer

Memory management and cache control are discussed in the  manual.

## Volatile Memory and Optimizing Compilers

Advanced optimizing facilities in the compiler system improve the performance of the object programs by getting rid of unused calls, variables, and so on. If you were to look at your program before and then after it has gone through an optimizer, it could be rearranged. For example, the following program segment illustrates some code before optimization:

```
unsigned *control_reg = 0xfff035a4;
unsigned *data_out_reg = 0xfff035a8;
unsigned i, buf_len, buffer[1024];
...
*control_reg = 0x0000ff01; /*Set controller for "output" */
for (i = 0; i < buf_len; i++)
  {
   *data_out_reg = buffer[i]; /*Prepare word of data */
   *control_reg = 0x80000000; /*Pulse output strobe line */
   }
```

The compiler, which assumes the device registers are ordinary memory locations, also assumes that only the last value stored into each variable matters; therefore, it might optimize the code to behave like this:

```
*control_reg = 0x0000ff01;
i = buf_len -1;
*control_reg = 0x80000000;
*data_out_reg = buffer[i];
```

**NOTE:** The compiler has also swapped two statements, probably to make better use of the CPU pipeline.

For memory–mapped devices, a driver often must store into or load from a variable in precisely the right sequence to cause I/O transfers. However, some of the operations in an optimized program may be eliminated or rearranged. To prevent this problem, you can either never optimize your code or you can declare the ANSI C *volatile* storage class on any variable that has hardware "side effects". For example:

```
volatile unsigned *control_reg = 0xfff035a4;
volatile unsigned *data_out_reg = 0xfff035a8;
```

If you are unable to add the volatile declaration to your program, then you can specify the **–volatile** compiler option, which is like declaring every variable to be "volatile". A good strategy when you port an existing driver is to suppress optimization or use **–volatile** until you get the driver working. When the driver is working, set only the declarations that need to be volatile. In general, expect that every pointer to a structure of device registers needs to be volatile. See **Appendix E**, page E–5 for an example of how the volatile attribute is used.

### Write Buffer Considerations

The M/120 provides a four–word deep FIFO (first–in, first–out) write buffer that captures all data and the associated addresses written by the processor. The 32–bit wide write buffer subsequently passes the captured data through to the specified address when it can obtain access to the system buses. This technique lets the processor quickly output data without being limited by slower response time of main memory and without competing for access to the AT bus or private memory bus. The write buffer also lets the processor perform memory read operations without waiting for a previous write to be completed.

Although the write buffer enhances the performance of the memory system, it can introduce problems when programs are writing information to I/O devices. You must consider three special situations.

- To reduce the number of write operations, the write buffer sometimes collapses sequential words written to the same word address into a single FIFO entry. Thus, if you write consecutive words to an address, not all of the data may be written out to the device.
- If bytes or half–words of data are written to the same word address, the write buffer may gather them into a single word. However, the write buffer may not write these bytes out in the same sequence in which they were written by the processor. Although this is not a problem when writing to memory, it can introduce problems when writing to I/O devices or to the AT bus.
- The write buffer includes logic that guards against the processor reading from an address before the write buffer has completed a preceding write to that same word address. However, if the read and subsequent write are to different addresses, then the write (for example, to a device's control register) may not complete before an attempt is made to read (for example, from the same device's data or status register.).

To avoid this problem, programs performing I/O must use the *wbflush()* routine after write operations. This routine empties the write buffer and thus ensures that data actually passes through the write buffer before a subsequent write or read is performed.

## SCSI Devices

As delivered, the M/120 supports two types of SCSI devices; 5 1/4" hard disks and 1/4" tape. Devices such as optical disks, floppy disks, and 1/2" tape are not supported. Modifications to the SCSI driver source code are necessary to support these devices.

All internal SCSI hard disks should work with the SCSI driver. The operating system requires a volume header containing device geometry and partition information to be written to block 0 of each disk drive.

The source code for the M/120 SCSI device driver has been 'modularized' into three files: a high–level driver named *scsi.c* and two low–level source files named *spc.c* and *spc_poll.c*. The high level driver is a 'generic' hard disk and 1/4" tape driver. This driver calls *spc.c,* which contains low–level interrupt driven routines for the SPC (Fujitsu SCSI protocol controller) and UDC (AMD universal dma controller) chips. For early pre–interrupt conditions such as autoconfiguring, calls are made to *spc_poll.c,* which contains polled mode routines to drive the SPC and UDC chips. The low–level code is designed as an interface to higher level drivers such as scsi.c, thus avoiding low–level hardware details.

For other devices, you probably need to either modify the scsi.c driver or use it as a model to produce your own high–level driver code. In the low–level files, you need only modify the specific routines that *scsi.c* calls.

The standalone program **format**, located on the original installation tape shipped with the M/120 or in the */stand* directory, is used to install the volume header for a drive. The format program is preconfigured for the following drives: CDC 94161, CDC 94171, Fujitsu 2249, and Fujitsu 2246sa. Other types of SCSI drives need to be configured manually using format program interaction. See the **format(8)** manual page or **Chapter 5** in this manual for more information.

For other hard disks, you must also make an entry in the UMIPS disktab database */etc/disktab* in order to be able to build file systems on the drive. See the **disktab(5)** manual page for information.

SCSI Logical Units (LUN) other than 0 (embedded SCSI) are supported in software. In some cases the drivers have to be modified. If you are using SCSI controllers that support more than one LUN, consult MIPS Computer System customer support for help. The telephone number is listed on the inside cover of this book.

# Debugging Drivers

This section describes certain situations that you need to deal with when debugging your new device drivers. You can use the dbx debugging facility for this purpose; refer to **dbx(1)** in the MIPS Reference Manual for details.

The System Programmer's Package (SPP), a separate product available from MIPS Computer Systems, also provides extensive facilities for debugging device drivers and other software implemented on RISComputer systems. If you have access to this product, refer to the *System Programmer's Package Reference* manual for additional information.

## Halting the System

At times, device drivers encounter error conditions that require the attention of someone at a system console. If you need to halt the system, use caution. Except during debugging, a driver should halt the system only for errors that affect the operation of the entire system.

The *panic()* function, which is called when unresolvable fatal errors happen, halts the machine. This function, which should not be called directly (see the discussion below in the System Error Messages section), accepts as arguments a message (character string) to be printed on the system console. The *panic()* function does the following things:

- identifies the reason for panic
- saves the state of the machine
- exits the operating system by returning to the firmware

## System Error Messages

Drivers should not call the kernel *panic()* or *printf()* directly. Instead, the kernel provides the function *cmn_err()* that calls *printf()* or *panic()*. In the following example, ARGS represents a *printf()* argument string with a maximum of six arguments.

```
#include <sys/cmn_err.h>
cmn_err(level, ARGS)
int level;
```

The *cmn_err()* function is passed two arguments:

- the first argument is a defined constant, which shows the severity level of the error condition
- the second argument is the set of arguments that would be passed to *printf()*

The first argument has these four severity levels:

- CE_CONT specifies that the error message is a continuation of the previous message. Use this level when the error message is too long to be passed as one string.
- CE_NOTE reports system events that do not necessarily require user action, but that might be of interest to the user. For example, a sector on a disk that needs to be accessed repeatedly before it can be accessed correctly might qualify as such an event.
- CE_WARN reports system events that require immediate attention; that is, if no action is taken, the system might panic. For example, when a peripheral device does not initialize correctly, use this level.
- CE_PANIC results in a system panic. Drivers should specify this level only when the error condition means the system cannot continue to function.

**NOTE:** You can use *cmn_err()* with the CE_NOTE argument and the kernel *printf()* function as a debugging tool; however, this approach changes system timing characteristics.

The *cmn_err()* function has an additional feature. If the first character of the second argument (the *printf()* format string) is an exclamation mark (!), the *cmn_err()* output goes into the kernel buffer *putbuf[]*, that is set aside for this output. If the first character of the second argument is a caret (^), then the output goes only to the system console; otherwise, output goes to both the kernel buffer *putbuf[]* and the system console. You can examine *putbuf[]* only from the debugger. If you are debugging a driver with some timing dependencies, you might want to send the output to the kernel buffer instead of the system console.

# Chapter 5
# PROM Monitor

## Introduction

This chapter describes the PROM Monitor. The PROM Monitor provides the tools for examining and changing PROM memory, downloading programs over serial lines (RS–232C), and booting programs from disk, tape, or Ethernet. The PROM Monitor also provides tools for altering configuration power–up options in non–volatile RAM.

### Description

The PROM Monitor resides in PROM on the Motherboard and is entered when the system is reset or the system is powered up. The PROM Monitor initializes the R2000 processor, the R2400 CPU Card, and the R2450 Memory Cards.

The R2000 processor is initialized by initializing the system coprocessor Status and Cause registers and flushing the translation buffer.

The R2400 CPU Card is initialized by sizing and flushing the instruction and data caches, by inspecting the contents of non–volatile memory and reinitializing it if necessary, and by initializing environment variables from non–volatile memory.

The R2450 Memory Cards are initialized by probing to determine how many cards exist, determining the best memory interleave configuration, configuring the boards for refresh slot assignment, and assigning base addresses.

### Memory Usage

The PROM Monitor uses system memory between physical addresses 0x500 and 0x10000. The include file *prom/entrypt.h* describes conventions for memory use by standalone programs.

## File Name Syntax

When the PROM Monitor program requires a file name, the file name is constructed in different ways depending on the device. The different file name formats are shown below, and Table 5.1 describes the different parts of the file names.

| | |
|---|---|
| **SCSI disk** | **dkis (LUN, target, partition) path** |
| **SCSI tape** | **tqis (LUN, target, partition) path** |
| **Console uart** | **tty (port #)** |
| **Pseudo console** | **console (port #)** |
| **Boot server** | **bfs ()** |

**Table 5.1. File Name Syntax**

| File name Part | Description |
|---|---|
| **LUN** | Logical Unit Number. Each SCSI target can have up to 8 (0–7) logical units, but MIPS currently uses only embedded SCSI devices, which only support LUN 0 (zero). |
| **target** | The target number indicates the embedded SCSI device from 0–5, with 0 being the main cabinet device. Targets 1–5 are the SCSI devices in the Expansion Cabinet. The tape drive is hard–wired to device 6, but it is simply entered as 0 (zero) for the tqis device. If you do not specify a unit number, the default value of 0 is used. |
| **partition** | Disk devices are frequently broken down into logical subunits, called partitions. The partition field selects a disk partition within a unit. The partition's base cylinder and size is determined by accessing the disk volume header stored on the disk itself. If you do not specify the partition field, the default value of 0 is used. For Tape devices, this field specifies the number of the file on the tape. Files are numbered on the tape starting with zero. |
| **path** | The path indicates a particular file on the media specified by the device, controller, unit, and partition fields. The file referred to by path is located by consulting a directory located on the device itself. If you do not specify a path, the file name is assumed to refer to the raw device. |
| **port #** | The port # field indicates the serial I/O port number. This number can be either 0 (zero) or 1 (one). |

## Environment Variables

The PROM Monitor maintains environment variables that are passed to booted programs. These variables function like UNIX system shell environment variables. Some of the environment variables affect the operation of the PROM Monitor and are maintained in non–volatile memory. This means that when you reset the machine or power it down, the Monitor still maintains these variables. The PROM Monitor variables are defined and described in Table 5.2.

**Table 5.2. PROM Monitor Environment Variables**

| Variable | Description |
|---|---|
| **netaddr** | Specifies the internet address for the node. This is used by the **bootfile** service software in the standalone I/O (saio) library. |
| **lbaud** | Specifies the baud rate for tty(0), which is uart A on the R2400 CPU Card and typically the local console. You can set the baud rate to: 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, or 19200. If you specify an illegal baud rate, 9600 baud is used. |
| **rbaud** | Specifies the baud rate for tty(1), which is uart B on the R2400 CPU Card and typically the remote console. You can set it to: 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, or 19200. If you specify an illegal baud rate, 9600 baud is used. |
| **bootfile** | Specifies the default program that boots when you don't specify the **–f** option to the **boot** command. |
| **bootmode** | Controls the PROM Monitor's action in response to system resets. If **bootmode** is **m,** then the PROM Monitor enters the command mode after a reset. If **bootmode** is **c**, then the PROM Monitor does a cold boot. A cold boot loads the file specified by the environment variable **bootfile** and passes it the argument **–a**. Typically, the **bootfile** is the standalone shell (sash). The sash interprets the **–a** option as a request to load the operating system as specified in the volume header of the device from which sash loaded. If the **bootmode** is **w,** then the PROM Monitor attempts a warm boot on reset. A warm boot transfers control to a memory image that was loaded before you reset the system. The PROM Monitor looks for a properly formatted restart block to determine if the memory image is present. A cold boot is performed if one of the following occurs: the restart block is incorrectly formatted, the PROM Monitor does not find a restart block, or a warm boot has already been attempted with the restart block. If the **bootmode** is **d**, then the PROM Monitor enters command mode immediately and preserves the contents of memory across resets. For all other modes besides **d**, the Power On Diagnostics are run. |
| **console** | This variable selects which console devices are to be considered consoles on power–up and after resets. When set to 'l' (the letter L), only tty(0) is initially enabled as a console. If console is 'r', both tty(0) and tty(1) are enabled as consoles. You can enable and disable consoles by command after a reset. Refer to the **enable** or **disable** command pages. |

**Table 5.2. PROM Monitor Environment Variables** (continued)

| Variable | Description |
|---|---|
| **cpuid** | Reserved for future use. Currently this variable must be set to zero. |
| **resetepc** | This variable indicates the program counter the machine was executing when the machine was reset. |
| **resetra** | This variable indicates the contents of the Return Address register when the machine was reset. |
| **memparity** | Setting this variable to one (1) enables parity, and setting this variable to zero (0) disables parity. This variable should be used in conjunction with the kernel argument "disable_parity" to enable and disable parity when running UNIX. |
| **version** | This variable indicates the version of the installed PROMs, and it is used by the kernel to determine which PROMs are installed in the machine. This environment variable cannot be changed. |

## Input Editing

Table 5.3 lists the basic editing commands available for the PROM Monitor.

**Table 5.3. Basic Editing Commands**

| Command | Description |
|---|---|
| **Control–H or DEL** | Erases the previous character. |
| **Control–U** | Erases the entire line. |
| **Control–C** | Aborts the program that is currently running and returns control to the PROM Monitor. |
| **Control–Z** | Causes the current program to execute a breakpoint instruction. This command is used in conjunction with the standalone program dbgmon. |
| **Control–D** | Causes the standalone program to exit normally. |

## Time of Day and Non–Volatile RAM

The PROM Monitor initializes the non–volatile RAM (NVRAM) and the time–of–day clock, which resides in the MK48T02B real time clock chip on the R2400 CPU Card. A single location in non–volatile RAM location on this chip is reserved to indicate to the operating system kernel whether the time–of–day clock is valid. This location is defined as NVADDR_STATE in the PROM include file *prom.h.*

NVADDR_STATE is an offset relative to the general purpose NVRAM on the MK48T02B. Two bits are defined in this byte location to indicate the validity of the time–of–day clock and the rest of non–volatile RAM. If the PROM considers the time–of–day clock valid and clear, it sets NVSTATE_TODVALID. NVSTATE_RAMVALID is a bit in the NVADDR_STATE location that the PROM Monitor uses internally. This bit is generally not of interest to the operating system.

The operating system can verify whether the time–of–day clock is valid by checking that the VRT bit in register D of the MK48T02B real time clock is asserted and that the NVSTATE_TODVALID bit is set in the non–volatile RAM location NVADDR_STATE. The reading of register D of the MK48T02B sets VRT; therefore, this bit is read–once. If either of these bits is not set, the operating system should correctly set the time in the MK48T02B and logically OR NVSTATE_TODVALID into location NVADDR_STATE in the non–volatile RAM. The PROM Monitor and UMIPS kernels both use the time–of–day clock to maintain time as the number of seconds from the beginning of the current year. To calculate seconds from the beginning of the year, the time–of–day clock is always set to some date in the year 1972 and the offset in seconds is calculated from this base.

## Using Breaks to Change Baud Rate

You can also cycle the baud rate for tty(0) and tty(1) among the baud rates, 110, 300, 1200, 2400, 4800, 9600, and 19200 by entering a BREAKs. Baud rate changes made by BREAKs are temporary until the next reset or until a new program is loaded. To change the baud rate permanently, change either the **lbaud** or **rbaud** environment variable.

## Extending the PROM Monitor

If you give the PROM Monitor a command that is not built in, then the Monitor uses the first word of the command as the name of a file and tries to boot that file passing any other arguments on the command line onto the booted program. If the environment variable **path** is undefined, then the first word of the command must be a complete file name specification, consisting of a device name, controller, unit, partition fields as necessary, and a file path. If the environment variable **path** is defined, the PROM Monitor tries to boot the program file formed by prepending the contents of **path** to the command. If **path** is a list of prefixes separated by spaces, then the PROM Monitor tries each prefix from **path** until the file boots successfully or all prefixes have been tried.

# Command Set

The PROM Monitor commands are listed and described in Table 5.4. Following this table, each PROM Monitor command is described on a separate page.

### Table 5.4. PROM Monitor Commands

| Command | Description |
|---|---|
| **auto** | Initiates the two–level operating system autoboot sequence. |
| **boot** | Loads the specified program. |
| **cat** | Displays the contents of the files listed on the console. |
| **disable** | Does not allow input from and output to the specified console device. |
| **dump** | Formats and displays the contents of memory. |
| **enable** | Allows input from and output to the specified console device. |
| **fill** | Fills the specified range of memory with the specified pattern. |
| **g** | Displays the contents of a single memory location in decimal, hexadecimal, and ASCII character formats. |
| **go** | Transfers control to code that is assumed to have been previously loaded. |
| **help** | Displays the syntax for all commands. |
| **init** | Reinitializes the PROM Monitor software state. |
| **init_tod** | Initializes the time–of–day chip. |
| **load** | Allows you to load memory over a serial line connection. |
| **p** | Puts or sets the contents of a single memory location to a specified value. |
| **printenv** | Displays the value of the PROM environment variables. |
| **pr_tod** | Prints the contents of the time–of–day register. |
| **setenv** | Used to create a new environment variable or to change an existing environment variable. |
| **sload** | Accepts a subset of the Motorola S–record protocol. |
| **spin** | Generates reference patterns for diagnostic use. |
| **unsetenv** | Used to delete an existing environment variable. |
| **warm** | Examines memory for a restart block. |

# auto

## Synopsis

**auto**

## Description

The PROM Monitor **auto** command initiates the two–level operating system autoboot sequence. Once initiated, this sequence waits for about 20 seconds. During this delay, you can abort the autoboot by typing a Control–C at the console, or you can expedite the boot process by pressing the **Enter** key on the keyboard. When the delay expires or when the **Enter** key is pressed, the program specified by the PROM Monitor environment variable **bootfile** is loaded and passed the current environment and the argument **–a.**

The default setting for the **bootfile** environment variable is *dkip(0,0,8)sash.* For the M/120, the **bootfile** setting should be *dkis(0,0,8)sash,* which will need to be set using the **setenv** command.

# boot

## Synopsis

**boot** [–f file] [–n] [ args ]

## Description

The **boot** command loads the program specified by the **–f** option. If **–f** is unspecified, **boot** loads the file specified by the environment variable **bootfile**. If **–n** is specified, **boot** loads the requested file, but does not transfer control to the program. The program can be initiated later using the **go** command, but no arguments can be passed in this case. If present, *args* are passed to the program and are accessible from the standard *argc, argv* mechanism. Any argument that begins with a "–" must be prepended with an additional "–"; this extra dash is removed before the argument is passed to the program. The current environment is passed to the program as the third parameter to the main routine and also from the external variable *environ.*

## Environment Variable

If the environment variable **path** is defined and the **boot** command has a file to load that does not have a device specification, **boot** tries to load a file name formed by prepending the contents of **path** to the original file name. If **path** is a list of space separated prefixes, the **boot** command tries each prefix from **path** until the file can be successfully booted or all prefixes have been tried.

## See Also

**go, load,** and **sload**

# cat

## Synopsis

**cat** [ files ]

## Description

The **cat** command displays the contents of the listed files on the console.

The PROM Monitor cannot locate files on filesystems. This command would be better used through sash.

# disable

## Synopsis

**disable** [console–dev]

## Description

The **disable** command does not allow input from and output to the specified console device. Using the **disable** command without an argument displays the current set of enabled console devices.

## See Also

**enable**

# dump

## Synopsis

**dump** [ format ] [ width ] range

## Description

The **dump** command formats and displays the contents of memory. You can display the contents of memory in hexadecimal, octal, decimal, unsigned decimal, ASCII, or binary. The contents of memory can be dumped in byte, halfword, or word size units.

The default for **format** is hexadecimal (**–x**). You can select an alternative **format** by entering one of the following characters in the command line as an argument.

| Character | Format |
|---|---|
| **–B** | binary |
| **–c** | ASCII character |
| **–d** | decimal |
| **–o** | octal |
| **–u** | unsigned decimal |
| **–x** | hexadecimal |

The default for **width** is word (32 bits). An alternative **width** can be selected by entering one of the following characters on the command line as an argument.

| Character | Width |
|---|---|
| **–b** | byte (8 bits) |
| **–h** | halfword (16 bits) |
| **–w** | word (32 bits) |

# dump (continued)

The **range** specification indicates the amount of memory to be displayed. You can specify the **range** in one of the following ways.

| Range | Description |
|---|---|
| **base** | Displays the contents of the memory address at *base*. |
| **base#count** | Displays the contents of memory starting at *base* and ending at *base* + *count*. |
| **base:limit** | Displays the contents of the memory addresses starting at *base* and ending at *limit*. |

## Example

The following example shows a **base#count** range specified in **halfwords**. The default for the format of **hexadecimal** is used because no argument was specified. The specified range is displayed on the screen horizontally.

```
>>dump -h 0xbfc04000#5

0xbfc04000:  8dce  514   6  6900  1a3
```

## See Also

**g**, **p**, and **fill**

# enable

## Synopsis

**enable** [ console_dev ]

## Description

The **enable** command allows input from and output to the specified console device from the PROM Monitor. Specifying the **Enable** command without arguments displays the current set of enabled console devices.

## See Also

**disable**

# fill

## Synopsis

**fill** [width] [–v val] range

## Description

This command fills the contents of a specified **range** of addresses with **val.**

The default for **width** is word (32 bits). An alternative **width** can be selected by entering one of the following characters on the command line as an argument.

| Character | Width |
|---|---|
| **–b** | byte (8 bits) |
| **–h** | halfword (16 bits) |
| **–w** | word (32 bits) |

The **range** specification indicates the amount of memory to be displayed. You can specify the **range** in one of the following ways.

| Range | Description |
|---|---|
| **base** | Fills the contents of the memory address at *base*. |
| **base#count** | Fills the contents of memory starting at *base* and ending at *base* + *count*. |
| **base:limit** | Fills the contents of the memory addresses starting at *base* and ending at *limit*. |

#

## Synopsis

**g** [ width ] address

## Description

The **g** (Get) command displays the contents of a single memory location in decimal, hexadecimal, and ASCII character formats.

The default for **width** is word (32 bits). An alternative **width** can be selected by entering one of the following characters on the command line as an argument.

| Character | Width |
|---|---|
| **–b** | byte (8 bits) |
| **–h** | halfword (16 bits) |
| **–w** | word (32 bits) |

## See Also

**p, dump,** and **fill**

#

## Synopsis

go [ entry ]

## Description

The **go** command transfers control to code assumed to have been previously loaded with the **boot, load,** or **sload** commands. The **entry** argument is the address of the entry point. If you do not specify **entry**, then the **go** command transfers control to the entry point of the last loaded (or booted) module.

## Bugs

When an entry point is not specified, **go** does not check that a module has previously been loaded.

## See Also

**load, sload,** and **boot**

# help

## Synopsis

**help** [ commandlist ]

## Description

The **help** command displays the syntax for all the commands in **commandlist**. The **commandlist** argument can be one or more commands separated with a space. If you do not specify a **commandlist**, then the **help** command displays the syntax for all commands. You can also get help by typing a question mark (?), which also displays the syntax for all commands.

# init

## Synopsis

**init**

## Description

The **init** command reinitializes the PROM Monitor software state; however, the environment variables that are stored in non–volatile ram are preserved.

# init_tod

## Synopsis

**init_tod** [ secs ]

## Description

This command initializes the time–of–day chip. It is very important that the time–of–day chip is running; otherwise, the operating system will not work properly. This command is normally executed at the factory.

**Secs** is the number of seconds since 1972. Refer to the previous section entitled **Time of Day and Non–Volatile RAM** for additional information. The recommended method is to type the **init_tod** command without any arguments, and then run the **date(1)** command after the operating system has been booted.

## See Also

**pr_tod**

# load

## Synopsis

**load** console_device

## Description

The **load** command allows you to load memory over a serial line connection from a system running the RISC/os program **tip (1)**. To download an image, use the **tip** command to establish communication with either the local or remote console port of the machine to be downloaded. For additional information, refer to the **tip (1)** command in the User's Reference Manual.

If you transfer data to the remote port, be sure that the remote port is enabled. Refer to the **enable** command for additional information. After trying several PROM Monitor commands to verify that **tip** is communicating successfully with the remote port, enter the **load** command, specifying either tty(0) or tty(1) to reflect the serial port with which **tip** is communicating. After the **load** command, the PROM expects you to download an image. If you want to abort this mode, type a Control–C. To download the image, refer to the **tip** command in the User's Reference manual. The PROM Monitor returns to command mode after the download completes. Use the PROM Monitor **go** command to run the downloaded program.

## See Also

**enable, go,** and **sload**

# p

## Synopsis

**p** [ width ] address value

## Description

The **p** (Put) command sets the contents of a single memory location (**address**) to **value**.

The default for **width** is word (32 bits). An alternative **width** can be selected by entering one of the following characters on the command line as an argument.

| Character | Width |
|---|---|
| **–b** | byte (8 bits) |
| **–h** | halfword (16 bits) |
| **–w** | word (32 bits) |

## See Also

**g, dump,** and **fill**

# printenv

## Synopsis

**printenv** [ varlist ]

## Description

The **printenv** command displays the value of the PROM environment variables in **varlist**. The **varlist** argument can be one or more variables separated by a space. If no environment variables are specified, **printenv** shows all currently defined environment variables.

## See Also

**setenv,** and **unsetenv**

# pr_tod

## Synopsis

**pr_tod**

## Description

This command prints the contents of the time–of–day register from the time–of–day chip. You can determine whether the time–of–day chip is functioning by running this command twice. If you run the command and then run it again 5 seconds later, then the value displayed the second time should be about 5 more than the first time. If the displayed value does not change or if the value decreases, then run the **init_tod** command to correct this situation.

## See Also

**init_tod**

# setenv

## Synopsis

**setenv** var value

## Description

The **setenv** command is used to create a new environment variable or to change the value of an existing environment variable. Environment variables are represented as ASCII strings. The current values of the environment variables are passed to programs booted by the PROM Monitor. Refer to Table 5.2 at the beginning of this chapter for a list of PROM Monitor environment variables.

## See Also

**printenv** and **unsetenv**

# sload

## Synopsis

**sload** [ –a ] console_device

## Description

The **sload** command accepts a subset of the Motorola S–record protocol. The record types accepted are 0, 3, and 7. You can use the System Programmer's Package command **convert** to produce S–record images.

If you do not specify **–a**, the PROM replies with an ASCII ACK to each S–record received that has a valid checksum. The PROM replies with an ASCII NACK for records that have incorrect checksums.

## Bugs

The **sload** command has not been debugged. Consider it a starting point.

## See Also

**load**

# spin

## Synopsis

**spin** [[ –c count ] [ –v value ] –(r|w)(b|h|w) address ] [ –c count ]

## Description

The **spin** command generates reference patterns for diagnostic use. You can specify a sequence of reads and/or writes of byte (**b**), halfword (**h**), or word (**w**) width with combinations of **–r** and **–w** options. The **–c** option specifies the repetition or copies count that applies to all subsequent reads and writes. The **–v** option indicates a value for writes that applies to all subsequent writes. A final **count** specification indicates the number of times the entire preceding pattern should be repeated. **Count** defaults to 1 and **value** defaults to 0. A negative **count** is interpreted as infinity.

## EXAMPLE

```
>>spin -c 2 -v 1 -wb 0x4 -c 4 -rh 0x2 -c 10
```

The example shown above repeats the following two instructions 10 times.

Write 1 to the byte at address 0x4 two times
Read the halfword at address 0x2 four times

# unsetenv

## Synopsis

**unsetenv** var

## Description

Use **unsetenv** to delete an existing environment variable.

## See Also

**printenv,** and **setenv**

# warm

## Synopsis

**warm**

## Description

The **warm** command examines memory for a restart block. If a correctly formatted restart block is found, control transfers to the existing memory image at the entry point given in the restart block. A restart block contains information that tells how to re–enter an existing image. Typically, the existing image has earlier aborted or terminated due to a device failure.

# Appendix A
# AT Bus Compatibility Considerations

The M/120 system supports expansion through an implementation of IBM's PC/AT bus. This four–slot bus is designed to support virtually any card that works in a PC, PC/AT or equivalent machine. Integration of most AT cards into the M/120 system should be straightforward. However, some cards may make assumptions about the bus implementation that are not valid in this instance. Therefore, to make it easier to determine the compatibility of a particular card with the M/120 system AT bus, this appendix describes the guidelines and assumptions that were followed in the M/120 system implementation of the bus.

The description of the bus found in the *IBM RT PC Technical Reference* has been used as a guide to the bus implementation. The AT bus interface performs all functions necessary to let the MIPS processor function as a master or a slave on the AT bus. The AT bus is coupled via address and data transceivers to the M/120 I–bus (see the block diagram in Figure 3.1). The AT bus interface generates all of the necessary strobes, handshakes, setup and hold times, and so on, to let cards designed for an IBM PC/AT function on this AT bus also. The primary functions that the AT bus interface provides are:

– Handshake to and from the I–Bus
– I/O Read and Write Cycle timing on AT bus
– Memory Read and Write Cycle timing on AT bus
– DMA 8–bit Read and Write Cycle timing on AT bus
– DMA 16–bit Read and Write Cycle timing on AT bus
– Memory refresh
– An interrupt path from AT bus devices to the R2000 processor

## AT Bus Memory Mapping Options

The M/120 processor can directly access 16 MBytes of memory and I/O addresses in the AT bus address space for 8– and 16–bit transfers. One DMA channel is dedicated to the AT bus and this channel can be programmed to directly control DMA devices or to perform memory–style operations.

AT bus master cards (*Alternate Controllers*) can access other AT 16–bit cards and the M/120's main memory. A one MByte section of the M/120's memory can be mapped into the AT bus. The AT address at which the mapping occurs is set with jumpers on the motherboard. The default setting of the jumpers specifies an AT address with the most–significant bits (A23..A20 ) set to "1". The address in M/120 main memory that is mapped is set via the *AT Control Register* described in **Chapter 3, Programming Model**. The base of the 1MB

mapped section can be set at any 1MB boundary in AT and M/120 space. Figure A.1 is a simplified logic illustration of how the jumpers and AT Control Register bits function to map the 1MB of main memory to make it accessible to an AT bus controller.

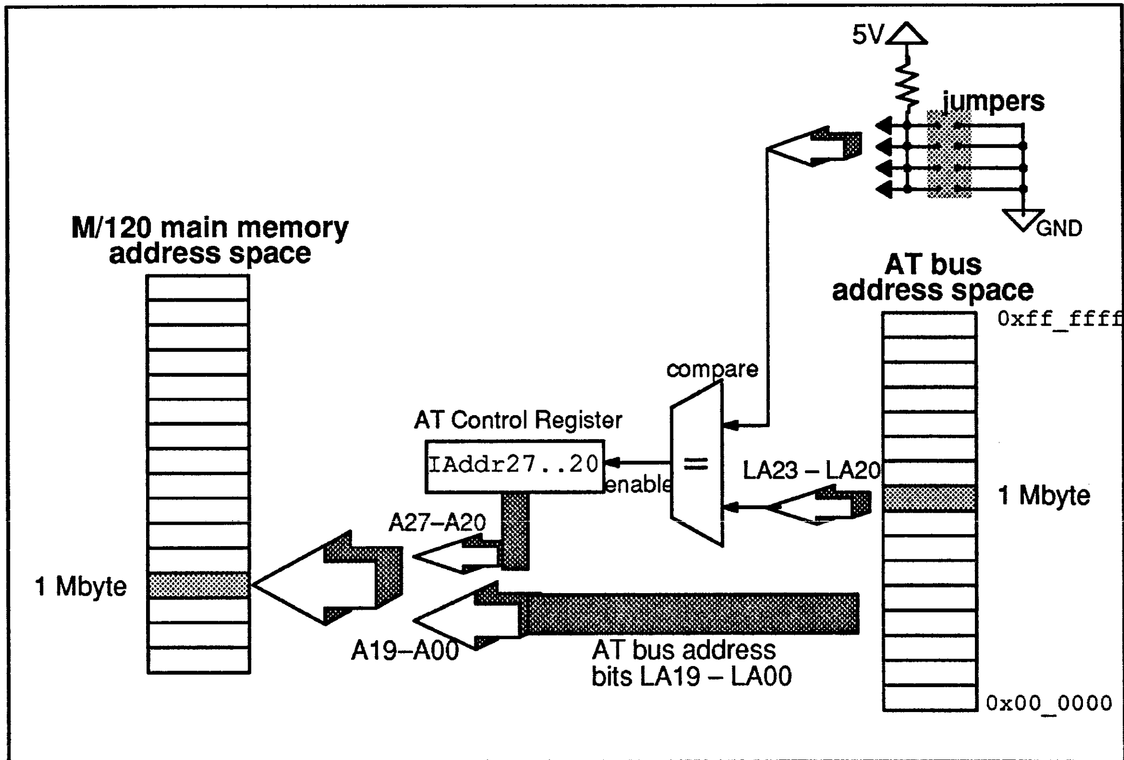



*Figure A.1 AT Bus Memory Mapping Options*

The M/120 is shipped with no jumpers installed for the AT bus address mapping. Thus, since this is equivalent to setting the four most significant AT bus address bits (LA23..LA20) to "1", the 1MB of high address space (0xff_0000 to 0xff_ffff) on the bus is mapped into the M/120 main memory address space. The jumper connections used to specify the mapping is as follows:

| Jumper Location | Pin Numbers | Function |
|---|---|---|
| JPD8: | pin 8 – pin 9 | *absent*: AT Match address LA(23) == 1<br>*present:* AT Match address LA(23) == 0 |
| JPC8B: | pin 8 – pin 9 | *absent:* AT Match address LA(22) == 1<br>*present*: AT Match address LA(22) == 0 |
| JPF9: | pin 7 – pin 10 | *absent:* AT Match address LA(21) == 1<br>*present*: AT Match address LA(21) == 0 |
| JPF9: | pin 8 – pin 9 | *absent:* AT Match address LA(20) == 1<br>*present:* AT Match address LA(20) == 0 |

### AT Bus Memory Refresh

The M/120 provides refresh cycles to the AT bus at approximately 15 microsecond intervals to refresh any dynamic RAM located on the AT bus. A jumper is provided on the motherboard to disable the refresh cycles if a system does not require them. (Disabling refresh may provide a modest gain in performance since bus activity is reduced.) The location of the jumper is listed below. The factory setting has the jumper removed (refresh enabled).

| Jumper Location | Pin Numbers | Function |
|---|---|---|
| JPF9: | pin 6 – pin 11 | absent: AT refresh enabled<br>present:AT refresh disabled |

## AT Bus Device Drivers

MS–DOS or Xenix device driver software must usually be modified or completely rewritten to run on M/120's R2000 processor with RISC/os (UMIPS). Refer to **Chapter 4** for guidelines to writing RISC/os device drivers for the M/120.

## AT Bus Connectors

The M/120 AT bus provides standard 62–pin and 36–pin connectors that accommodate most AT–compatible cards. The M/120's AT bus does not support cards that require connections beyond those provided on the two standard connectors. For example, the system does not support cards such as the Intel IN Board®, which has a cable that is supposed to plug into the Intel 80286 socket, because the M/120 does not have an 80286. The system also can't support cards that rely on access to system resources outside the AT bus, such as BIOS ROMs.

## AT Bus Timing

The M/120's implementation of the AT bus is based closely on the IBM RT PC as described in the *IBM RT PC Hardware Technical Reference Manual : Volume 1, Section 6 , I/O Channel* and the bus timing adheres to the specification in this section. The biggest difference between the standard AT bus implementation and the M/120's implementation is that the M/120 provides only one DMA channel.

## AT Bus DMA Operations

The M/120 system uses a 9516 DMA controller instead of the 8237 device that is standard in PCs. The 9516 provides two DMA channels, and the M/120 dedicates one of the channels to the SCSI interface and the second channel to the AT bus. Thus, the system supports only one DMA channel for the AT bus as compared to seven on a standard AT bus. While it is possible to emulate DMA

using the R2000 (the M/120 CPU), that approach consumes valuable processing cycles. Therefore, boards should be chosen that don't require much bandwidth, or that can be serviced by the one DMA channel. The most complete solution to this limitation is to use boards that operate as *Alternate Controllers* and have their own DMA channels, thus providing higher bandwidth without slowing down the CPU. Refer to the discussion on Alternate Controllers later in this appendix for more details on their operation in the M/120 system.

## DMA Request and Acknowledge Options

Since the M/120 supports only one AT bus DMA channel, the system provides jumper options for selecting which DMA request (DRq0 .. Drq7) signals from the bus are passed on to the 9516. Another set of jumpers map the DMA acknowledge (DAck0 .. DAck7) signals from the 9516 back to the AT bus. Note that the DReq inputs can be globally enabled/disabled by the ATReqEn bit in the *AT Control Register* and the DAck signals can be individually enabled/disabled via four bits in the *AT DAckEn Register*. Refer to **Chapter 3, Programming Model** for a description of these two registers.

Figure A.2 shows the jumper options that are installed when the system is shipped. Note that both DRq1 and DRq2 are jumpered to the DReq input to the 9516. If two boards on the AT bus are using these request inputs, software must ensure that only one board at a time is allowed to assert its DMA request. If a board is not capable of putting its DRq signal in the high–impedance state, then it would be impermissible to have both of the DRq signals jumpered. As shown in Figure A.1, there are five other request inputs (DRq0, DRq3, DRq5 .. DRq7) that can be connected in addition to or instead of the those shown. However, DRq5 should not normally be used since it is already assigned for use by an Alternate Controller for bus requests in the standard configuration of the system.

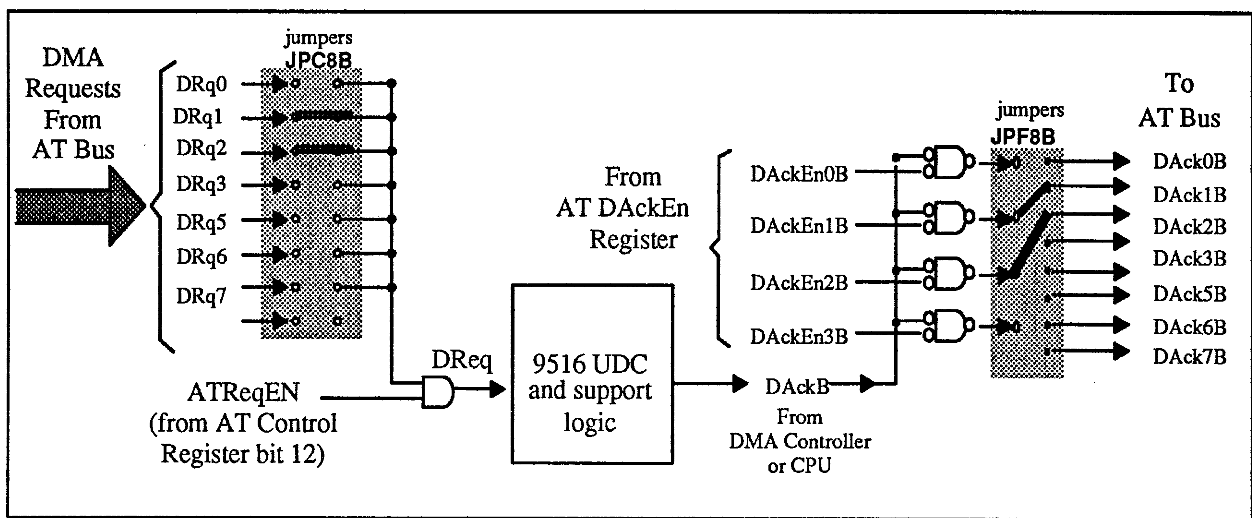



*Figure A.2 AT bus DMA Request/Acknowledge Logic*

When the 9516 is prepared to service a request, it asserts the DAckB signal, which is further enabled by the DAckEn bits. The system is shipped with jumpers connecting the DAckEn1B and DAckEn2B bits to the AT bus DAck1B and DAck2B signals so that the acknowledge signals correspond to the request signals from the AT bus.

## DMA Terminal Count (TC) Signal

Some AT bus devices require assertion of a signal named TC (Terminal Count) during the last bus cycle of a DMA transfer operation. The 9516, however, cannot generate the required TC signal. To allow boards requiring the TC signal to be used in the system, the M/120 provides a combination of interrupt hardware and software–controlled signals that can simulate the expected sequence. Figure A.3 illustrates the logic involved.

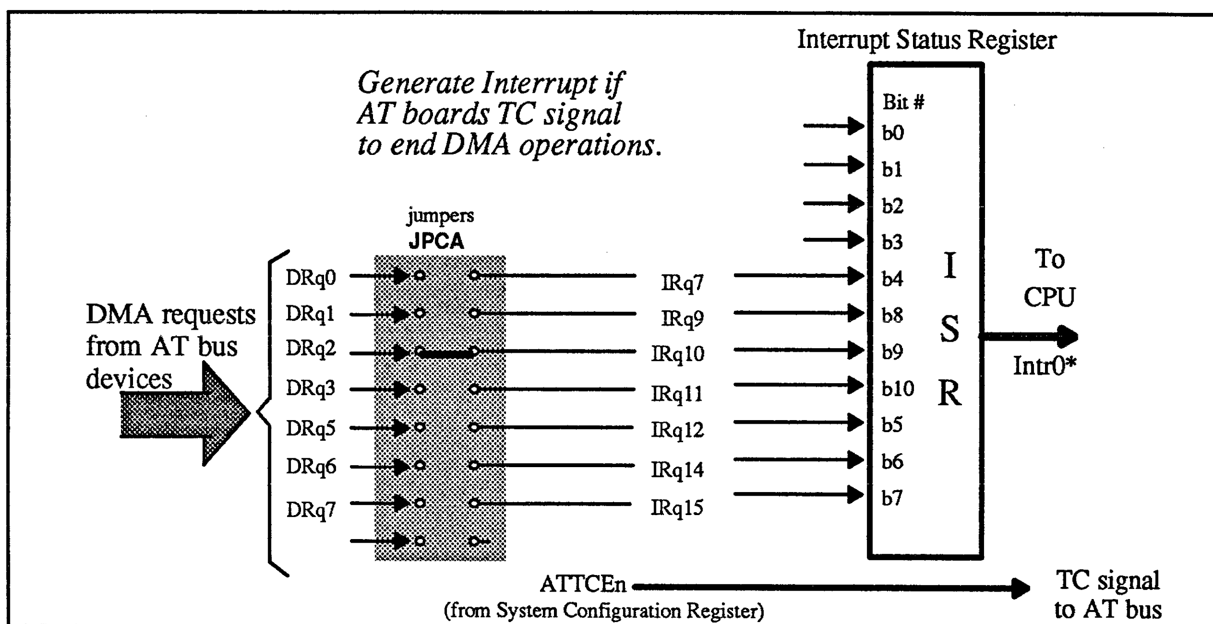



*Figure A.3 Interrupt Logic for DMA TC Signal*

The sequence that must be used to generate the TC signal is:

- The device that requires the TC signal must have its DRq signal jumper–connected to one of the interrupt inputs to the system's *Interrupt Status Register.* The system is shipped with DRq2 jumper–connected to the AT bus interrupt request signal IRq10 as shown in Figure A.3.
- Software must set up the 9516 to transfer all but the last data item (either bytes or half word). That is, if *N* items are to be transferred, then the 9516 must be programmed to handle *N–1* items.
- When the 9516 has transferred all of the specified data *(N–1* items), it interrupts the CPU to tell it that the DMA transfer is complete. Software can then enable IRq10 via the Interrupt Status Register.

- When the DRq2 signal is asserted the next time, the Intr0* signal to the R2000 is asserted and software can determine that this interrupt is requesting that the last DMA cycle (and accompanying TC signal) be performed.
- Software can then enable assertion of the TC signal by setting the ATTCEn bit in the *System Configuration Register* and then have the CPU transfer the last word of data to complete the DMA transfer. After the transfer is completed, software can de–assert the TC signal (reset ATTCEn) and disable the IRq10 interrupt.

# AT Bus Interrupts

The AT bus defines two different kinds of interrupt signals: *shared* interrupts, which can also be described as "pulsed interrupts", and *non–shared* interrupts which are sometimes described as "level interrupts". Shared interrupts are supported by the wired–OR function of the bus and shared interrupt lines are normally high. Non–shared interrupt lines are normally low. In the M/120 system, shared (pulsed) interrupts must use interrupt levels IRq3 through IRq6. The system provides logic on the IRq3 through IRq6 inputs to "catch" pulses (actually rising edges), while the remaining IRq's require sustained levels in order to be seen by the processor. Cards that generate non–shared (level) interrupts can use any of the IRq lines. In order to save IRq3 through IRq6 for those cards which generate "pulsed interrupts", it is advisable that the "level interrupt" cards use IRq's other than 3 through 6.

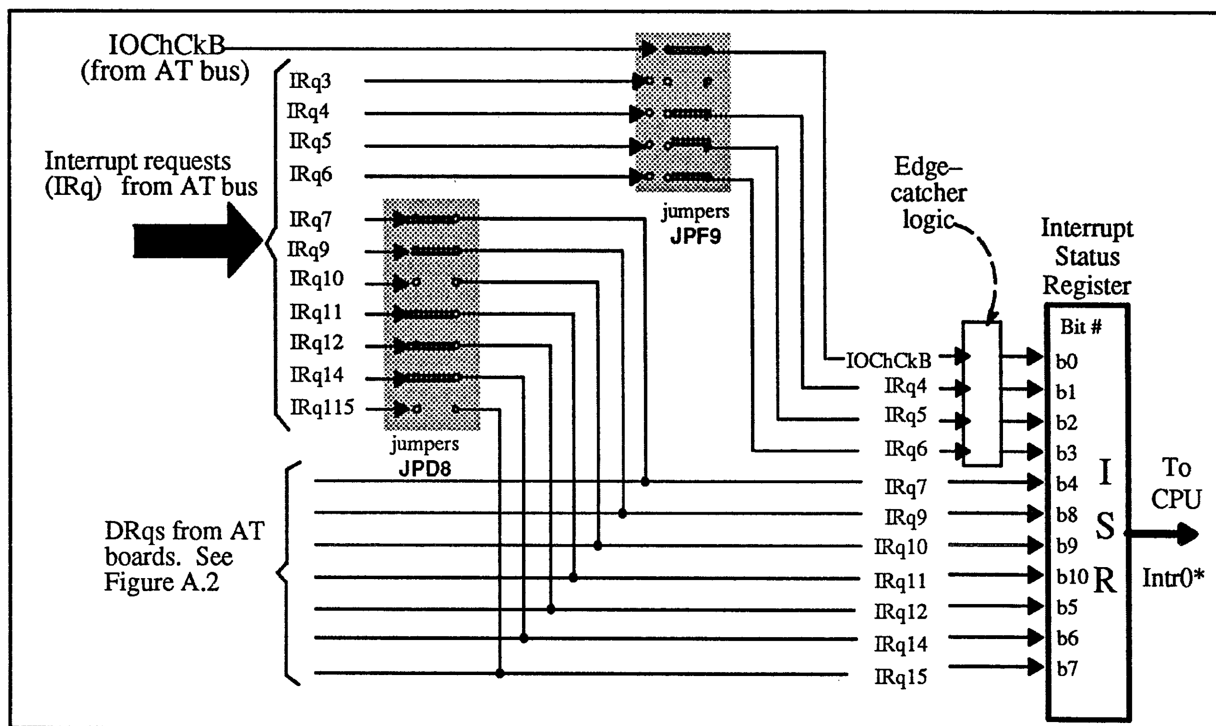



*Figure A.4 AT Bus Interrupt Connections*

Figure A.4 illustrates the AT bus interrupt connections when the M/120 system is shipped from the factory. Note that IRq3 is not connected through the jumper to the Interrupt Status Register. Instead, this interrupt input is connected to the IOChCkB signal from the AT bus. Boards can use the IOChCkB signal to indicate that an uncorrectable system error has occurred.

The IRq10 signal is also not connected in the standard configuration of the system since it is assigned to the DRq2 signal to handle devices that need the TC signal. Refer to the earlier description of the TC signal for details.

# AT Bus Alternate Controllers

*Alternate Controllers* must only perform 16–bit transfers; 8–bit transfers are not allowed. An alternate controller is defined as a card that has its own DMA controller on board, and is designed to take over mastership of the AT bus when it wants to move data to and from memory.

## Bus Access Control for Alternate Controllers

AT bus alternate controllers are boards that provide their own DMA capability and therefore do not use the M/120 system's 9516 controller. When these alternate controllers need access to the system's buses to move data to and from main memory, they can use the same DRq input signals that other boards use to request service from the 9516. In this case, however, the requests are routed to the system's bus arbitration logic as shown in Figure A.5.

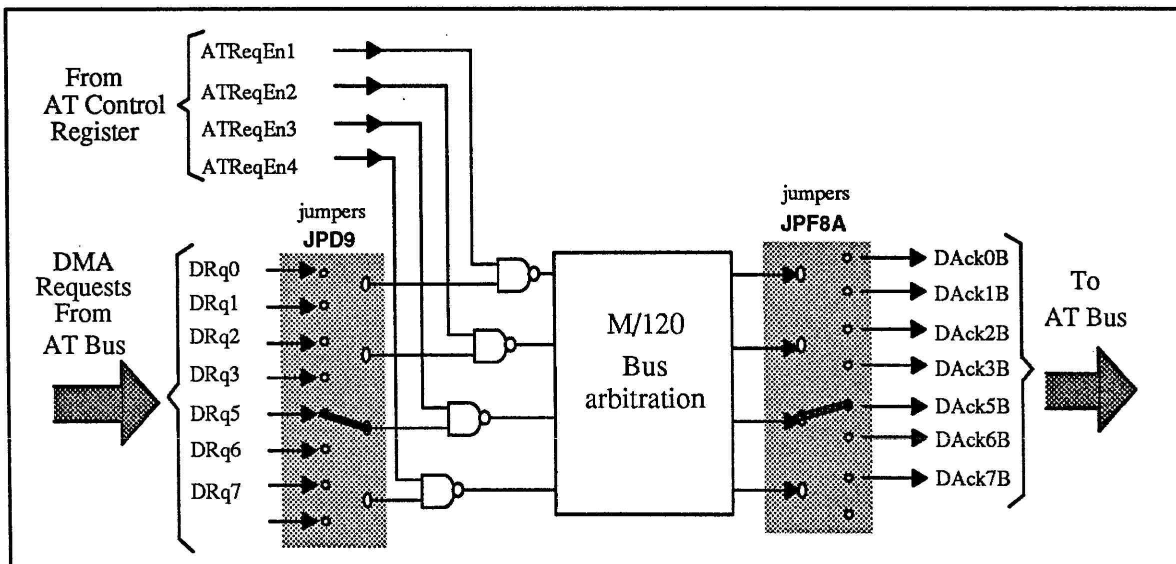



*Figure A.5 Alternate Controllers Bus Access Logic*

The bus access requests from the alternate controllers are routed through a jumper block and can then be enabled/disabled via the ATReqEn bits in the *AT Control Register* (described in

**Chapter 3, Programming Model**). The system is shipped with DRq5 from the AT bus jumpered through to the bus arbitration logic and enabled by the ATReqEn3 bit. When the system is ready to grant bus access to the alternate controller, it asserts a DMA acknowledge signal (DAck5B in the shipped configuration). The assignment of DRq5 and DAck5B to alternate controller bus access requests is rather arbitrary — it was chosen in this configuration so that it would not conflict with the other assignments of request/acknowledge signals existing in the system. The system provides request/acknowledge paths for up to four alternate controllers.

Alternate controllers are capable of holding the AT bus and the I–Bus for extended periods. Extended bus occupancy can interfere with other uses of these buses. In particular, the Ethernet controller needs frequent access to the M/120;'s M–Bus. Additionally, if devices on the AT bus need refresh, the alternate controller must not stay on the bus so long that refresh cycles are missed. Therefore, you must ensure that alternate controllers relinquish control of the buses to accomodate Ethernet activities and refresh operations.

## AT Bus Option and Jumper Summary

Figure A.6 summarizes the factory settings for the option jumpers. Figure A.7 is a logical summary of the connections that determine the configuration of the AT bus DMA and interrupt signals. Refer to the descriptions earlier in this chapter for details.

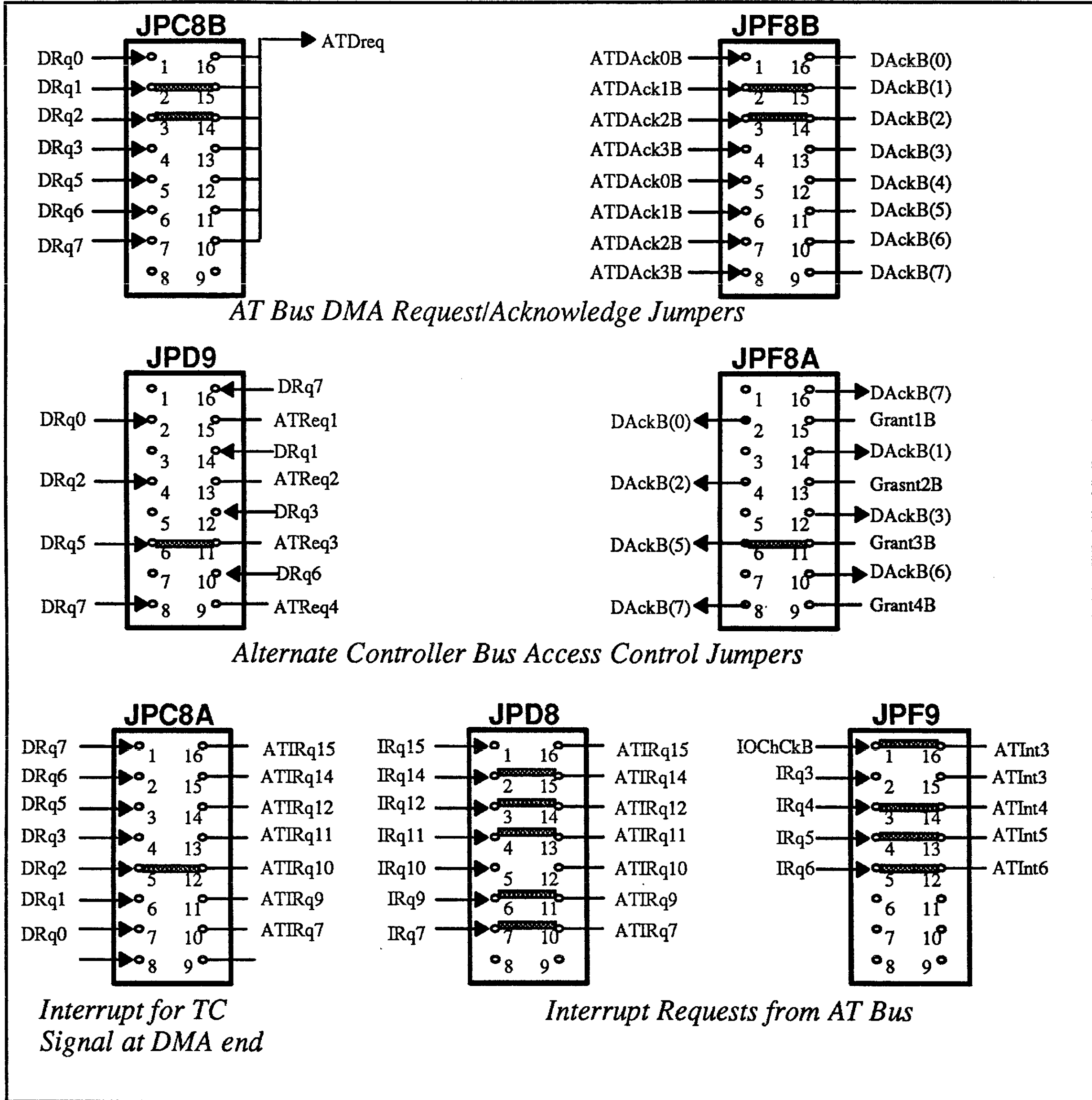



*Figure A.6 Default Jumper Settings*

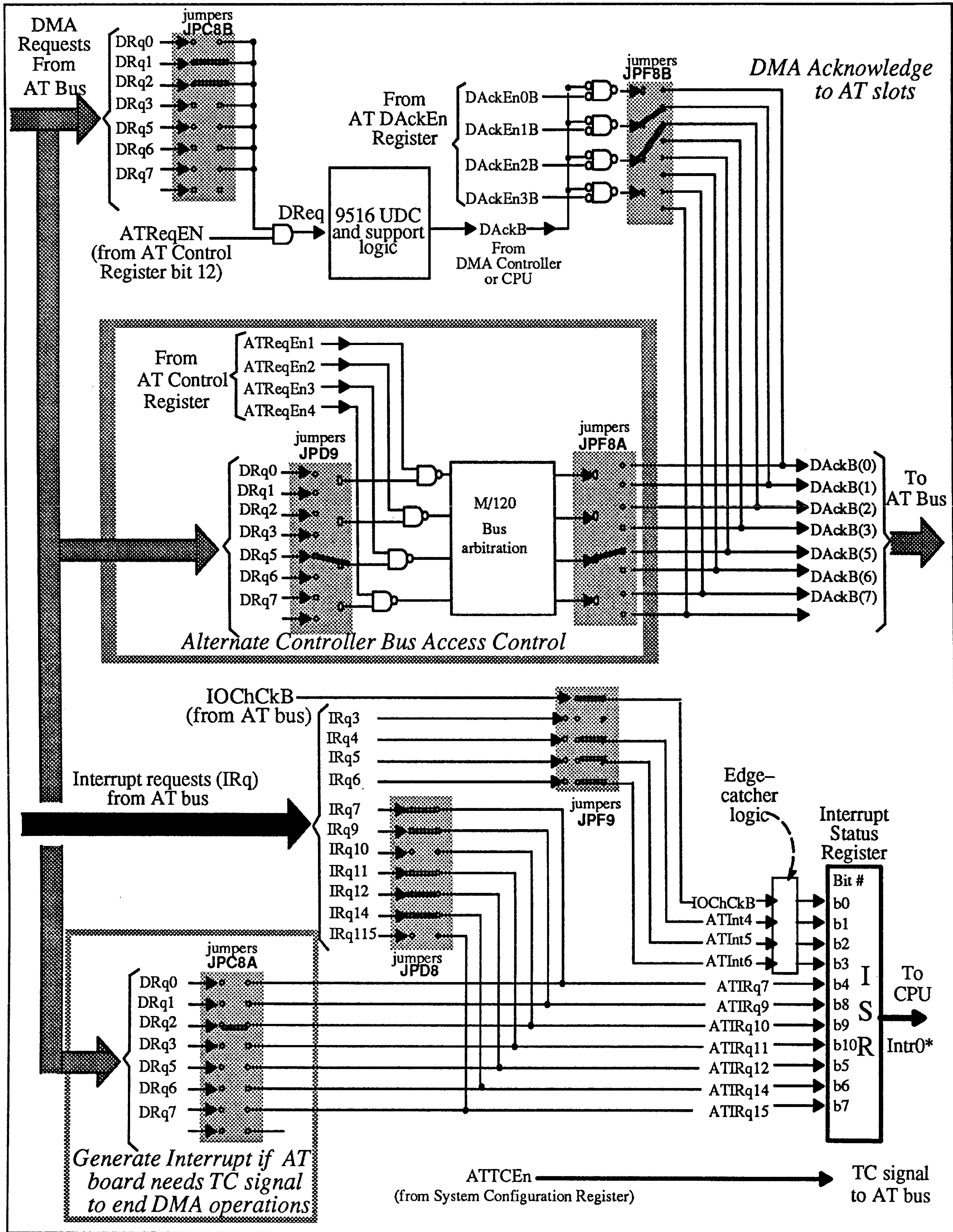



*Figure A.7. Summary of AT–bus Jumper–Selectable Options*

# AT Bus Pin Assignments

Figure A..8 lists the signal assignments for the M/120's 62–pin and 36–pin AT bus connectors

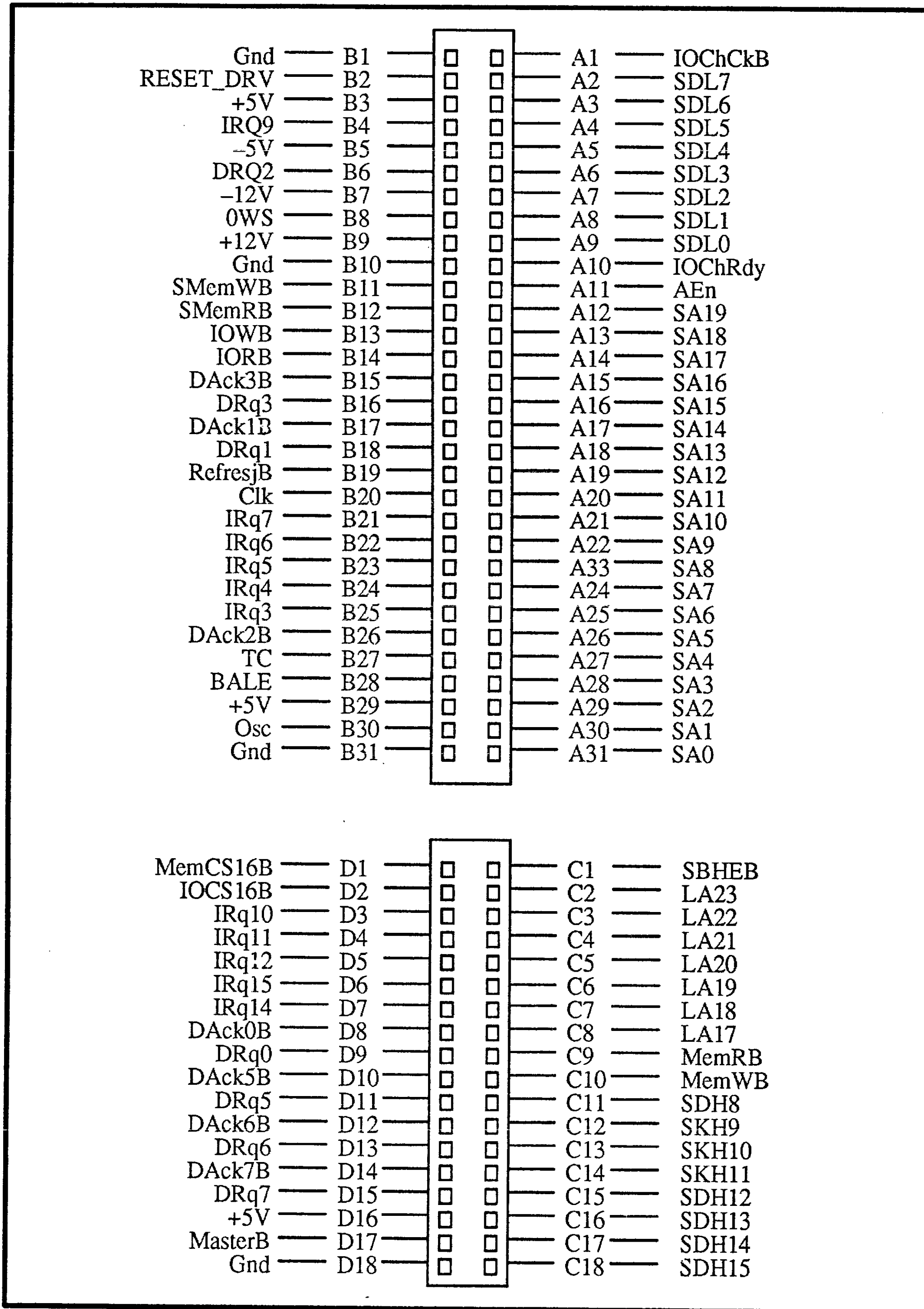



*Figure A.8 AT Bus Connector Pin/Signal Assignments*

# Appendix B
# Tape Drive Operation and Maintenance

## Operation

The Tape Drive operation instructions and information include the following items:

- Tape Cartridge
- Cartridge Loading and Ejecting
- Write Protection

### Tape Cartridge

The Tape Drive records on industry–standard tape cartridges, which have been qualified for operation at either 8,000 bpi or 10,000 bpi. The following tape cartridges can be used.

- 3M DC600A (600 ft.) or equivalent
- 3M DC600XTD (600 ft.) or equivalent

### Cartridge Loading and Ejecting

The Tape Drive is designed so that the tape cartridge can be loaded in only one way. The cartridge is loaded by inserting it into the Tape Drive and pushing the tape inward until it reaches a hard stop. As the cartridge is inserted, a slight resistance from the ejector assembly is encountered. This resistance cushions the loading action. Just before reaching the stop point, the cartridge protective door is opened, which exposes the tape. The stop point is reached when the cartridge metal base drops behind the lip of the front panel. Move the head loading lever towards the cartridge as far as it will move. The head loading lever secures the cartridge and moves the head assembly into operating position.

## Write Protection

The tape cartridge is equipped with a write–protect plug, which can be rotated by the user before cartridge insertion to either the Safe or Unsafe position. Refer to Figure B.1. The Safe position only allows the tape to be read, and writing is inhibited. To write to a tape, the write–protect plug must be in the Unsafe position.

*Figure B.1. Cartridge Write Protect Plug*

# Maintenance

This section contains the cleaning procedure for the M/120 Tape Drive. The procedure includes cleaning the heads and sensor holes. A small amount of dust can affect the data and inhibit performance. Therefore, it is recommended that a cleaning schedule is established.

## Recommended Cleaning Schedule

The read, write, and erase heads should be cleaned:

- After each initial pass with a new tape cartridge
- After each 8 hour period of read, write, or erase activity.

The sensor opening and tape cartridge cavity should be cleaned:

- Whenever dust or debris is visible inside the cartridge cavity.

## Cleaning Supplies

The following supplies are needed for cleaning the Tape Drive. These supplies are available from Archive Corporation, 1650 Sunflower Avenue, Costa Mesa, California, 92626 (714)641–0279. If an Archive streamer head cleaner cassette is not available, then the heads may be cleaned using Archive head cleaning fluid or Freon–TF and 6–inch long swabs, made from lintless cotton or any other industry acceptable head–cleaning swabs.

Required Items:

- An Archive streamer head cleaner kit (Archive part number: 14916–001)
- Low pressure air in aerosol can

Optional Items:

- Archive streaming head cleaning fluid (Archive part number: 14917–001)
- Archive head cleaning pads (Archive part number: (14918–001)
- Head–cleaning swabs

## Cleaning Procedure

1. Turn off the computer system and disconnect the power cord from the power source.

2. Engage the tape head assembly by sliding the tape head lever to the right. Refer to Figure B.2.

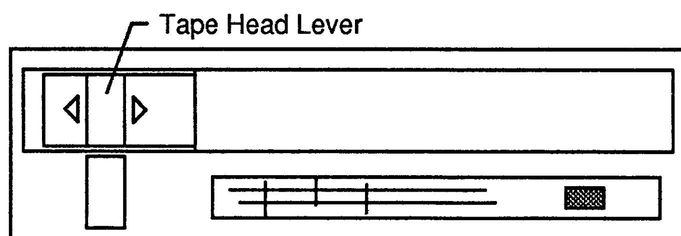



*Figure B.2. Front View of Tape Drive*

3. Visually inspect the interior of the Tape Drive. If contamination is visible in the sensor holes or within the cartridge cavity of the drive, then carefully blow out the visible dust using low pressure air from an aerosol can. The location of the sensor holes is shown in Figure B.3 on the following page.

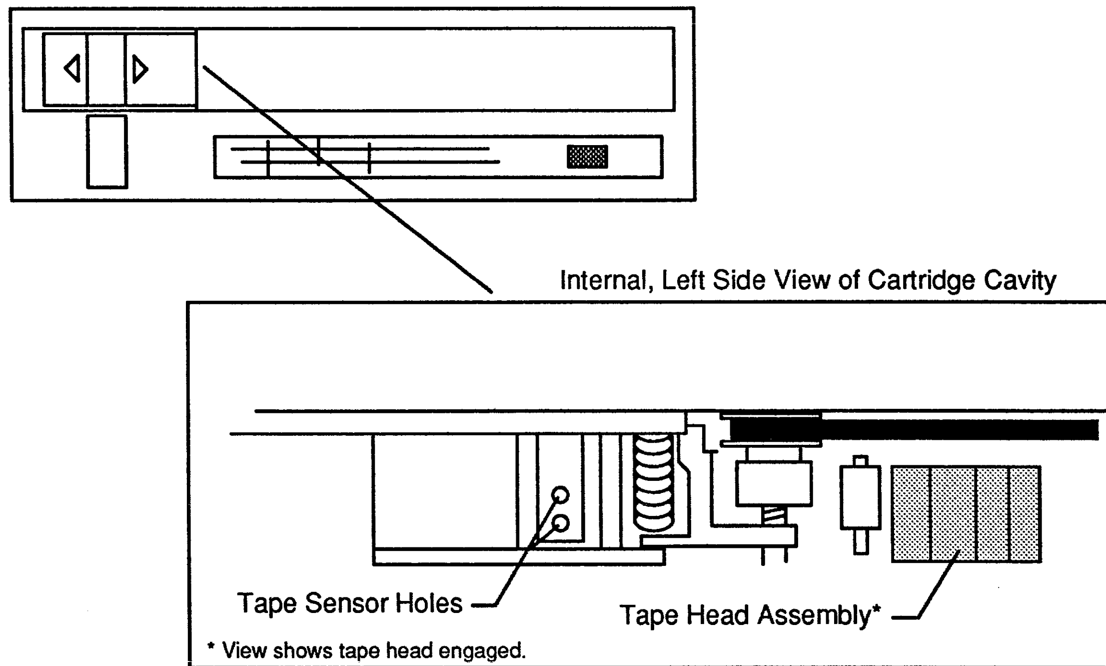



*Figure B.3. Location of the Sensor Holes and Tape Head Assembly*

2. Follow the instructions provided in the streamer head cleaner kit, and clean the head assembly. Figure B.3 shows the location of the tape head assembly. If you do not have the head cleaning kit, then skip to step 3.

3. Perform the following steps to clean the tape head assembly if a head cleaning kit is not available. This procedure requires head cleaning fluid and swabs.

   a. Moisten the swab with the head cleaning solution until it is saturated, but not dripping. Alcohol should not be used to clean the tape head assembly.

   b. Carefully wipe the swab across the tape head assembly in the directions that the tape moves. Do not wipe the head using an up and down motion as residue may collect in the minute crevices of the head. Do not use a scrubbing or circular motion.

   c. Discard the first swab. Moisten a second swab and repeat the wiping motion described in step b. Continue wiping until all residue has been removed from the head surface. This may require you to use a third swab.

   d. Wipe the head assembly with a clean, dry swab using the same motion as described in step b until the head assembly is dry.

   e. Move the tape head lever away from the cartridge insertion opening to move the head assembly out of the cartridge area.

   f. Reconnect the power cord, and power up the computer system.

# Appendix C
# Installing Disk Drives in the Expansion Cabinet

This chapter contains the instructions for installing additional Disk Drives in the Expansion Cabinet.

Up to five additional disk drives can be installed in an expansion cabinet. Both 156 megabyte and 328 megabyte disk drives can be added. This section provides the rules and installation procedures for adding one or more disk drives.

The Small Computer Systems Interface (SCSI) controller can support up to 8 devices (0 – 7). The SCSI devices are assigned as given in Table C.1.

**Table C.1. SCSI Device Assignments**

| Device # | Peripheral Device | Device Priority |
|---|---|---|
| 7 | Initiator, located on Motherboard | Highest |
| 6 | Tape Drive, Main Cabinet | |
| 5 | Disk in slot 5, Expansion Cabinet | |
| 4 | Disk in slot 4, Expansion Cabinet | |
| 3 | Disk in slot 3, Expansion Cabinet | |
| 2 | Disk in slot 2, Expansion Cabinet | |
| 1 | Disk in slot 1, Expansion Cabinet | |
| 0 | Disk Drive, Main Cabinet | Lowest |

1. Set the power switch in the off position, and disconnect any power cords.
2. Remove the three screws along the right edge of the rear panel of the Expansion Cabinet. Refer to Figure C.1.
3. Slide the side panel towards the back of the Expansion Cabinet about one inch. Then, lift the side cover from the Cabinet. The latching tabs located on the top and bottom, front edges of the side panel must clear the other half of the latching assemblies before the side panel can be removed. Refer to Figure C.2.

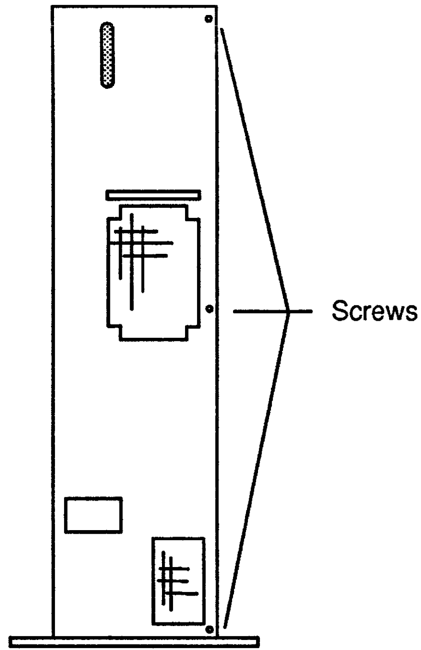



*Figure C.1. Expansion Cabinet Side Panel Screws*

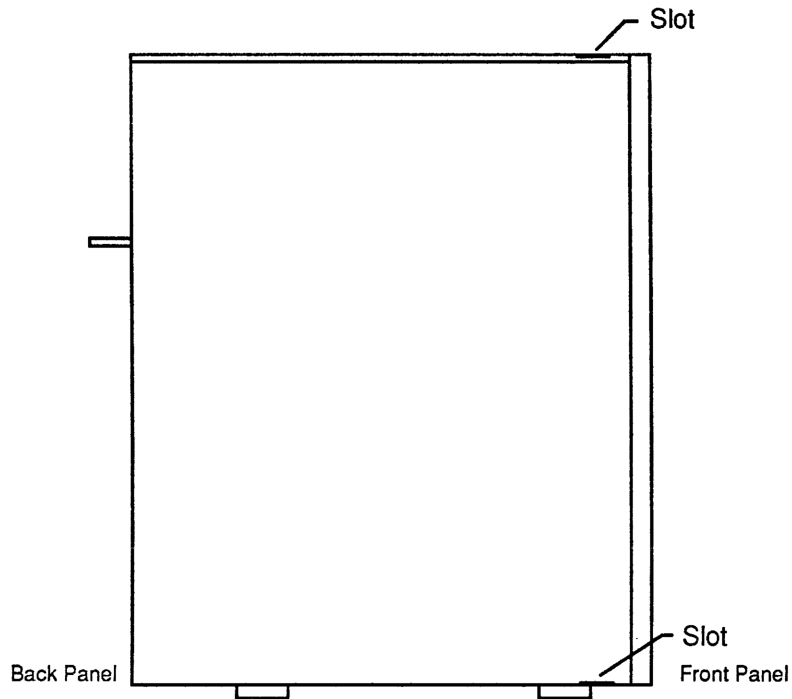



*Figure C.2. Location of Latching Tab Slots*

4. Specify the ID number using the the jumper block and shorting plug(s) on the rear panel of the Disk Drive. The ID number can range from 1 to 5 and is the same number as the slot where the Disk Drive will be installed. Refer to Figure C.3 for the slot assignments.

   The Disk Drives are mounted from the bottom of the cabinet up, to keep the center of gravity as low as possible. The priority and cabling scheme also requires that additional disks be installed from the bottom up.

   The jumper blocks for the 156 MB and 328 MB disk drives are located in different positions on the rear panel. The significant pin sets also differ. Figure C.4 on the following page shows the rear panels for both the 156 MB and 328 MB Disk Drives, and the significant pin sets on each jumper block.

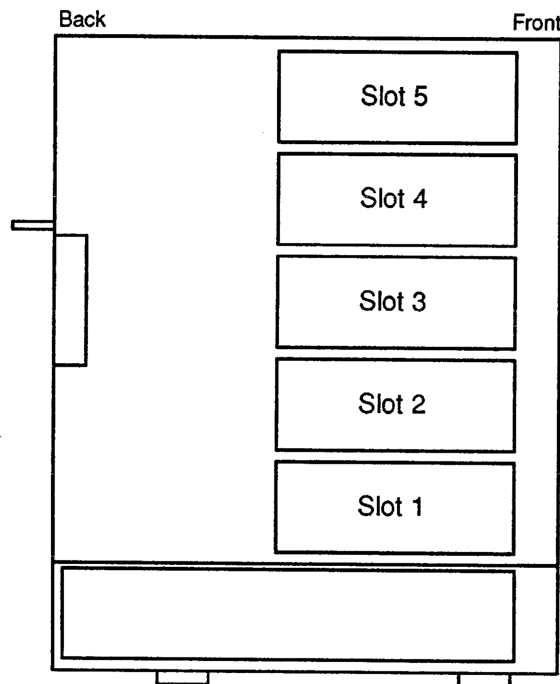



*Figure C.3. Expansion Cabinet Slot Assignments*

## Note

The jumper blocks illustrated in Figure C.4, show the significant pins for specifying the ID number. The other pin sets that are shown with shorting plugs enable parity checking. For the 156 MB disk drive, pin set 1 enables termination power source and pin 3 is jumpered to enable parity. For the 328 MB Disk Drive, pin set 5 is jumpered to enable parity.

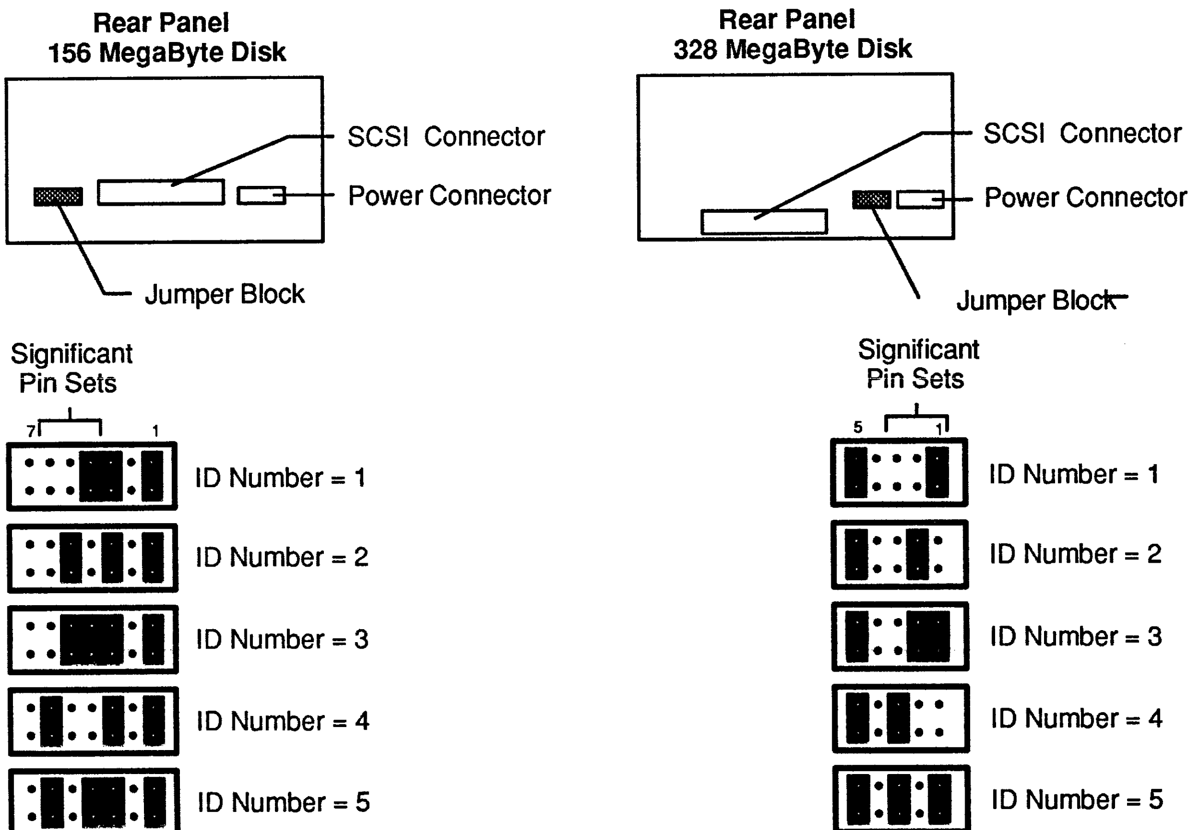



*Figure C.4. Specifying the ID Number*

5. Complete this step if the Disk Drive is being installed in slot 1. Refer to Figure C.3 for the slot assignments. Otherwise, skip this step and proceed with step 6.

   If the Disk Drive is to be installed in slot 1, then the Disk Drive must be terminated. All other Disk Drives must not be terminated. The procedure for terminating a 328 MB Disk Drive differs from terminating a 156 MB Disk Drive. Use the section of information that applies to the Disk Drive that you are installing.

5. Continued

**328 MB Disk Drive.** The 328 megabyte Disk Drive is terminated by plugging two terminating resistor packs into the sockets on the rear panel of the Disk Drive. Refer to Figure C.5.

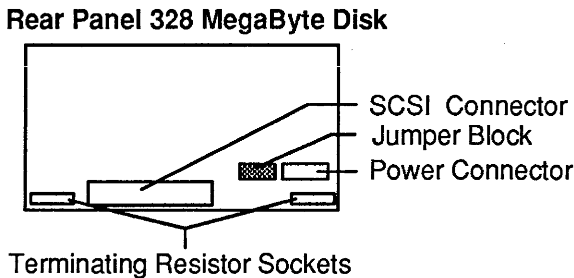



*Figure C.5. Installing the Terminating Resistor (328 MB Disk Drive)*

**156 MB Disk Drive.** To terminate the 156 MB Disk Drive, the drive must be opened up. Proceed as follows to terminate the 156MB Disk Drive.

a. Remove the two torx–head screws from the side panels that are near the back panel as shown in Figure C.6.

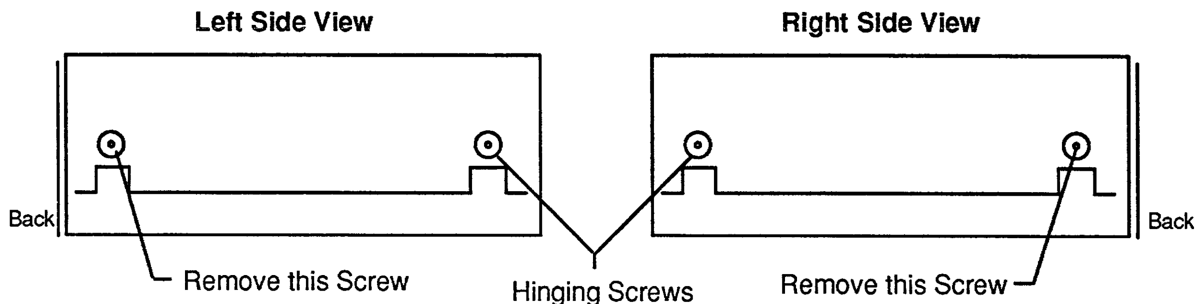



*Figure C.6. Opening the Disk Drive*

b. Open the Disk Drive by pulling the top and bottom edges apart towards the back of the Disk Drive. The Disk Drive hinges on the two torx–head screws towards the front of the Disk Drive as the Drive is pulled apart at the back. The hinging screws are also shown in Figure C.6.

5. Continued

c. Install the three 8–pin Terminating Resistors in the sockets shown in Figure C.7. Make sure that pin one of the Terminating Resistor is placed in the correct pin hole on the socket.

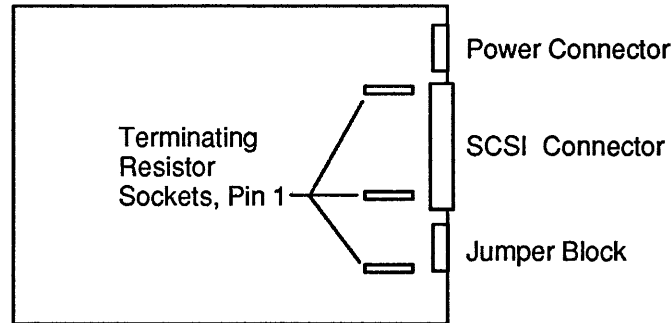



*Figure C.7. Installing the Terminating Resistors (156 MB Disk Drive)*

d. Carefully close the Disk Drive, and reinstall the two torx–head screws.

e. Reconnect the Motor Control Cable to the Motor Control Cable Connector. This cable was removed from the connector when the Disk Drive was opened. See Figure C.8.

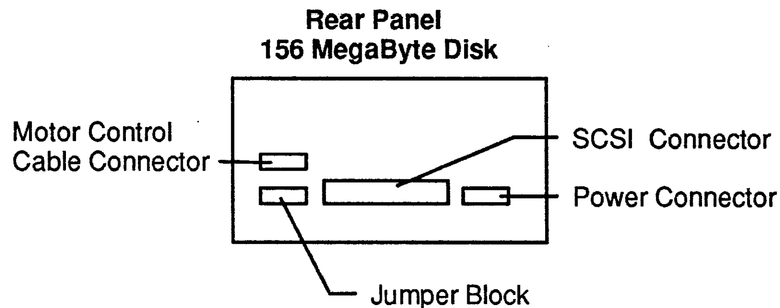



*Figure C.8. Reconnecting the Motor Control Cable*

6. Install the Disk Mounting Bracket on the Disk Drive using the following information. Depending on the slot that the Disk Drive will be installed in, the bracket is installed differently.

   When looking at the rear panel of the Disk Drive, the Disk Mounting Bracket is installed with the lip on the right side of the Disk Drive. For Disk Drives installed in slots 1 through 4, the mounting bracket is installed on the Disk Drive with the lip pointing up or towards the Disk Drive. Refer to Figure C.9. For slot 5 (top position) the Mounting Bracket is installed with the lip pointing down or away from the Disk Drive.

   Install the Disk Mounting Bracket on the bottom of the Disk Drive using Figure C.9 and the four screws that were provided.

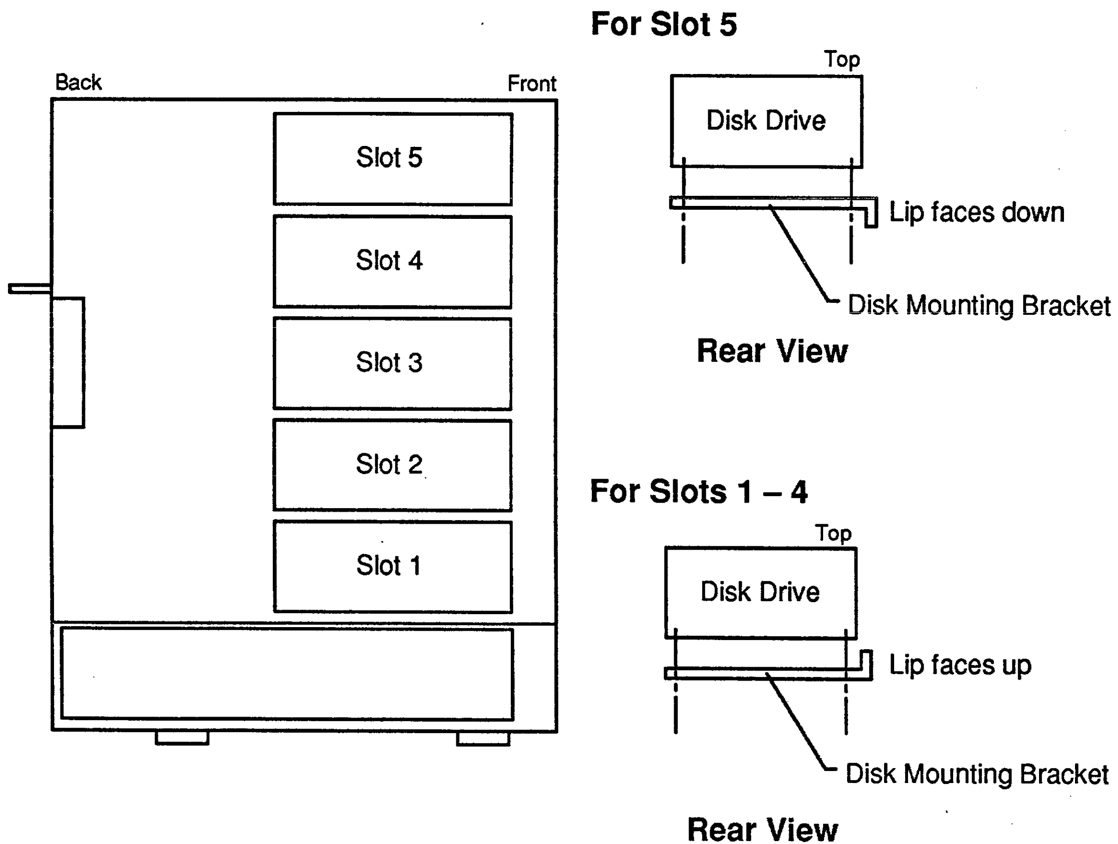



*Figure C.9. Installing the Disk Mounting Bracket*

7. Slide the Disk Drive partially into the mounting slot in the Expansion Cabinet. Before sliding the Disk Drive completely into the mounting slot, connect the LED cable to the front panel of the Disk Drive. The LED cable has a 2–pin connector that is keyed. Figure C.10 shows the location of the LED cable connector on the front panel of the Disk Drive.

   Slide the Disk Drive into the mounting slot until the Mounting Bracket is flush with the Disk Drive Mounting Rack. If the Mounting Bracket installed in step 4 was mounted on the bottom of the Disk Drive correctly, then the front of the Disk Drive should be towards the front of the Expansion Cabinet.

   If the front of the Disk Drive is not towards the front, then remove the Mounting Bracket from the bottom of the Disk Drive and remount the Bracket with the lip on the other side of the Disk Drive. Looking at the Disk Drive from the back, the lip of the Bracket should be on the right side of the Disk Drive.

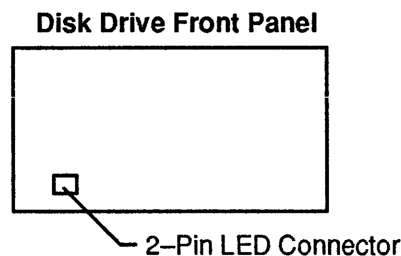



*Figure C.10. Location of LED Cable Connector*

8. Install the two screws on the Disk Mounting Bracket to secure the Disk Drive.
9. Repeat steps 1 through 8 for each additional Disk Drive that you are installing in the Expansion Cabinet. When all additional Disk Drives have been installed in the Expansion Cabinet, then proceed to step 10.
10. Connect the SCSI Cable coming from the rear panel of the Expansion Cabinet to each Disk Drive. The SCSI Cable has multiple, key–coded connectors.
11. Reinstall the side panel by aligning the latching tab on the side panel with the slot at the top of the computer chassis. Refer to Figure C.4. Press the latching tabs into the slots, and slide the side panel towards the front of the machine.
12. Reinstall the three screws on the rear panel that secure the side panel.
13. Remove the SCSI Bus Terminator from the SCSI connector on the rear panel of the computer. Keep the terminator in a safe place. If in the future you want to disconnect the Expansion Cabinet, then the SCSI Bus Terminator must be reinstalled on the SCSI connector. Refer to Figure C.11.

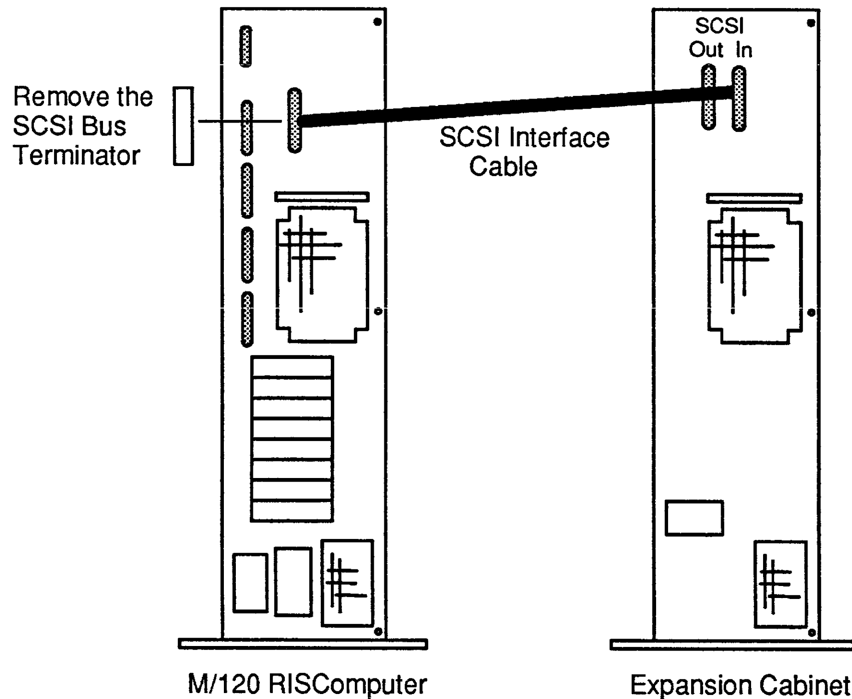



*Figure C.11. Installing the SCSI Interface Cable*

14. Connect the SCSI interface cable to the SCSI connector on the rear panel of the computer and to the SCSI IN connector on the rear panel of the Expansion Cabinet and lock the bail clips. Refer to Figure C.11.

15. Verify that the Power Supply is set to the same voltage in the Expansion Cabinet as it is in the computer. The voltage is preset at the factory according to the purchase order. The voltage is specified on the rear panel of the computer and the Expansion Cabinet.

    If required, refer to **Chapter 2** and the **Select the Voltage Setting** section for instructions on changing the voltage setting.

# Appendix D
# Power On Diagnostics

## Introduction

The Power On (PON) Diagnostics are a group of tests that check the integrity of the MIPS Computer Systems. Currently, the Power On Diagnostic package is used for the M/120, the M/1000, and the M/2000. This appendix only includes information on the diagnostics used for the M/120 RISComputer System.

The PON diagnostics were designed as *go/no–go* tests. The tests begin with the lowest level of IC devices and system registers an move upwards to higher levels of system functionality. If a system specific test is encountered for a different system, then the test is skipped.

The PON diagnostics are not run when a keyswitch reset is performed. When the Power ON button is set in the ON position, then the following sequence of events occurs.

1. The Central Processing Unit (CPU) Cause and Status registers are initialized.
2. The bootmode variable is checked. If the Bootmode variable is NOT set to *d*, then the PON routines are called. If the Bootmode is set to *d*, then the PON diagnostics are not executed, and memory is not cleared.
3. The PON diagnostics are executed. The following diagnostics are performed sequentially by column from top to bottom, and left to right.

| | | |
|---|---|---|
| Pon_Leds | Pon_Scr | Pon_UdcSlave |
| Pon_Duart | Pon_VM | Pon_Chain1 |
| Pon_Banner | Pon_Allexc | Pon_Chain2 |
| Pon_Cache1 | Pon_Parity | Pon_ScsiSlave |
| Pon_Cache 2 | Pon_NVram | Pon_ScsiMaster |
| Pon_Cache 3 | Pon_Timers* | Pon_EnetProm |
| Pon_Cache 4 | Pon_Duarts* | Pon_LanceSlave |
| Pon_IdProm | Pon_Imr | Pon_LanceMaster |
| Pon_WB | Pon_Fp1 | Pon_Atreg |
| Pon_Memory | Pon_Fp2 | |

* These tests can return more than one error code.

When a test is started, a message banner is displayed on the screen that indicates which test is running. Each test has an identifying LED pattern that is displayed on the two LED blocks when the test is started. The LED blocks are located on the Motherboard and are shown in Figure D.1. These LEDs can be seen if the side panel of the computer has been removed and can also be observed by looking through the perforated rear panel above the fan. All identifying LED patterns have at least two LEDs turned on, which distinguishes the LED pattern from the single LED walking pattern that is set upon the successful entry into the PON Diagnostics. The single LED walking pattern is also displayed upon the completion of the PON Diagnostics and entry into the PROM Monitor.

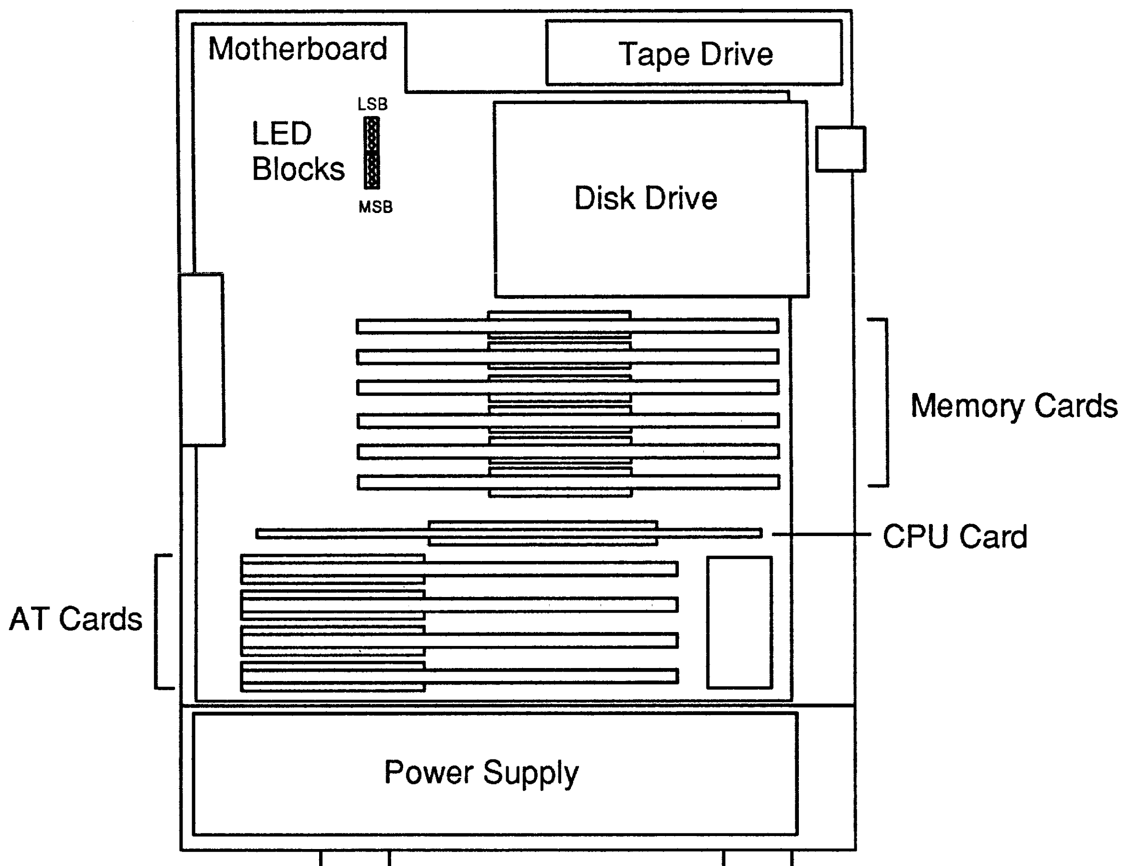



*Figure D.1. Location of LED Blocks*

The diagnostic software keeps track of the sections of the system that failed. This information is stored in four bytes of the non–volatile RAM (NVRAM). If a test is dependent on a previous test passing, then the bytes in non–volatile RAM are checked.

If any of the previous dependent tests failed, then the current test sets its dependency mark and sends a "SKIPPED" message to the console. When a test is skipped because of a previous dependent test failing, then there is no flashing LED code. This type of return is called a "norun" because the test was skipped and not run. A single test failure can cause a ripple effect when other tests appear to fail because of "noruns."

If all previous dependent tests have passed, then the test is executed. If the test passes, then a "PASSED" message is displayed on the console after the test name.

If an error is found while a test is executing and the test fails, then the following sequence of events occurs.

- A "FAILED" message is displayed on the screen.
- The LED pattern for that test flashes on and off for approximately seven seconds.
- The dependency mark is set in NVRAM if applicable.
- The bootmode variable is set to *e* by the PON diagnostic program.

Table D.1 provides a brief description of each test and lists the LED code. Table D.2 specifies the test dependencies, the name of the bit in NVRAM that is set when a test fails, and the LED code for each test.

**Table D.1. PON Diagnostic Descriptions**

| Test | Description |
|---|---|
| Pon_Leds | Walking ones pattern |
| Pon_Duart | Internal loopback of console port |
| Pon_Banner | Displays the PON headers |
| Pon_Cache1 | Verifies cache mapping of the R2000 |
| Pon_Cache 2 | Verifies instruction cache timing |
| Pon_Cache 3 | Checks the Data Cache |
| Pon_Cache 4 | Checks the Instruction Cache |
| Pon_IdProm | Verifies ID PROM checksum |
| Pon_WB | Verifies byte, half–word, tri–byte, and word write operations |
| Pon_Memory | Knaizuk Hartman algorithm main memory test |
| Pon_Scr | Write/read test of the System Configuration Register |
| Pon_VM | Checks the R2000 Translation Lookaside Buffer operation |
| Pon_Allexc | Checks the R2000 exceptions handling |
| Pon_Parity | Checks parity error detection and fault registers |
| Pon_NVRAM | Non–destructive write/read of NVRAM |
| Pon_Timers | Checks the 8254 and time–of–day clocks |
| Pon_Duarts | Internal loopback of the remaining ports other than the console using various baud rates |
| Pon_Imr | Write/read test of the interrupt mask register |
| Pon_Fp1 | Verifies R2010 operations and exceptions |
| Pon_Fp2 | Verifies R2010 operations and exceptions |
| Pon_UdcSlave | Write/read test of the UDC registers |
| Pon_Chain1 | UDC chaining operation on channel 1 |
| Pon_Chain2 | UDC chaining operation on channel 2 |
| Pon_ScsiSlave | Write/read test of the SPC registers |
| Pon_ScsiMaster | Check for all SCSI devices and performs a disk write/read when possible |
| Pon_EnetProm | Verifies the Ethernet PROM checksum |
| Pon_LanceSlave | Write/read test of the Lance registers |
| Pon_LanceMaster | Internal loopback test |
| Pon_Atreg | Write/read test of the PCAT register |

### Table D.2. PON Troubleshooting Information

| Test | Dependency | NVRAM Bit | LED Code |
|---|---|---|---|
| Pon_Leds | none | – | none |
| Pon_Duart | none | console | 0x03 |
| Pon_Banner | none | – | none |
| Pon_Cache1 | cache | cache | 0x05 |
| Pon_Cache 2 | cache | cache | 0x06 |
| Pon_Cache 3 | cache | cache | 0x07 |
| Pon_Cache 4 | cache | cache | 0x09 |
| Pon_IdProm | none | IDPROM | 0x0A |
| Pon_WB | none | WB | 0x0B |
| Pon_Memory | none | memory | 0x0C |
| Pon_Scr | none | SCR | 0x41 |
| Pon_VM | cache, memory | TLB | 0x0D |
| Pon_Allexc | cache, memory, TLB | except | 0x0E |
| Pon_Parity | memory | fault | 0x42 |
| Pon_NVRAM | none | NVRAM | 0x0F |
| Pon_Timers | none | timer | 0x11 |
| | | TOD | 0x12 |
| Pon_Duarts | none | duarts (port 2) | 0x13 |
| | | duarts (port 3) | 0x14 |
| | | duarts (port 4) | 0x15 |
| Pon_Imr | none | IMR | 0x43 |
| Pon_Fp1 | FP | FP | 0x16 |
| Pon_Fp2 | FP | FP | 0x17 |
| Pon_UdcSlave | none | UDC | 0x44 |
| Pon_Chain1 | memory, SCR, UDC | UDC | 0x45 |
| Pon_Chain2 | memory, SCR, UDC | UDC | 0x46 |
| Pon_ScsiSlave | none | SPC | 0x47 |
| Pon_ScsiMaster | IMR, memory, SCR, SPC, UDC | SPC | 0x48 |
| Pon_EnetProm | none | ENETPROM | 0x49 |
| Pon_LanceSlave | none | Lance | 0x4A |
| Pon_LanceMaster | IMR, Lance, memory | Lance | 0x4B |
| Pon_Atreg | none | AT | 0x4C |

The remainder of this appendix includes a section for each Power On diagnostic routine. Each section describes one test and lists the messages, if any, that might be displayed.

## Pon_Leds

This test displays a walking ones pattern on the LED block on the Motherboard, and verifies that each LED can be turned on independently.

This test does not print a message on the console.

## Pon_Duart

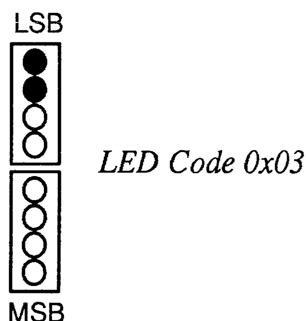
The console port is configured to transmit and receive characters with the following characteristics.

- No parity
- 8 bits per character
- 1 stop bit
- 9600 baud
- local loopback mode (Feature of the SCN 2681 where the data received is the data transmitted.)

This test uses polled reads of the DUART's status register for transmit and receive operations. This test transmits twenty different data patterns and checks the data received after each transmit. The test prints a fail message on the screen if the data received and the data transmitted is not the same. A fail message is also printed on the screen if a timeout occurs while waiting for the transmitter or the receiver to become ready.

Once the test is complete, the console port is re–programmed for normal operation.

This test attempts to print a fail message. However, if the console is not connected or has failed, then the message cannot be displayed. If the DUART is not working, then the LED pattern shown in the following illustration flashes on the LED blocks on the Motherboard for approximately seven seconds. The program then starts the next test.




*LED Code 0x03*

## Pon_Banner

This routine displays the Power On Diagnostics header shown below. This routine indicates that the PON diagnostics are executing.

**Running Power–On Diagnostics...**

## Pon_Cache1

This test verifies the following:

- That the cache mapping of the R2000 is correct.
- That if the data is not in the cache, then the processor reads main memory and puts a copy of the data in the cache.
- That if the data is in the cache, then the processor does not access main memory but uses the data in the cache.
- That, during write operations, the data is written–through to memory.
- That valid bit logic is basically correct.

The three possible messages for the Pon_Cache1 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Cache Test #1...PASSED**
**Cache Test #1...SKIPPED**
**Cache Test #1...FAILED**

LSB

MSB

*LED Code 0x05*

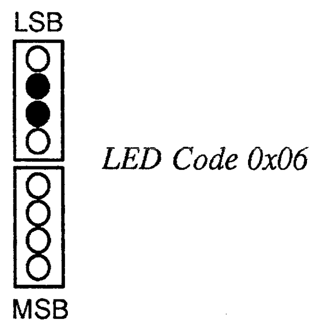
## Pon_Cache2

This test verifies the following:

- That the instruction cache gets loaded on instruction fetches.
- That the instruction cache is utilized when valid.
- That the instructions can execute at the rate of one instruction per clock when the instruction cache is valid.

The three possible messages for the Pon_Cache2 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Cache Test #2...PASSED**
**Cache Test #2...SKIPPED**
**Cache Test #2...FAILED**

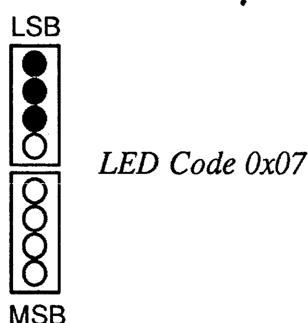


## Pon_Cache3

This test performs a MATS (modified algorithmic test sequence) memory test on the data cache to check for addressing and data stuck–at faults. A "stuck–at fault" occurs when a bit is stuck either high or low and cannot be toggled.

The three possible messages for the Pon_Cache3 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Data Cache MATS+ Test...PASSED**
**Data Cache MATS+ Test...SKIPPED**
**Data Cache MATS+ Test...FAILED**



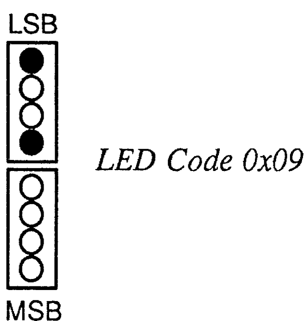

*LED Code 0x07*

## Pon_Cache4

This test performs a MATS (modified algorithmic test sequence) memory test on the instruction cache to check for addressing and data stuck–at faults. A "stuck–at fault" occurs when a bit is stuck either high or low and cannot be toggled.

The three possible messages for the Pon_Cache4 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Instruction Cache MATS+ Test...PASSED**
**Instruction Cache MATS+ Test...SKIPPED**
**Instruction Cache MATS+ Test...FAILED**



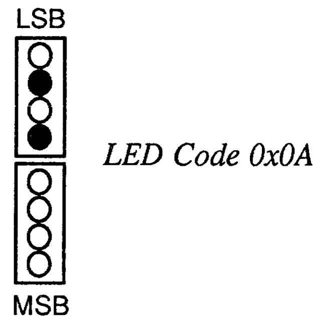

*LED Code 0x09*

## Pon_IdProm

This test computes and verifies the checksum in the ID PROM.

The two possible messages for the Pon_IdProm test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**ID PROM Test...PASSED**
**ID PROM Test...FAILED**

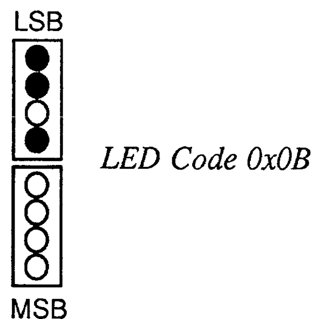



## Pon_WB

This test verifies that byte, half–word, tri–byte, and word write operations to memory are performed correctly. All writes go through the R2020 Write Buffers.

The two possible messages for the Pon_WB test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Write Buffer Test...PASSED**
**Write Buffer Test...FAILED**

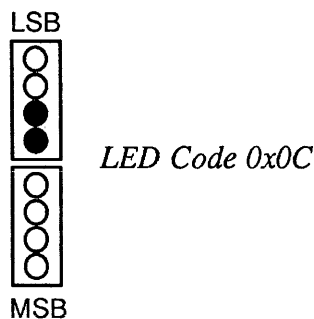


## Pon_Memory

This test determines how large the system memory is and then performs the Knaizuk Hartman algorithm memory test to check for addressing and data stuck–at faults. If the cache tests passed, then the memory tests will execute in cached mode. Otherwise, the test will execute in uncached mode. Executing this test in cached mode significantly reduces the test time.

The two possible messages for the Pon_Memory test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Memory Test...PASSED**
**Memory Test...FAILED**



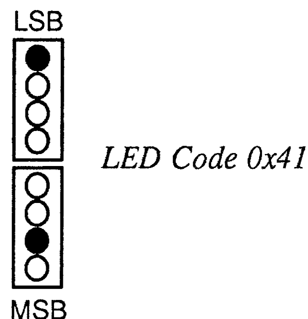

## Pon_Scr

This test is for the M/120 only.

This test is a write/read test for the writable/readable bits in the System Configuration Register (SCR).

The two possible messages for the Pon_Scr test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**SCR Test...PASSED**
**SCR Test...FAILED**



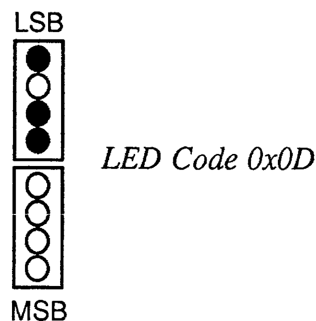
## Pon_VM

This test verifies the following things about the Translation Lookaside Buffer (TLB) in the R2000. Refer to the *MIPS R2000 RISC Architecture* book for a more complete description of the Translation Lookaside Buffer.

- Toggles each write/read bit in all TLB registers and verifies that all undefined bits are read back as zero.
- That all TLB slots respond to probes upon address match.
- That the virtual and physical addresses match.
- That the TLB entry is marked as valid when the valid bit is not set and an exception occurs.
- That page modification can be controlled by the dirty bit when the bit is not set and an exception occurs.
- That the global bit causes the R2000 to ignore the Process ID match requirement for valid translation.
- That the non–cacheable bit forces the R2000 to access memory instead of the cache.

The three possible messages for the Pon_VM test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Failed TLB Test...PASSED**
**Failed TLB Test...SKIPPED**
**Failed TLB Test...FAILED**

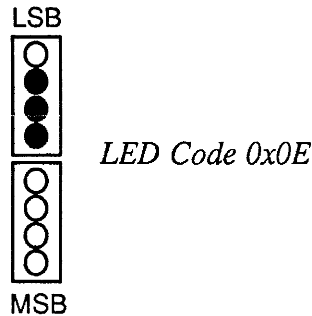


## Pon_Allexc

This test creates multiple exceptions simultaneously that check the R2000 and its handling of the exceptions. This test stores all possible sequences of three instructions that cause exceptions (excluding bus errors) into the last three locations of a page. The test then executes the instructions and checks to see if all the expected exceptions were generated.

The three possible messages for the Pon_Allexc test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**All Exception Test...PASSED**
**All Exception Test...SKIPPED**
**All Exception Test...FAILED**




*LED Code 0x0E*

## Pon_Parity

This test is for the M/120 only.

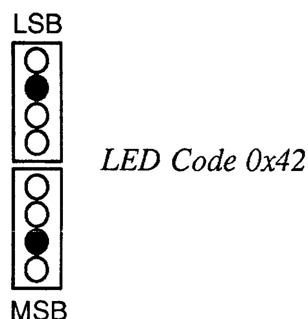
This test forces a byte parity error by using the "Force Bad Parity" feature in the System Configuration Register (SCR). The test then reads the word location and checks to see if the data was detected correctly. The test verifies the following:

- That parity errors are not detected if parity checking is disabled.
- That when parity is enabled and a "word" read is performed on the bad location, that an interrupt occurs and that the correct error information is contained in the system's Fault Address Register (FAR) and Fault ID Register (FID).
- That when parity is enabled and a "byte" (either the bad byte itself or one adjacent in the same word) read is performed on the bad location, that an interrupt occurs and that the correct error information is contained in the FAR and FID.

If this test passes, then a PASSED message is displayed on the screen following the test name. If the test is skipped because of a test dependency, then a SKIPPED message is displayed on the screen following the test name. The test also checks the information in both the FAR and FID. If the information is not what was expected, then a FAILED message is displayed on the screen.

The three possible messages for the Pon_Parity test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Parity Test...PASSED**
**Parity Test...SKIPPED**
**Parity Test...FAILED**



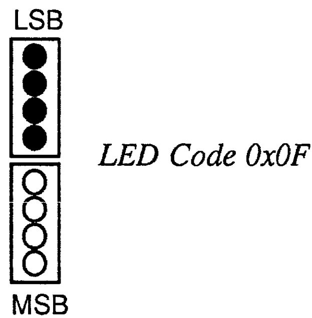

*LED Code 0x42*

## Pon_NVram

This test does a non–destructive write/read test of the non–volatile RAM in the system.

The possible messages for the Pon_NVram test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**NVRAM Test...PASSED**
**NVRAM Battery Test...PASSED**
**NVRAM Test...FAILED**
**NVRAM Battery Test...FAILED**

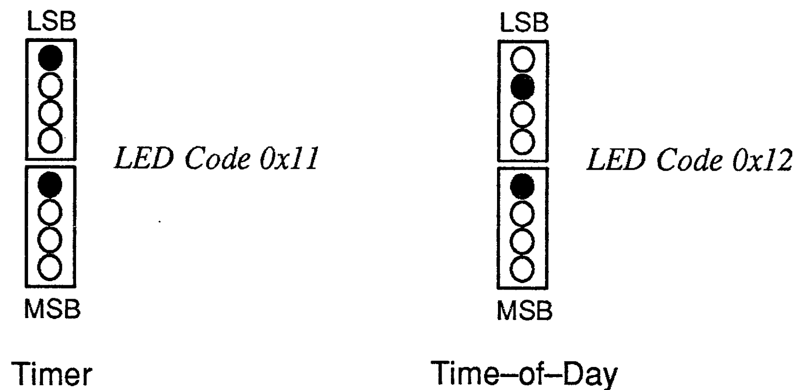


## Pon_Timers

This test checks the 8254 counter/timer and the MK48T02 time–of–day clock device. Two of the counters are programmed to generate an interrupt every 1/60th and 1/100th of a second. The time–of–day seconds is read and then each counter is checked for three seconds. The timer interrupts are polled at the R2000 and if a timeout occurs waiting for an interrupt, then a fail status is returned. After the second timer is checked for three seconds, the time–of–day seconds is read again. The elapse time for the test should be between 5 and 7 seconds (6 +/– 1) for the test to pass.

The possible messages for the Pon_NVram test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. One of the following LED patterns will flash for approximately seven seconds if one of the tests fail.

**8254 Timer Test...PASSED**
**Time–of–Day Clock Test...PASSED**
**8254 Timer Test...FAILED**
**Time–of–Day Clock Test...FAILED**



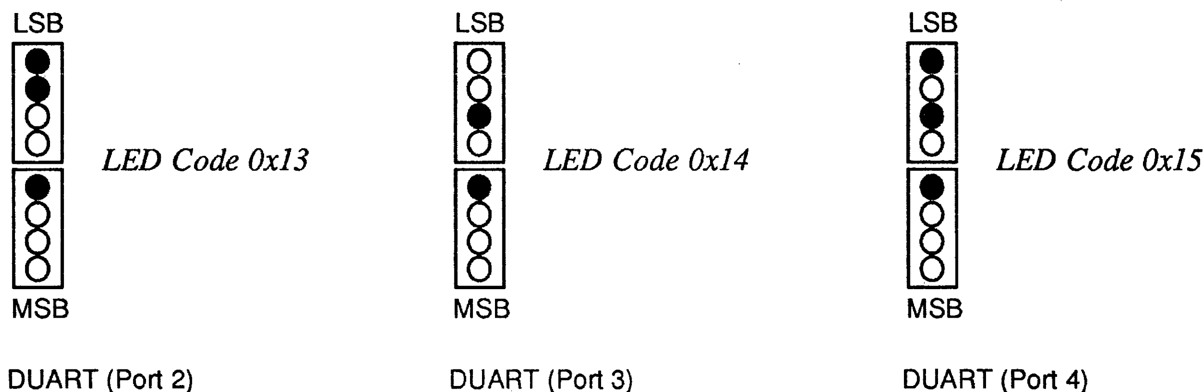

If the time–of–day test fails, then run the **init_tod** command from the PROM Monitor prompt to initialize the time–of–day clock.

## Pon_Duarts

This test is similar to the Pon_Duart test, which uses an internal loopback, except that it checks all DUART ports (excluding the console) at various baud rates. The baud rates tested are 19.2K, 9600, 7200, 4800, 2400, 600, and 300. Up to three ports are tested on the M120. Therefore, one of the following three LED patterns can be displayed when a failure occurs. This test is terminated after the first reported error. The Motherboard LEDs flash on and off in one of the patterns shown below if any one of the three DUART tests fail.

**Duart 1 Channel B Test...PASSED**
**Duart 2 Channel A Test...PASSED**
**Duart 2 Channel B Test...PASSED**

**Duart 1 Channel B Test...FAILED**
**Duart 2 Channel A Test...FAILED**
**Duart 2 Channel B Test...FAILED**




DUART (Port 2) DUART (Port 3) DUART (Port 4)

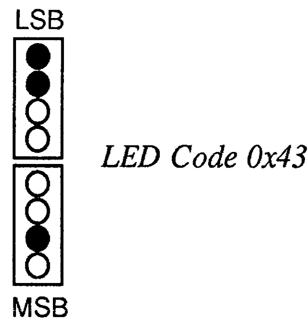
Once the test is complete, port 1 is re–programmed for normal operation, since this port can be used for downloading.

## Pon_Imr

This test is a write/read test of the system Interrupt Mask Register (IMR).

The two possible messages for the Pon_Imr test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**IMR Test...PASSED**
**IMR Test...FAILED**



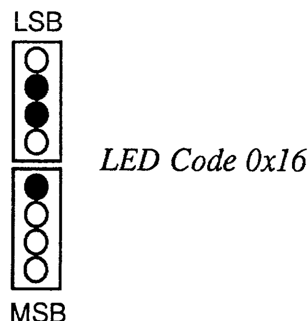

*LED Code 0x43*

## Pon_Fp1 and Fp2

These tests verify the basic 2010 floating point operations if a Floating Point Accelerator is detected in the system. The System Configuration Register is checked to determine if there is a Floating Point Accelerator in the system. This test also checks that the R2000 is able to communicate with the R2010 when performing basic operations, and checks that the R2010 is able to generate exceptions such as overflow, underflow, and divide by zero.

The three possible messages for the Pon_Fp1 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

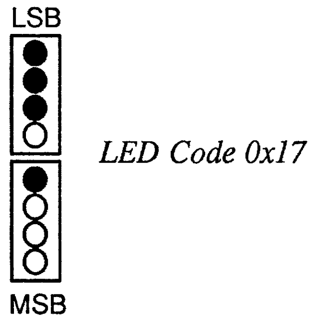
**FP Test #1...PASSED**
**FP Test #1...SKIPPED**
**FP Test #1...FAILED**




*LED Code 0x16*

The three possible messages for the Pon_Fp2 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**FP Test #2...PASSED**
**FP Test #2...SKIPPED**
**FP Test #2...FAILED**



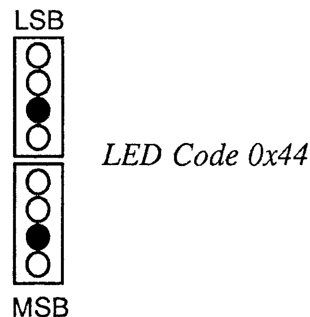

## Pon_UdcSlave

This test is for the M/120 only.

This test is a write/read test of the writeable/readable registers in the AMD 9516 UDC (Universal DMA Controller).

The two possible messages for the Pon_UdcSlave test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**UDC Slave Test...PASSED**
**UDC Slave Test...FAILED**



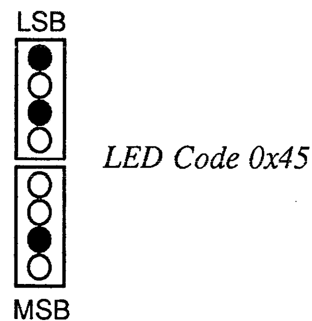
## Pon_Chain1

This test is for the M/120 only.

This test verifies the ability of the UDC to become bus master and read from main memory. The test causes the UDC to request control of the bus, access a pre–initialized buffer, and load its internal registers. At the completion of this operation (chaining), the UDC registers are read and compared with the contents of the buffer. If the data does not compare and match correctly, then a "FAILED" message is displayed on the console.

The three possible messages for the Pon_Chain1 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**UDC Channel 1 Chain Test...PASSED**
**UDC Channel 1 Chain Test...SKIPPED**
**UDC Channel 1 Chain Test...FAILED**



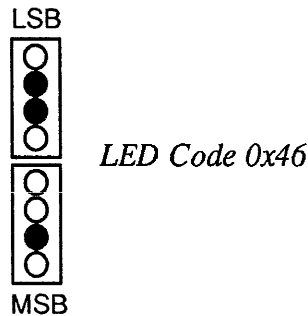
## Pon_Chain2

This test is for the M/120 only.

This test is the same as the Pon_Chain1 test, except that it uses channel 2 of the UDC.

The three possible messages for the Pon_Chain2 test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**UDC Channel 2 Chain Test...PASSED**
**UDC Channel 2 Chain Test...SKIPPED**
**UDC Channel 2 Chain Test...FAILED**



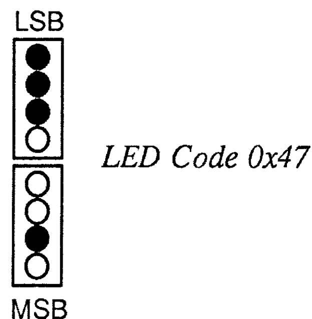

## Pon_ScsiSlave

This test is for the M/120 only.

This test is a write/read test of the writable/readable registers in the Fujitsu MB87030 SPC (SCSI Protocol Controller).

The two possible messages for the Pon_ScsiSlave test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**SCSI Slave Test ...PASSED**
**SCSI Slave Test...FAILED**



## Pon_ScsiMaster

This test is for the M/120 only.

This test initializes the SCSI Protocol Controller (SPC) and sets its SCSI device to ID 7 to give the system the highest priority on the SCSI Bus. This test then searches for other devices on the bus by issuing a Test Unit Ready command on all other SCSI device ID's and logical unit combinations. If a device is found and identified to be valid, an Inquiry command is issued to determine what type of device it is. Additional testing is done on the disk devices only. If any SCSI command to a device fails, then the next SCSI device ID is checked.

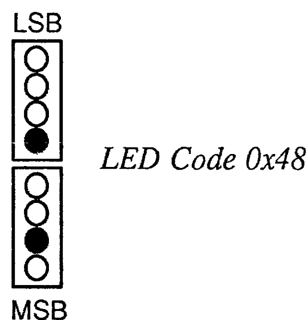
When a disk is found, its capacity (number of blocks) is determined and the Volume Header (usually physical block 0) is checked. A random read–only test is performed, which reads four different blocks on the disk. The first random block selected is read twice, at the start and end. The data is compared and checked to see if the disk seeked correctly. The read–only test assumes that the blocks selected will contain different data.

If the read–only test passes and there are scratch blocks in the Volume Header area of the disk, then the write/read test is performed. The write test is similar to the read–only test, except random data is written to the first random block selected (in the Volume Header area). Two random reads are done followed by a read of the first random block. The data read is compared to the data written.

During this test, interrupts are enabled at the SCSI Protocol Controller and the Interrupt Mask Register but are polled at the R2000.

The three possible messages for the Pon_ScsiMaster test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**SCSI Master Test...PASSED**
**SCSI Master Test...SKIPPED**
**SCSI Master Test...FAILED**




*LED Code 0x48*

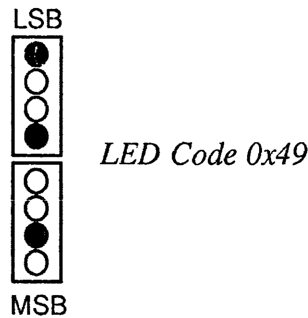
## Pon_EnetProm

This test is for the M/120 only.

This test computes and verifies the checksum in the Ethernet PROM.

The two possible messages for the Pon_EnetProm test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Ethernet ID PROM Test...PASSED**
**Ethernet ID PROM Test...FAILED**



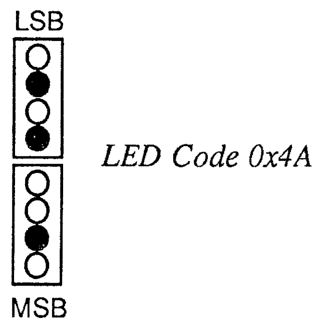

## Pon_LanceSlave

This test is for the M/120 only.

This test is a write/read test of the Lance registers. The STOP bit in the Control/Status Register 0 (CSR0) is held high during this test.

The two possible messages for the Pon_LanceSlave test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Lance Slave Register Test...PASSED**
**Lance Slave Register Test...FAILED**



## Pon_LanceMaster

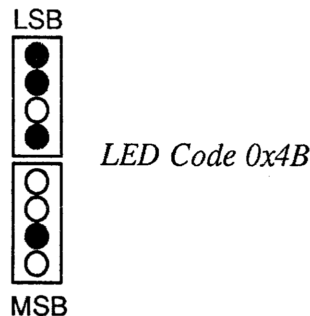
This test is for the M/120 only.

The AMD 7990 Lance is programmed for internal loopback operation. This test initializes the Lance and sets up transmit and receive descriptor rings in memory to communicate with the Lance. In loopback mode, the Lance reads the data from the transmit buffer into its internal buffer, and then writes back the data into a receive buffer. When this cycle is completed, the data in the transmit and receive buffers are compared.

During this test, interrupts are enabled at the Lance and the Interrupt Mask Register (IMR), but are polled at the R2000.

The three possible messages for the Pon_LanceMaster test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**Lance Master Test...PASSED**
**Lance Master Test...SKIPPED**
**Lance Master Test...FAILED**



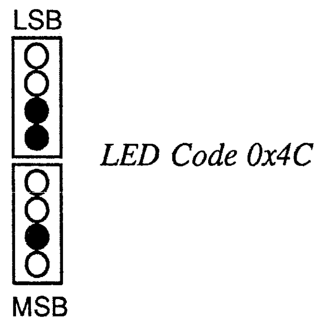
## Pon_Atreg

This test is for the M/120 only.

This test is a write/read test of the PC/AT register.

The two possible messages for the Pon_Atreg test are shown below. The LED pattern shown in the following illustration is displayed on the two Motherboard LED blocks when the test is started. This LED pattern will flash for approximately seven seconds if the test fails.

**AT Register Test...PASSED**
**AT Register Test...FAILED**



# Appendix E
# Sample Driver Listing

The following C listings are for a stream driver program for the DSC COM_8 serial card. The listings include the driver program, a customized header file, and the customized library that the driver program uses. You should be able to use much of this program and the library when writing your own serial device driver.

## The c8.c Driver Program

```
/* -------------------------------------------------------------------    */
/* | Copyright Unpublished, MIPS Computer Systems, Inc. All Rights        /*/
/* | Reserved. This software contains proprietary and confidential        | */
/* | information of MIPS and its suppliers. Use, disclosure or            | */
/* | reproduction is prohibited without the prior express written         | */
/* | consent of MIPS.                                                     | */
/* -------------------------------------------------------------------    */

/*
 *  c8 - terminal device driver for DSC COM-8 serial card.
 *
 *  Adapted from sample terminal driver in Microsoft XENIX
 *  Software Development Guide, Appendix C.
 *
 *  D. E. Messinger, D. E. Germann, 1985-05-14.
 *  Copyright (c) 1985, DSC.
 */

static char c8_copyright[] = "Copyright (c) 1985, DSC.\n";

#include "sys/sbd.h"
#include "sys/cpu_board.h"
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/systm.h"
#include "sys/signal.h"
#include "sys/pcb.h"
#include "sys/immu.h"
#include "sys/region.h"
#include "sys/fs/s5dir.h"
#include "bsd/sys/time.h"
```

```
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/termio.h"
#include "sys/file.h"
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/strids.h"
#include "sys/stty_ld.h"
#include "sys/sysinfo.h"
#include "sys/debug.h"
#include "sys/cmn_err.h"
#include "sys/cpreg.h"
#include "sys/fs/bfs_bio.h"
#include "sys/buf.h"
#include "sys/edt.h"
#include "sys/ss.h"

/* registers */
#define RRDATA 0    /* received data */
#define RTDATA 0    /* transmitted data */
#define RSTATUS 5   /* status */
#define RMstat 6    /* modem status */
#define RMctrl 4    /* modem control */
#define RCtrl  3    /* line control */
#define RIENABL 1   /* interrupt enable */
#define RSPEED 0    /* data rate */
#define RIIR   2    /* interrupt identification */

/* status register information */
#define SRRDY  0x01 /* received data ready */
#define STRDY  0x20 /* transmitter ready */
#define SOERR  0x02 /* received data overrun */
#define SPERR  0x04 /* received data parity error */
#define SFERR  0x08 /* received data framing error */
#define SDCD   0x80 /* status of carrier detect */
#define SCTS   0x10 /* status of CTS */

/* control register */
#define CBITS5  0x00 /* five bit characters */
#define CBITS6  0x01 /* six bit characters */
#define CBITS7  0x02 /* seven bit characters */
#define CBITS8  0x03 /* eight bit characters */
#define CDTR    0x01 /* data terminal ready */
#define CRTS    0x02 /* request to send */
#define CSTOP2  0x04 /* two stop bits */
#define CPARITY 0x08 /* parity on */
#define CEVEN   0x10 /* even parity */
#define CBREAK  0x40 /* send break (space) */
#define CDLAB   0x80 /* enable divisor latch access */
```

```
/* interrupt enable */
#define ERECV   0x01  /* receiver ready */
#define EXMIT   0x02  /* transmitter ready */
#define ERLSTAT 0x04  /* line status interrupt */
#define EMS     0x08  /* modem status change */
#define ALLINTS (EXMIT | ERECV | EMS)
#define ENINTR  0x08  /* board interupt enable */

/* interrupt identification */
#define IRLSTAT 0x06  /* recieve line status interrupt */
#define IRECV   0x04  /* Receive interrupt active */
#define IXMIT   0x02  /* Transmint interrupt active */
#define IMS     0x00  /* Modem interrupt active */
#define IMASK   0x07  /* Documentation lies. Upper bits are not alays zero */
#define IACTIVE 0x01  /* when bit is 0=> interrupt is active */

/* status register info */
#define C8IDLE            0xff
#define c8devtoboard(x)   ((x>>3) & 3)
#define c8devtoline(x)    (dev&7)

/* Minor device number decoding */
#define   C8UNIT(m)    (((m)>>3)&0x3)
#define   C8LINE(m)    ((m)&0x07)

#define C8_BMAX    4  /* Max number of boards */
#define C8_LMAX    8  /* Max lines per board  */
#define C8_OUTCMAX 1  /* Max number of chars which can be output by outc */

/* Base address of AT Bus I/O Space */
#define ATBASEADDR 0x13000000

/*
 * Define the Standard Board and Status Registers.
 * These addresses are only used to catch premature interrupts.
 * The operational addresses for the DSC COM-8 are obtained from
 * the edt information vector passed to c8edtinit.
 * Default addrs: IRQ7, Base 140, Status 13E
 */
#define C8ADDR     (ATBASEADDR + 0x140)
#define C8STADDR   (ATBASEADDR + 0x13e)

#define C8ADDRINC 0x40 /* inc to next board addr */

int c8_act(), c8_zap(), c8_outc(), c8_setline();

struct ss_devdep_info c8dd =
    { "C8", SS_TTY, C8_BMAX, C8_LMAX, C8_OUTCMAX,
      c8_act, c8_zap, c8_outc, c8_setline, ss_null, ss_null
    };
```

```
struct ss_struct c8board[C8_BMAX];

/*
 * Interrupt status register for all boards.
 * Set to default addr, to do something for premature ints.
 */
int c8_inta = C8STADDR;

/*
 * The Digi Com 8 board can set a separate I/O address for each
 * line. System configuration conventions will probably allocate
 * line addresses contiguously; however, the driver will use an
 * array to hold the address for each line.
 */

/*
 * Line local information is connected to ss structure using ss_llocal
 * Note: Since there is only one element in line info,
 * it could occupy the ss_llocal entry in the ss_line array. However, there is
 * no need for this extra level of indirection.
 */
 struct c8_linfo {
   int li_addr;    /* line address */
};

/* allocate storage for line info */
struct c8_linfo c8_linfo[C8_BMAX][C8_LMAX];

/*
 * The digi board presents interrupts which are delivered to c8intr
 * before the c8edtinit procedure is called.
 *
 * c8_initdone == 0 => ignore interrupts. c8edtinit has not completed.
 */
int c8_initdone;

/*
 * streams linkage information
 */
static struct module_info dum_info = {
  STRID_MUX,    /* module ID */
  "MUX",        /* module name */
  0,            /* minimum packet size */
  1024,         /* maximum packet size */
  128,          /* high water mark */
  16,           /* low water mark */
};
```

```
int c8_open();

static struct qinit c8_rinit = {
  NULL, ss_rsrv, c8_open, ss_close, NULL, &dum_info, NULL
};

static struct qinit c8_winit = {
  ss_wput, NULL, NULL, NULL, NULL, &dum_info, NULL
};

struct streamtab c8info = {&c8_rinit, &c8_winit, NULL, NULL};

/*
 * This allows for selected print statements to be turned on and off
 * while the kernel is running.
 */
/* c8 debug points */
#define C8DBG_PT1   0x0001      /* interrupt status */
#define C8DBG_PT2   0x0002      /* transmitt flow */
#define C8DBG_PT3   0x0004      /* reciever flow */
#define C8DBG_PT4   0x0008      /* ioctl flow */
#define C8DBG_PT5   0x0010      /* DCD interrupts */
#define C8DBG_PT6   0x0020      /* if set, allow spurious intr prints */
int c8_print = 0;

/* baud rate conversion table */
#define C8_BR    16  /* number of possible baud rates */
int c8_bconv[C8_BR] = {
  /* 0    */     0,
  /* 50   */  2304,
  /* 75   */  1536,
  /* 110   */  1047,
  /* 134   */   857,
  /* 150   */   768,
  /* 200   */   576,
  /* 300   */   384,
  /* 600   */   192,
  /* 1200   */   96,
  /* 1800   */   64,
  /* 2400   */   48,
  /* 4800   */   24,
  /* 9600   */   12,
  /* EXTA   */    6,   /* 19200 */
  /* EXTB   */   58    /* 2000 */
  };
```

```
/*
 * The Digi Com-8 board does not have a hardware input silo. Due to the
 * interrupt latency of UNIX, a software silo is used to quickly collect the
 * data now, and process it later.  Note that two silos are used to process
*incomming data before outgoing data.
*/

 *
 * Software managed input silos
 */
#define SILOMAX (C8_BMAX*C8_LMAX*2)
struct c8_silo {
  struct ss_line   *si_c8l;
   uchar      si_iir;
   uchar      si_status;
   uchar      si_c;
};
struct c8_silo c8_rsilo[SILOMAX];
struct c8_silo c8_tsilo[SILOMAX];
int c8_rsiloread, c8_rsilowrite;
int c8_tsiloread, c8_tsilowrite;
c8_maxsilo;

int c8_brktmr();

/*
 * Write I/O byte
 */
outb(addr, value)
   int addr;
   uchar value;
{
   register volatile uchar *ioaddr;

   ioaddr = (volatile uchar *) PHYS_TO_K1(addr);
   *ioaddr = value;
   wbflush();
}

/*
 * Read I/O byte
 */
uchar
inb(addr, value)
   int addr;
   uchar value;
{
   register volatile uchar *ioaddr;
```

```
    ioaddr = (volatile uchar *) PHYS_TO_K1(addr);
    value = *ioaddr;
    return (value);
}

/*
 * Open a spcific tty line.
 * This routine decodes the device number into a board and line number
 * and calls ss_open to do the real work.
 *
 * Called from:
 * s5openi
 *   stropen
 *
 */
c8_open (rq, dev, flag, sflag)
    queue_t *rq;
    dev_t   dev;
    int     flag;
    int     sflag;
{
    register struct ss_struct *c8b;
    register struct ss_line    *c8l;   /* line of interest */
    register int line, myminor;
    register int   c8addr;

    myminor = minor(dev);
    c8b = &c8board[C8UNIT(myminor)];
    line = C8LINE(myminor);
    if (C8UNIT(myminor) > C8_BMAX || line >= c8b->ss_nlines ||
        c8b->ss_line[line].ss_lenable == 0)
          return(OPENFAIL);

    c8l = &c8b->ss_line[line];

    c8addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;
    outb(c8addr + RIENABL, ALLINTS); /* allow ints for this port */

    return (ss_open (c8l, rq, dev, flag, sflag));
}

/*
 * c8edtinit() initializes the controller structure,
 * probes for the presence of the board, and
 * enables lines that are present.
 *
 * Called from:
 * main
 */
c8edtinit(e)
```

```
  struct edt *e;
{
  register struct ss_struct *c8b;
  register struct ss_line  *c8l;
  register int  c8addr;
  register int  c8board_addr;
  register int  ctlr;
  register int  num_ctlrs = 0;
  register int  lmax;
  register int  line;
  register int  numlines;  /* diagnostic output control */
  register unsigned short newimr;
  register int  linesfound=0;

  /*
   * This section of logic performs a structured initialization.
   * It is separated into two controller loops that protect the driver from
   *unexpected interrupts by using partially initialized structures.
   * (Contrary the the hardware documentation, this really happens)
   */

  /*
   * Set addr of status register
   * If ODD IRQ, then use ODD status address
   */
  if(e->e_atbusintr_info->a_irq & 1);
    c8_inta  = (ATBASEADDR + e->e_base) - 1;
  else
    c8_inta  = (ATBASEADDR + e->e_base) - 2;

  for (ctlr = 0; ctlr < C8_BMAX; ctlr++) {
    c8b = &c8board[ctlr];

    if (C8_LMAX > SS_MAXLINES) {
      cmn_err(CE_CONT,"c8init: ss lib can only handle %d lines\n",
                                SS_MAXLINES);
      lmax = SS_MAXLINES;
    }
    else
      lmax = C8_LMAX;
    c8b->ss_nlines = lmax;

    c8b->ss_devdep = &c8dd;
    c8b->ss_bconv  = c8_bconv;

    c8board_addr = (ATBASEADDR + e->e_base) + (ctlr * C8ADDRINC);

    for (c8l = &c8b->ss_line[0], line=0; c8l < &c8b->ss_line[lmax];
                               c8l++, line++)
    {
```

```
            c8l->ss_line   = line;
            c8l->pss       = c8b;
            c8l->ss_lenable = 0;     /* disable */

            /* make local pointer point to associated storage area */
            c8l->ss_llocal = (char *) &c8_linfo[ctlr][line];

            /* remember line address */
            ((struct c8_linfo *) c8l->ss_llocal)->li_addr =
                    c8board_addr + (line * 8);
        }
    }

    /* Now talk to the hardware */
    for (ctlr = 0; ctlr < C8_BMAX; ctlr++) {
      c8b = &c8board[ctlr];

            if (showconfig)
        cmn_err(CE_CONT,"c8init: Board %d: ",ctlr);
      numlines = 0;

      for (c8l = &c8b->ss_line[0], line=0; c8l < &c8b->ss_line[lmax];
                                c8l++, line++)
      {
        c8addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;

        /* Check for the presence of a board/line */
        outb(c8addr + RCtrl, 0x55);
        if (inb(c8addr + RCtrl) != 0x55) {
          /* Line is not present */
          continue;
        }

        if (numlines == 0) {
          if (showconfig)
                    cmn_err(CE_CONT,"lines:");
          numlines++;
          linesfound++;
        }

        if (showconfig)
                  cmn_err(CE_CONT," %d",line);

        c8l->ss_lenable = 1;

        /*
         * Sets the line control register to 0 and disables all interrupts.
         * The interupts are not enabled until an ss_open() occurs.
         * The digi board often generates premature interrupts.
         * Enable "Board" interrupts for this line.
         */
```

```
      outb(c8addr + RCtrl, 0);      /* Line control register */
      outb(c8addr + RIENABL, 0);    /* disable ints for this port */
            c8_iodelay();
      outb(c8addr + RMctrl, ENINTR); /* allow board interrupts */
    }

    /* finishes up line availability diagnostic output */
          if (showconfig) {
      if (numlines == 0)
        cmn_err(CE_CONT,"NO LINES PRESENT \n");
      else
        cmn_err(CE_CONT," PRESENT \n");
          }
  }

  /* allow interrupts at this point */
  if (linesfound) {
    irq_unmask(e->e_atbusintr_info->a_irq);
    newimr = *(volatile unsigned short *)PHYS_TO_K1(IMR);
          if (showconfig)
      cmn_err(CE_CONT,"c8init: Unmask IRQ %d, new IMR = %x\n",
        e->e_atbusintr_info->a_irq, newimr);

    c8_initdone = 1;
  }
}

/*
 * Activate the specified line
 * and return True if the carrier (DCD) is up
 *
 * Since this routine is checking and returning the state of the line,
 * the caller must guarantee that hardware interrupts are blocked.
 *
 * Called from:
 * ss_open
 */
c8_act (c8l)
  register struct ss_line   *c8l;
{
  register int addr;
  register int stat;

  addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;

  outb(addr+RMctrl, inb(addr+RMctrl) | CDTR | CRTS);

  stat = inb(addr+RMstat);

  if (stat & SDCD)
    return(1);
```

```
  else
    return(0);
}

/*
 * Turn off the specified line
 */
c8_zap (c8l)
  register struct ss_line *c8l;
{
  register int addr;

  addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;

  outb(addr+RMctrl, inb(addr+RMctrl) & ~CDTR & ~CRTS);
}

/*
 * This procedure outputs "len" chars pointed to by "cp" on line "cpl".
 * Called from:
 * ss_tx
 */
c8_outc(c8l, cs, len)
  register struct ss_line   *c8l;
  register char *cs;
  register int  len;
{
  register int addr;

  ASSERT(len <= 1); /* the Digi C8 board can only do one character at a time */

  if (len) {
    sysinfo.outch += len;
    c8l->ss_state |= SS_BUSY;

    /*
     * Ensure that we do not have new data, before giving the controller
     * more work.
     */
    c8_addsilo(C8IDLE);

    addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;
    outb(addr+RTDATA, *cs);

  }
}
```

```
/*
 * This procedure sets the baud, parity, and line parameters
 *that are contained in "cflag" of the line "c8l".
 */
c8_setline (c8l, cflag)
  register struct ss_line   *c8l;
  int cflag;
{
  register int addr;
  register int temp;

  addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;

  /* set up speed */
  outb(addr+RCtrl, inb(addr+RCtrl) | CDLAB);
  c8_iodelay();

  temp = c8l->pss->ss_bconv[cflag & CBAUD];

  outb(addr+RSPEED, temp & 0xff);
  c8_iodelay();
  outb(addr+RSPEED+1, temp >> 8);
  c8_iodelay();
  outb(addr+RCtrl, inb(addr+RCtrl) & ~CDLAB);
  c8_iodelay();

  /* set up line control */
  temp = (cflag & CSIZE) >> 4;   /* length */
  if (cflag & CSTOPB)
    temp |= CSTOP2;
  if (cflag & PARENB) {
    temp |= CPARITY;
    if ((cflag & PARODD) == 0)
      temp |= CEVEN;
  }
  outb(addr+RCtrl, temp);
}
```

```
/*
 * Hardware level interrupt procedures.
 * that gather valid input and place it in the silo.
 * This data will be processed by c8_dispatch.
 */
c8intr(vec)
int vec;
{
  register int dev;
  register int validint=0;

  /* Clear interrupt, gather input and place in silo */
  if ((dev = inb(c8_inta)) != C8IDLE) {

    validint = 1;

    if (c8_initdone == 0) {
      c8_gobbleint(dev);
      return;
    }

    c8_addsilo(dev);
  }

  /*
   * In the future, c8_dispatch will be called at a lower interrupt level.
   * This will allow the software silo to protect the driver from line overflows.
   */
  c8_dispatch();

  if (!validint) {
    if (c8_print & C8DBG_PT6)
      cmn_err(CE_CONT,"c8: spurious interrupt\n");
  }
}

/*
 * Process the silo data.
 */
c8_dispatch()
{
  uchar iir;
  uchar c;
  uchar status;   /* line status */
  struct ss_line   *c8l;

  /* now process the input */
  for (; c8_rsiloread!=c8_rsilowrite || c8_tsiloread!=c8_tsilowrite; ) {

    /* get silo info, advance c8_siloread pointer */
    c8_getsilo(&c8l, &iir, &status, &c);
```

```
    switch (iir) {
      case IRLSTAT:
      case IRECV:
        c8_rint(c8l, c, status);
        break;

      case IXMIT:
        c8_tint(c8l, status);
        break;

      case IMS:
        c8_cint(c8l, status);
        break;
    }
  }
}

/*
 * "Gobble up" premature interrupts
 * Once the premature interrupt issues are resolved, this routine, which
 * uses default board addresses, should be removed.
 */
c8_gobbleint(dev)
  register int dev;
{
  uchar iir;
  uchar c;
  uchar status;  /* line status */
  struct ss_line  *c8l;

  register board, line;
  register int addr;

  for (; dev != C8IDLE ;dev=inb(c8_inta)) {
    board = c8devtoboard(dev);
    line  = c8devtoline(dev);

    cmn_err(CE_CONT,"\nc8[%d][%d]: ",board,line);
    cmn_err(CE_CONT,"Interrupt before c8init is called/done\n");

    addr = C8ADDR + (board * C8ADDRINC) + line*8;

    while (((iir = (inb(addr+RIIR) & IMASK)) & IACTIVE) == 0) {
      switch (iir) {
        case IRLSTAT:
        case IRECV:
          c    = inb(addr+RRDATA);
          status = inb(addr+RSTATUS);
          break;
        case IXMIT:
          status = inb(addr+RSTATUS);
```

```
          break;
        case IMS:
          status = inb(addr+RMstat);
          break;
        default:
          break;
      }
    }
  }
}

/*
 * Collect all the currently active events and add them to the silo.
 */
c8_addsilo(dev)
  register int dev;
{
  register uchar iir;
  register uchar c;
  register uchar status;   /* line status */
  register struct ss_line   *c8l;

  register board, line;
  register int addr;

  if (dev == C8IDLE)
    dev = inb(c8_inta);

  for (; dev != C8IDLE ;dev=inb(c8_inta)) {

    board = c8devtoboard(dev);
    line  = c8devtoline(dev);

    c8l  = &c8board[board].ss_line[line];
    addr = ((struct c8_linfo *) c8l->ss_llocal)->li_addr;

    while (((iir = (inb(addr+RIIR) & IMASK)) & IACTIVE) == 0) {
      switch (iir) {
        case IRECV:
          status = inb(addr+RSTATUS);

          c = inb(addr+RRDATA);

                      if (c8l->ss_lenable) {
                              c8_putsilo(c8l, iir, status, c);
                      } else {
                              cmn_err(CE_CONT,"\nc8[%d] line%d: unexcepted ",
                                        board, line);
                              cmn_err(CE_CONT,"receive interrupt\n" );
                      }
          break;
```

```
        case IXMIT:
          status = inb(addr+RSTATUS);

          if (c8l->ss_lenable) {
            c8_putsilo(c8l, iir, status, c);
          } else {
            cmn_err(CE_CONT,"\nc8[%d] line%d: unexpected ",
                board, line);
            cmn_err(CE_CONT,"transmit interrupt\n" );
          }
          break;

        case IMS:
          /* Clear interrupt. Get status */
          status = inb(addr+RMstat);

          if (c8l->ss_lenable)  {
            c8_putsilo(c8l, iir, status, c);
          } else {
            cmn_err(CE_CONT,"\nc8[%d] line%d: unexpected ",
                board, line);
            cmn_err(CE_CONT,"modem interrupt\n" );
          }
          break;

        default:
          cmn_err(CE_CONT,"\nc8[%d][%d]: ",board,line);
          cmn_err(CE_CONT,"OTHER interrupt %x\n",iir);
          break;
      }
    }
  }
}

/*
 * Place a single event on the silo
 */
c8_putsilo (c8l, iir, status, c)
  register struct ss_line  *c8l;
  register uchar      iir;
  register uchar      status;
  register uchar      c;
{
  register struct c8_silo *siloptr;
  register int silo_size;


  if (iir == IRECV)
          siloptr = &c8_rsilo[c8_rsilowrite];
  else
          siloptr = &c8_tsilo[c8_tsilowrite];
```

```
    siloptr->si_c8l   = c8l;
    siloptr->si_iir   = iir;
    siloptr->si_status = status;
    siloptr->si_c     = c;

    if (iir == IRECV) {
            if (++c8_rsilowrite == SILOMAX)
            c8_rsilowrite = 0;

            if (c8_rsilowrite == c8_rsiloread) {
            cmn_err(CE_CONT,"c8: rsilo overflow\n");
            c8_rsilowrite--;
            if (c8_rsilowrite < 0)
            c8_rsilowrite = SILOMAX - 1;
            }
    }else{
            if (++c8_tsilowrite == SILOMAX)
            c8_tsilowrite = 0;

            if (c8_tsilowrite == c8_tsiloread) {
            cmn_err(CE_CONT,"c8: tsilo overflow\n");
            c8_tsilowrite--;
            if (c8_tsilowrite < 0)
            c8_tsilowrite = SILOMAX - 1;
            }
    }
}

/*
 * Remove a single event from the silo.
 * The read silo is processed before the write silo to reduce line the
 * possibility of line overflow.
 */
c8_getsilo (c8l, iir, status, c)
    struct ss_line   **c8l;
    uchar     *iir;
    uchar     *status;
    uchar     *c;
{
    register struct c8_silo *siloptr;

    if (c8_rsilowrite != c8_rsiloread)  {
            siloptr = &c8_rsilo[c8_rsiloread];

        *c8l   = siloptr->si_c8l;
        *iir   = siloptr->si_iir;
        *status = siloptr->si_status;
        *c     = siloptr->si_c;

        if (++c8_rsiloread == SILOMAX)
          c8_rsiloread = 0;
```

```
  } else
  if (c8_tsilowrite != c8_tsiloread) {
          siloptr = &c8_tsilo[c8_tsiloread];

    *c8l    = siloptr->si_c8l;
    *iir    = siloptr->si_iir;
    *status = siloptr->si_status;
    *c      = siloptr->si_c;

    if (++c8_tsiloread == SILOMAX)
      c8_tsiloread = 0;
  } else {
      cmn_err(CE_CONT,"c8: silo empty\n");
      return;
  }
}

/*
 * Process receive data interrupt.
 *
 * Called from:
 * c8intr
 */
c8_rint(c8l, c, status)
  register struct ss_line   *c8l;
  register int c;
  register int status;
{
  register int c8;
  register int err_frame = 0;
  register int err_par = 0;
  register int err_overrun = 0;
  register int any_brks = 0;      /* breaks to timeout for */

  if ((status & SRRDY) == 0 )
    return;

  if (!(c8l->ss_state & SS_ISOPEN))
    return;

  c8 = c8l->pss - &c8board[0];   /* get ss structure index */

  /*
   *  Were there any errors on input?
   */
  if (status & SOERR)       /* overrun error */
    err_overrun++;
  if (status & SPERR)       /* parity error */
    err_par++;
  if (status & SFERR)       /* framing error */
    err_frame++;
```

```
    /* process start stop */
    if (ss_startstop (c8l, c))
    {
      /* have a legitimate char */

      /*
       * Data Overrun error
       */
      if (err_overrun) {
        c8l->ss_overflow++;
        /*
        cmn_err(CE_CONT,"c8[%d]: line %d overflow\n",
          c8, c8l->ss_line);
        */
      }

      if (err_frame) {
        if (c8_print & C8DBG_PT3) {
          cmn_err(CE_CONT,"c8_sint: framing error\n");
        }

        if (!(c8l->pss->ss_breaks & (1 << c8l->ss_line))) {
          c8l->pss->ss_breaks |= (1 << c8l->ss_line);
          any_brks |= (1 << c8l->ss_line);
        }
        else
          /* ignore breaks while break timeout is out standing */
          return;

        if (c8l->ss_iflag & PARMRK) {
          ss_slowr (c8l, 0377);
          ss_slowr (c8l, 0);
        }
        else {
          c = '\0';
        }
      }

      /*
       * break timeout
       */

      if (any_brks) {
        timeout(c8_brktmr, ((c8 << 16) |
              (any_brks & 0xFFFF)), HZ/3);
      }

      /* put char on the input queue */
      ss_inc (c8l, c);
    }
  }
```

```
/*
 * Process transmit interrupt.
 *
 * Called from:
 * c8intr
 */
c8_tint(c8l, status)
  register struct ss_line   *c8l;
  register int status;
{
  register mblk_t *wbp;

  sysinfo.xmtint++;

  if (status & STRDY)
  {
    c8l->ss_state &= ~SS_BUSY;

    /*
     * If the user process or line discipline
     * sent ^S/^Q we won't have a write buffer
     */
    if (wbp = c8l->ss_wbp) {
      /* account for character output */
      wbp->b_rptr++;
    }

    if (!(c8l->ss_state & SS_TXSTOP))
      ss_tx (c8l);
  }
}

/*
 * Process carrier transition interrupt.
 *
 * Called from:
 * c8intr
 */
c8_cint(c8l, status)
  register struct ss_line *c8l;
  register int status;
{
  register struct tty *tp;

  if (status & SDCD) {
    if (!(c8l->ss_state & SS_DCD))
      /* Call ss to mark Carrier as on */
      ss_con (c8l);
  } else {
    if (c8l->ss_state & SS_DCD)
```

```
        /* Call ss to mark Carrier as off */
        ss_coff (c81);
    }
}

/**** timers ****/
/*
 * Clear breaks on lines that are passed in information.  Breaks will now be
 * accepted on these ports.
 */

int
c8_brktmr(info)
uint info;
{
    register uint board;

    board = (info >> 16) & 0xFFFF;
    c8board[board].ss_breaks &= ~(info & 0xFFFF);

}

/* slow down. Don't hammer registers so fast */
c8_iodelay()
{
    int i;

    for (i=0; i < 100 ; i++);
}
```

# The Header file ss.h

```
/* Make sure your driver has less than or equal to this number of lines */

#define  SS_MAXLINES        16

/* ssdd_dev_type */
#define SS_TTY  0
#define SS_OTHER 1

/* used by drivers for null hardware procs */
int ss_null();

/* ss stream entry points. Used by drivers in qinit structures */
int ss_open(), ss_rsrv(), ss_close(), ss_wput();

struct ss_devdep_info {
        char  *ssdd_devname;        /* Name of the device, for interest */
        int   ssdd_dev_type;        /* SS_TTY, SS_OTHER... */
        int   ssdd_maxboards;       /* MAX number of this type of board */
        int   ssdd_maxlines;        /* # of ports on per board */
        int   ssdd_maxoutc;         /* max chars that outc can take at one time */
        int  (*ssdd_act)();         /* Hardware, activate line */
        int  (*ssdd_zap)();         /* Hardware, deactivate line */
        int  (*ssdd_outc)();        /* Hardware, output characters */
        int  (*ssdd_setline)();/* Hardware, set line params */
        int  (*ssdd_nsopen)();      /* NonSerial open */
        int  (*ssdd_nsclose)();/* NonSerial close */
};

/*
 * Define uchar, rather than standard unchar,
 * to be compatible with ushort, uint, and ulong.
 */
typedef unsigned char        uchar;

struct ss_struct {
        int               ss_nlines;          /* # of ports on board */
        caddr_t           ss_addr;  /* board address */
        int               *ss_bconv;          /* baud rate conversion table */
        struct ss_devdep_info *ss_devdep; /* per board hardware dep info */
        struct ss_line {
                queue_t   *ss_rq,
                          *ss_wq;
                mblk_t    *ss_rmsg,           /* current message */
                          *ss_rmsge,
                          *ss_rbp,
                          *ss_wbp;
                char   *ss_llocal;  /* loophole line dependent stuff */
                int       ss_rmsg_len,
```

```
                        ss_rbsize;
                int     ss_line; /* line number */
                int     ss_lenable;     /* 1=> line is enabled */
                int     ss_tid;         /* service timer id */
                int     ss_state;       /* current state of port */
                int     ss_lcc;         /* last char count for xfer */
                int     ss_allocb_fail,
                        ss_rsrv_cnt;    /* non zero => delay timer on */
                int     ss_ldebug;      /* SS line debug flags */
                int     ss_overflow;    /* count of overflow errs */
                uchar   ss_litc; /* escape next char */
                uchar   ss_stopc;
                uchar   ss_startc;      /* start char */
                struct termio ss_termio;
#define ss_iflag ss_termio.c_iflag
#define ss_cflag ss_termio.c_cflag
#define ss_ttyline ss_termio.c_line
                struct  ss_struct *pss; /* pointer to controller struct */

        } ss_line[SS_MAXLINES];
        uint    ss_breaks;              /* Breaks that have happened */
};

/* bits in ss_state */
#define SS_ISOPEN  0x0001       /* device is open            */
#define SS_WOPEN   0x0002       /* waiting for carrier               */
#define SS_DCD          0x0004          /* we have carrier           */
#define SS_TIMEOUT 0x0008       /* delaying                          */
#define SS_BREAK   0x0010       /* breaking                          */
#define SS_BREAK_QUIET  0x0020          /* finishing break           */
#define SS_TXSTOP  0x0040       /* output stopped by received XOFF */
#define SS_LIT          0x0080          /* have seen literal character*/
#define SS_BLOCK   0x0100       /* XOFF sent because input full */
#define SS_TX_TXON 0x0200       /* need to send XON          */
#define SS_TX_TXOFF     0x0400          /* need to send XOFF         */
#define SS_BUSY         0x0800          /* xmit in progress          */

#define SS_FLOW         0x4000          /* do hardware flow control  */

#define IFLAGS  (CS8|HUPCL|CLOCAL|SSPEED)

/* bits in ss_ldebug */
#define SS_DBG_RSRVSLEEP 0x0001         /* set => waiting to be awoken */
#define SS_DBG_QENBSLEEP 0x0002         /* set => ss_inc did not qenable */

#define SSIDLE 0
#define SSALERT 1
```

# The ss.c Library

```
/*-----------------------------------------------------------------      */
/* | Copyright Unpublished, MIPS Computer Systems, Inc. All Rights /      */
/* | Reserved. This software contains proprietary and confidential |      */
/* | information of MIPS and its suppliers. Use, disclosure or   |        */
/* | reproduction is prohibited without the prior express written |       */
/* | consent of MIPS.                          | */
/*-----------------------------------------------------------------      */
/* $Header: ss.c,v 1.5 88/04/28 12:46:20 straff Exp $ */

/*
 *      ss - Serial Stream -- Stream support for serial (tty) drivers
 *
 *      Adapted from MIPS cp.c, the Integrated Solutions {Intelligent}
 *      Communications Processor driver.
 */

#include "sys/sbd.h"
#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/systm.h"
#include "sys/signal.h"
#include "sys/pcb.h"
#include "sys/immu.h"
#include "sys/region.h"
#include "sys/fs/s5dir.h"
#include "bsd/sys/time.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/termio.h"
#include "sys/file.h"
#include "sys/stream.h"
#include "sys/stropts.h"
#include "sys/strids.h"
#include "sys/stty_ld.h"
#include "sys/sysinfo.h"
#include "sys/debug.h"
#include "sys/cmn_err.h"
#include "sys/fs/bfs_bio.h"
#include "sys/buf.h"
#include "sys/ss.h"

extern struct stty_ld def_stty_ld;

#define  TTY_NODELAY
```

```
                    /* Let's use REAL delay */
/* #define DELAY(x) { register int _N_ = x; while (_N_--); } */
#define MIN(a, b)   ((a < b) ? a : b);

/*
 * This allows for selected print statements to be turned on and off
 * while the kernel is running.
 */

/* ss debug points */
#define SSDBG_PT1  0x0001           /* interrupt status */
#define SSDBG_PT2  0x0002           /* transmitt flow */
#define SSDBG_PT3  0x0004           /* reciever flow */
#define SSDBG_PT4  0x0008           /* ioctl flow */
#define SSDBG_PT5  0x0010           /* DCD interrupts */
#define SSDBG_PT6  0x0020           /* LP interrupts */
int ss_print;

mblk_t    *ss_getbp();

/* ss streams management definitions */
#define MIN_RMSG_LEN 16             /* minimum buffer size */
#define MAX_RMSG_LEN 2048           /* largest msg allowed */
#define XOFF_RMSG_LEN 256           /* send XOFF here */
#define MAX_RBUF_LEN 1024

#define MAX_RSRV_CNT 3                        /* continue input timer this long */
#define RSRV_DURATION (HZ/30)                 /* send input this often */

#ifdef DEBUG
#        define SSDEBUG(x)  x
#else
#        define SSDEBUG(x)
#endif

/* The null proc */
ss_null() { }

/*
 * Open the line specified by "ssl"
 *
 * Called by:
 * <driver>_open
 */
ss_open (ssl, rq, dev, flag, sflag)
        register struct ss_line       *ssl;     /* line of interest */
        queue_t   *rq;
        dev_t     dev;
        int       flag;
        int       sflag;
{
```

```
	register struct ss_struct	*ssb;
	register int ctlr, ss, line, s, unit;
	queue_t	*wq = WR(rq);

	if (!ssl->ss_lenable) {
		u.u_error = ENXIO;  /* No such device or address */
		return (OPENFAIL);
	}

	ssb = ssl->pss;

	s = spltty();
	if (!(ssl->ss_state & (SS_ISOPEN|SS_WOPEN))) {
		register ushort	cflag;

		cflag = IFLAGS;
		ssl->ss_state &= ~(SS_TXSTOP|SS_LIT|SS_BLOCK|
				SS_TX_TXON|SS_TX_TXOFF|SS_FLOW);

		ssl->ss_litc = CLNEXT;
		ssl->ss_stopc = CSTOP;
		ssl->ss_startc = CSTART;
		ss_cont (ssl, cflag, &def_stty_ld.st_termio);

		/*
		 * Wait for carrier if no dcd once actived,
		 * not local (using modem), and if we can delay.
		 */

		/* Activate device */
		if ((*ssl->pss->ss_devdep->ssdd_act) (ssl))
			ssl->ss_state |= SS_DCD;
		else
			ssl->ss_state &= ~SS_DCD;

		if (!(ssl->ss_cflag & CLOCAL) && !(flag & FNDELAY)) {
			do {
				ssl->ss_state |= SS_WOPEN;
				SSDEBUG((cmn_err(CE_CONT,"ss_open:
waiting\n")));
				if (sleep ((caddr_t)ssl, STIPRI|PCATCH)) {
					u.u_error = EINTR;
					(*ssl->pss->ss_devdep->ssdd_zap)
(ssl);
					ssl->ss_state &= ~SS_WOPEN;
					splx (s);
					return OPENFAIL;
				}
			} while (!(ssl->ss_state & SS_DCD));
		}
```

```
                rq->q_ptr = (caddr_t)ssl;
                wq->q_ptr = (caddr_t)ssl;
                ssl->ss_wq = wq;
                ssl->ss_rq = rq;
                ssl->ss_state |= SS_ISOPEN;
                ssl->ss_cflag |= CREAD;

                if (!strdrv_push (rq, "stty_ld", dev)) {
                        (*ssl->pss->ss_devdep->ssdd_zap) (ssl);
                        ssl->ss_state &= ~(SS_ISOPEN|SS_WOPEN);
                        splx (s);
                        return OPENFAIL;
                }
        }
        else {
                if (ssl->ss_rq == rq) {
                        ASSERT(ssl->ss_wq == wq);
                        ASSERT(ssl->ss_rq->q_ptr == (caddr_t)ssl);
                        ASSERT(ssl->ss_wq->q_ptr == (caddr_t)ssl);
                }
                else {
                        u.u_error = ENOSR;
                        splx (s);
                        return OPENFAIL;
                }
        }

        /* init debug and err counts */
        ssl->ss_ldebug  = 0;
        ssl->ss_overflow = 0;

        splx (s);
        return minor(dev);
}

/*
 * Close the tty associated with this read queue.
 */
ss_close (rq)
        queue_t  *rq;
{
        register struct ss_line     *ssl = (struct ss_line *)rq->q_ptr;
        register int unit, line, state, s;

        if (!ssl)
                return;

        s = spltty();

        ASSERT(ssl->ss_rq == rq);
```

```
        ss_flushr (ssl);
        ss_flushw (ssl);
        ssl->ss_rq = NULL;
        ssl->ss_wq = NULL;
        ssl->ss_state &= ~SS_ISOPEN;

        if (ssl->ss_cflag & HUPCL) {
                (*ssl->pss->ss_devdep->ssdd_zap) (ssl);
        }

        splx(s);
}

/*
 * finish a delay
 */
static
ss_delay (ssl)
register struct ss_line *ssl;
{
        register int s;

        s = spltty();
        if ((ssl->ss_state & (SS_BREAK|SS_BREAK_QUIET)) != SS_BREAK) {
                ssl->ss_state &= ~(SS_TIMEOUT|SS_BREAK|SS_BREAK_QUIET);
                ss_start(ssl);              /* resume output */

        } else {                    /* unless need to quiet break */
                ssl->ss_state |= SS_BREAK_QUIET;
                ssl->ss_tid = timeout(ss_delay, (caddr_t)ssl, HZ/20);
        }
        splx(s);
}

/*
 * Send a 1 or more characters up the stream.
 *      This is more effective than sending characters upstream one at a time.
 */
ss_rsrv (rq)
register queue_t *rq;                   /* our read queue */
{
        register mblk_t *bp;
        register struct ss_line *ssl = (struct ss_line *)rq->q_ptr;
        register int s;

        ASSERT(ssl->ss_rq == rq);
        ASSERT(ssl->ss_state & SS_ISOPEN);

        s = spltty();

        if (!canput(rq->q_next)) {
                /* We will be rescheduled by STREAM (we hope) */
```

```
                ssl->ss_ldebug |= SS_DBG_RSRVSLEEP;   /* mark asleep */
                splx(s);
                return;
        }
        ssl->ss_ldebug &= ~SS_DBG_RSRVSLEEP; /* awoken by STREAMS scheduling */
        ssl->ss_ldebug &= ~SS_DBG_QENBSLEEP; /* Service has run */

        if (0 != (bp = ssl->ss_rbp)) {
                /* add the current input message to the "active message" */
                register int sz;
                sz = (bp->b_wptr - bp->b_rptr);
                if (sz > 0
                  && (!ssl->ss_rsrv_cnt || !ssl->ss_rmsg)) {
                        str_conmsg(&ssl->ss_rmsg, &ssl->ss_rmsge, bp);
                        ssl->ss_rmsg_len += sz;
                        ssl->ss_rbp = 0;
                }
        }

        if (0 != (bp = ssl->ss_rmsg)) {
                /* we have an active message */
                ssl->ss_rmsg = 0;
                ssl->ss_rbsize = ssl->ss_rmsg_len;
                ssl->ss_rmsg_len = 0;
                splx(s);            /* without too much blocking, */
                putnext(rq, bp);    /* send the message */
                (void)spltty();
        }

        if (!ssl->ss_rmsg) {
                if (ssl->ss_state & SS_BLOCK) {         /* do XON */
                        ssl->ss_state |= SS_TX_TXON;
                        ss_start(ssl);
                }
        }

        if (!ssl->ss_rbp) {
                mblk_t *ss_getbp();
                (void)ss_getbp(ssl, BPRI_LO);
        }

        splx(s);
}

/*
 * Slow and hopefully infrequently used function to put characters
 *somewhere where they will go up stream.
 */
ss_slowr (ssl, c)
register struct ss_line *ssl;
```

```
u_char c;                              /* send this byte */
{
        register mblk_t *bp;

        if (ssl->ss_iflag & IBLKMD)  /* this kludge apes the old */
                return;                      /* block mode hack */

        if (! (bp = ssl->ss_rbp)             /* get buffer if have none */
          && ! (bp = ss_getbp (ssl,BPRI_HI))) {
                ssl->ss_allocb_fail++;;
                return;
        }
        *bp->b_wptr = c;
        if (++bp->b_wptr >= bp->b_datap->db_lim) {
                (void) ss_getbp (ssl,BPRI_LO);        /* send buffer when full */
        }
}

/*
 * process carrier-on interrupt
 *
 * Called from:
 * <driver>_cint -- carrier transition interrupt procedure
 */
ss_con (ssl)
register struct ss_line *ssl;
{
        if (ss_print & SSDBG_PT5)
                cmn_err(CE_CONT,"DCD on interrupt\n");
        ssl->ss_state |= SS_DCD;

        if (ssl->ss_state & SS_WOPEN) {
                wakeup((caddr_t)ssl);        /* awaken open() requests */
                ssl->ss_state &= ~SS_WOPEN;
        }
}

/*
 * process carrier-off interrupt
 *
 * Called from:
 * <driver>_cint -- carrier transition interrupt procedure
 */
ss_coff (ssl)
register struct ss_line *ssl;
{
        if (ss_print & SSDBG_PT5)
                cmn_err(CE_CONT,"DCD off interrupt\n");
        ssl->ss_state &= ~SS_DCD;      /* note the change */
```

```
        if (!(ssl->ss_cflag & CLOCAL)) {/* worry about it only for a modem */
                (*ssl->pss->ss_devdep->ssdd_zap) (ssl);
                if (ssl->ss_state & SS_ISOPEN) {
                        flushq(ssl->ss_wq, FLUSHDATA);
                        (void)putctl1(ssl->ss_rq->q_next,
                                  M_FLUSH,FLUSHW);
                        (void)putctl(ssl->ss_rq->q_next, M_HANGUP);
                }
        }
}

/*
 * Do start/stop processing
 * This routine is called from the recieve data interrupt.
 * Returns:
 *  0 => start or stop character. Ignore
 *  1 => valid input character
 *
 * Called from:
 * <driver>_rint - read interrupt procedure
 */
ss_startstop (ssl, c)
        register struct ss_line *ssl;
        char c;
{

        /*
         * Start or stop output (if permitted)
         */
        if (ssl->ss_iflag & IXON) {
                register uchar cs = c & 0x7f;

                if ((ssl->ss_state & SS_TXSTOP)
                && (ssl->ss_startc == cs
                  || ((ssl->ss_iflag & IXANY)
                        && (cs != ssl->ss_stopc
                              || ssl->ss_ttyline == LDISC0)))) {
                              if (ss_print & SSDBG_PT3) {
                                      cmn_err(CE_CONT,"ss_rint:
startc\n");
                              }
                              ssl->ss_state &= ~SS_TXSTOP;

                              /* Call ss to start the line */
                              ss_start (ssl);

                              if (cs == ssl->ss_startc)
                                      return(0);
                } else if (ssl->ss_state & SS_LIT) {
                        if (ss_print & SSDBG_PT3) {
```

```
                                        cmn_err(CE_CONT,"ss_rint: literal char");
                                }
                                ssl->ss_state &= ~SS_LIT;
                        } else if (cs == ssl->ss_stopc) {
                                if (ss_print & SSDBG_PT3) {
                                        cmn_err(CE_CONT,"ss_rint: stopc\n");
                                }
                                ssl->ss_state |= SS_TXSTOP;
                                return(0);
                        } else if (cs == ssl->ss_startc) {
                                if (ss_print & SSDBG_PT3) {
                                        cmn_err(CE_CONT,"ss_rint:ignored
                                                        startc\n");
                                }
                                return(0);          /* ignore extra ^Q's */
                        } else if (cs == ssl->ss_litc
                                && ssl->ss_ttyline == LDISC0) {
                                        if (ss_print & SSDBG_PT3) {
                                                cmn_err(CE_CONT,"ss_rint:literal
                                                                next\n");
                                        }
                                        ssl->ss_state |= SS_LIT;
                        }
                }
                return(1);
}

/*
 * Put a character on the input queue.
 *
 * Called from:
 * <driver>_rint - receive data interrupt.
 */
ss_inc (ssl, c)
                register struct ss_line *ssl;
                char c;
{
                register mblk_t    *bp;
                register int newbuf=0;

                if (ssl->ss_iflag & ISTRIP)
                        c &= 0x7f;
                else
                        c &= 0xff;

                if (!(bp = ssl->ss_rbp)) {
                        if (!(bp = ss_getbp (ssl, BPRI_HI))) {
                                if (ss_print & SSDBG_PT3) {
                                        cmn_err(CE_CONT,"ss_inc:ss_getbp
```

```
                                                          failed\n");
                        }
                        ssl->ss_allocb_fail++;
                        return;
                }
                newbuf++;
        }

        *bp->b_wptr = c;
        if (++bp->b_wptr >= bp->b_datap->db_lim) {
                (void) ss_getbp (ssl, BPRI_LO);
        }

        if (ss_print & SSDBG_PT3) {
                cmn_err(CE_CONT,"ss_inc: c = 0x%x\n", c);
        }

        /* Queue flow contol logic from putq */
        if ((ssl->ss_state & SS_ISOPEN)  && ssl->ss_rq &&
                 canenable(ssl->ss_rq) && (ssl->ss_rq->q_flag & QWANTR)) {
                qenable (ssl->ss_rq);
        } else {
                /* ss_inc did not qenable */
                /* cleared in ss_rsrv */
                ssl->ss_ldebug |= SS_DBG_QENBSLEEP;
        }
}

/*
 * SS streams module, ss_wput (put on write queue).
 *
 * Called from:
 * <streams module upstream>
 */
ss_wput (wq, bp)
        queue_t   *wq;
        register mblk_t     *bp;
{
        register struct ss_line *ssl = (struct ss_line *) wq->q_ptr;
        register struct iocblk *ioss;
        register int s;

        if (!ssl) {
                sdrv_error(wq,bp);          /* quit now if not open */
                return;
        }

        s = spltty();

        ASSERT(ssl->ss_wq == wq);
        ASSERT(ssl->ss_state & SS_ISOPEN);
```

```
        switch (bp->b_datap->db_type) {

        case M_FLUSH:
                if (*bp->b_rptr & FLUSHW) {
                        if (ss_print & SSDBG_PT4) {
                                cmn_err(CE_CONT,"ss_wput: m_flushw\n");
                        }
                        ss_flushw(ssl);
                        ssl->ss_state &= ~SS_TXSTOP;
                        ss_start (ssl);     /* restart output */
                }
                if (*bp->b_rptr & FLUSHR) {
                        if (ss_print & SSDBG_PT4) {
                                cmn_err(CE_CONT,"ss_wput: m_flushr\n");
                        }
                        ss_flushr(ssl);
                }
                sdrv_flush (wq,bp);
                break;

        case M_DATA:
        case M_DELAY:
                if (!(ssl->ss_state & SS_DCD)
                  && !(ssl->ss_cflag & CLOCAL)) {
                        freemsg(bp);        /* discard if !local & !dcd */
                        if (ss_print & (SSDBG_PT2|SSDBG_PT4)) {
                                cmn_err(CE_CONT,"ss_wput: !local & !dcd\n");
                        }
                }
                else {
                        if (ss_print & (SSDBG_PT2|SSDBG_PT4)) {
                                cmn_err(CE_CONT,"ss_wput:
m_data|m_delay\n");
                        }
                        ss_save(ssl, wq, bp);
                }
                break;

        case M_IOCTL:
                ioss = (struct iocblk*)bp->b_rptr;
                if (ss_print & SSDBG_PT4) {
                        cmn_err(CE_CONT,"ss_wput:   m_ioctl   switch   %d   ",
ioss->ioc_cmd);
                }
                switch (ioss->ioc_cmd) {
                case TCXONC:
                        ASSERT(ioss->ioc_count == sizeof(int));
                        if (ss_print & SSDBG_PT4) {
                                cmn_err(CE_CONT,"tcxonc switch = %d",
```

```
                            *(int*)(bp->b_cont->b_rptr));
            }
            switch (*(int*)(bp->b_cont->b_rptr)) {
            case 0:             /* stop output */
                    ssl->ss_state |= SS_TXSTOP;
                    ss_stop(ssl);
                    break;
            case 1:             /* resume output */
                    ssl->ss_state &= ~SS_TXSTOP;
                    ss_start(ssl);
                    break;
            case 2:
                    if (SS_FLOW & ssl->ss_state)
                            ;
                    if (!(ssl->ss_state & SS_BLOCK)) {
                            ssl->ss_state |= SS_TX_TXOFF;
                            ssl->ss_state &= ~SS_TX_TXON;
                    }
                    break;
            case 3:
                    if (ssl->ss_state & SS_BLOCK) {
                            ssl->ss_state |= SS_TX_TXON;
                            ss_start(ssl);
                    }
                    break;
            default:
                    ioss->ioc_error = EINVAL;
                    break;
            }
            bp->b_datap->db_type = M_IOCACK;
            ioss->ioc_count = 0;
            qreply(wq, bp);
            break;

    case TCSETA:
            ASSERT(ioss->ioc_count == sizeof(struct termio));
            (void)ss_tcset (ssl, bp);
            qreply(wq,bp);
            break;

    case TCSETAW:
    case TCSETAF:
            ASSERT(ioss->ioc_count == sizeof(struct termio));
            ss_save (ssl, wq, bp);
            break;

    case TCGETA:
            tcgeta(wq, bp, &ssl->ss_termio);
            break;
```

```
                case TCSBRK:
                        ss_save(ssl, wq, bp);
                        break;

                case FIONREAD:
                        fion(RD(wq),bp, (msgdsize(ssl->ss_rmsg)
                                        +msgdsize(ssl->ss_rbp)));
                        qreply(wq,bp);
                        break;

                case TCBLKMD:
                        ssl->ss_iflag |= IBLKMD;
                        ioss->ioc_count = 0;
                        bp->b_datap->db_type = M_IOCACK;
                        qreply(wq,bp);
                        break;

                default:
                        bp->b_datap->db_type = M_IOCNAK;
                        qreply(wq,bp);
                        break;
                }
                if (ss_print & SSDBG_PT4) {
                        cmn_err(CE_CONT,"\n");
                }

                break;

        default:
                sdrv_error(wq,bp);
        }

        splx(s);
}

/*
 * If the controller is not busy, give it some work to do.
 */
ss_start (ssl)
        struct ss_line     *ssl;
{
        if (!(ssl->ss_state &
          (SS_TIMEOUT|SS_BREAK|SS_BREAK_QUIET|SS_BUSY|SS_TXSTOP))) {
                ss_tx (ssl);
        }
}
```

```
/*
 * Transmit some amount of data on the specified line "ssl".
 */
ss_tx (ssl)
        register struct ss_line     *ssl;
{
        register mblk_t    *wbp, *tbp;
        register char      *p;              /* pointer to info */
        register int       nc;              /* number of char to xfer */
        u_char   c;

        while (1) {
                if ((ssl->ss_state & (SS_TXSTOP|SS_TIMEOUT|SS_ISOPEN))
                   != SS_ISOPEN) {
                        ss_stop (ssl);
                        return;
                }

                if (ssl->ss_state & SS_TX_TXON) {
                        c = ssl->ss_startc;
                        p = &c;
                        nc = 1;
                        ssl->ss_state &= ~(SS_TX_TXON|SS_TX_TXOFF|SS_BLOCK);
                }
                else if (ssl->ss_state & SS_TX_TXOFF) {
                        c = ssl->ss_stopc;
                        p = &c;
                        nc = 1;
                        ssl->ss_state &= ~SS_TX_TXOFF;
                        ssl->ss_state |= SS_BLOCK;
                }
                else {
                        if (!(wbp = ssl->ss_wbp)) {
                                wbp = getq (ssl->ss_wq);
                                if (!wbp) {
                                        ss_stop (ssl);
                                        return;
                                }

                                switch (wbp->b_datap->db_type) {
                                case M_DATA:
                                        break;

                                case M_DELAY:
                                        if (ss_print & SSDBG_PT4) {
                                                cmn_err(CE_CONT,"ss_tx:
                                                        m_delay\n");
                                        }
                                        ssl->ss_state |= SS_TIMEOUT;
```

```
                                        ssl->ss_tid = timeout (ss_delay,
                                (caddr_t)ssl, *(int *)wbp->b_rptr);
                                        freemsg (wbp);
                                        continue;

                                case M_IOCTL:
                                        if (ss_print & SSDBG_PT4) {
                                                cmn_err(CE_CONT,"ss_tx:
                                                        m_ioctl\n");
                                        }
                                        ss_i_ioctl (ssl, wbp);
                                        continue;

                                default:
                                        cmn_err (CE_PANIC, "bad isi_mux
                                                        msg");
                                        break;
                                }
                        }

                        if (wbp->b_rptr >= wbp->b_wptr) {
                                ASSERT(wbp->b_datap->db_type == M_DATA);
                                ssl->ss_wbp = rmvb (wbp, wbp);
                                freeb (wbp);
                                continue;
                        }
                        ssl->ss_wbp = wbp;
                        p = wbp->b_rptr;
                        nc = MIN((int) p & POFFMASK,
                                wbp->b_wptr - wbp->b_rptr);
                }

                if (nc) {
                        nc = MIN(ssl->pss->ss_devdep->ssdd_maxoutc, nc);
                        (*ssl->pss->ss_devdep->ssdd_outc)(ssl, p, nc);
                        break;
                }
        }
}
```

```
/*
 * interrupt-process an IOCTL
 *        This function processes those IOCTLs that must be done by the output
 *        interrupt.
 */
static
ss_i_ioctl (ssl,bp)
register struct ss_line *ssl;
register mblk_t *bp;
{
        register struct iocblk *ioss;

        ioss = (struct iocblk*)bp->b_rptr;

        if (TCSBRK == ioss->ioc_cmd) {
                if (0 == *(int*)bp->b_cont->b_rptr) {
                        ssl->ss_state |= (SS_TIMEOUT|SS_BREAK);
                        ssl->ss_tid = timeout(ss_delay, (caddr_t)ssl, HZ/4);
                }
                ioss->ioc_count = 0;
                bp->b_datap->db_type = M_IOCACK;

        } else if (ss_tcset (ssl,bp)
                  && TCSETAF == ioss->ioc_cmd) {
                (void)putctl1(ssl->ss_rq->q_next, M_FLUSH, FLUSHR);
        }

        putnext (ssl->ss_rq, bp);
}

/*
 * Set parameters from open or stty. Call the device dependent procedure
 * "ssdd_setline" to do the work.
 */
ss_cont (ssl, cflag, tp)
        register struct ss_line *ssl;
        int cflag;
        struct termio *tp;
{
        register uint diff;
        register int s;

        /*
         * Block interrupts so parameters will be set before line interrupts.
         */
        s = spltty();
        diff = cflag ^ ssl->ss_cflag;
        if (diff & (CBAUD|CSIZE|CSTOPB|PARENB|PARODD)) {
                if ((cflag & CBAUD) == 0)
                        (*ssl->pss->ss_devdep->ssdd_zap) (ssl);
```

```
                else
                        (*ssl->pss->ss_devdep->ssdd_setline)  (ssl,  cflag,
tp);
        }

        if (tp) {
                ssl->ss_termio = *tp;
        }
        ssl->ss_cflag = cflag;

        splx(s);
}

/*
 * Get a new buffer
 * Interrupts ought to be off here.
 *
 * Called from:
 * <driver>_rint - receive data interrupt procedure.
 *   ss_inc -- an data to input queue.
 */
mblk_t *                                /* return NULL or the new buffer */
ss_getbp (ssl, pri)
register struct ss_line *ssl;
uint pri;                               /* BPRI_HI=try hard to get buffer */
{
        register int size;
        register mblk_t *bp;
        register mblk_t *rbp;

        rbp = ssl->ss_rbp;
        if (ssl->ss_rmsg_len >= MAX_RMSG_LEN  /* if overflowing */
          || (0 != rbp                  /* or current buffer empty */
                && rbp->b_rptr >= rbp->b_wptr)) {
                bp = 0;
        } else {
                size = ssl->ss_rbsize;
                if (size > MAX_RBUF_LEN)      /* larger buffer */
                        size = MAX_RBUF_LEN;          /* as we get behind */
                for (;;) {
                        if (size < MIN_RMSG_LEN)
                                size = MIN_RMSG_LEN;

                        bp = allocb(size, pri);
                        if (0 != bp)
                                break;

                        if (BPRI_HI == pri
                          && size > MIN_RMSG_LEN) {
                                size >>= 2;
```

```
                                        continue;
                                }
                                break;
                        }
                }

                if (0 == rbp) {         /* if we have an old buffer */
                        ssl->ss_rbp = bp;
                } else if (0 != bp              /* & a new buffer */
                           || (rbp->b_wptr   /* or old buffer is full */
                             >= rbp->b_datap->db_lim)) {
                        str_conmsg(&ssl->ss_rmsg, &ssl->ss_rmsge, rbp);
                        ssl->ss_rmsg_len += (rbp->b_wptr - rbp->b_rptr);
                        ssl->ss_rbp = bp;
                }

                if (ssl->ss_rmsg_len >= XOFF_RMSG_LEN
                    || !ssl->ss_rbp) {
                        if ((ssl->ss_iflag & IXOFF)   /* do XOFF */
                          && !(ssl->ss_state & SS_BLOCK)) {
                                ssl->ss_state |= SS_TX_TXOFF;
                                ssl->ss_state &= ~SS_TX_TXON;
                                ss_start(ssl);
                        }
                }

                return bp;
}

/*
 * Set line parameters.
 * Call ss_cont, which calles "ssdd_setline", to do the work.
 */
static int /* 0=bad IOCTL */
ss_tcset (ssl, bp)
register struct ss_line *ssl;
register mblk_t *bp;
{
                register struct iocblk *ioss;
                register struct termio *tp;
                register uint cflag;
                register int baud;

                ioss = (struct iocblk*)bp->b_rptr;
                tp = STERMIO(bp);

                cflag = tp->c_cflag;
                baud = (cflag & CBAUD);

                ss_cont(ssl, cflag, tp);
                tp->c_cflag = ssl->ss_cflag; /* tell line discipline the results */
```

```
        ioss->ioc_count = 0;
        bp->b_datap->db_type = M_IOCACK;
        return 1;
}

/*
 * Flush input
 * interrupts must be safe here
 */
static
ss_flushr (ssl)
register struct ss_line *ssl;
{
        freemsg(ssl->ss_rmsg);
        ssl->ss_rmsg = NULL;
        ssl->ss_rmsg_len = 0;
        freemsg(ssl->ss_rbp);
        ssl->ss_rbp = NULL;

        qenable(ssl->ss_rq);                    /* turn input back on */
}

/*
 * flush output
 * Interrupts must have been made safe here.
 */
static
ss_flushw (ssl)
register struct ss_line *ssl;
{
        if ((ssl->ss_state & SS_TIMEOUT) == SS_TIMEOUT) {
                untimeout (ssl->ss_tid);    /* forget stray timeout */
                ssl->ss_state &= ~SS_TIMEOUT;
        }

        freemsg(ssl->ss_wbp);
        ssl->ss_wbp = NULL;
}

/*
 * save a message on our write queue,
 *and start the output interrupt, if necessary
 *
 *We must be safe from interrupts here.
 */
static
ss_save (ssl, wq, bp)
register struct ss_line *ssl;
queue_t *wq;
mblk_t *bp;
```

```
{
        putq(wq,bp);                        /* save the message */

        /*
         * ss_start will do the necessary checking to see if calling ss_tx
         * is really necessary.
         */
        ss_start(ssl);
}

ss_stop ()
{
}
```

# Appendix F
# Standalone Programs

## Introduction

A standalone program is independent from the operating system, and can be run and used without the operating system. This Appendix describes the following standalone programs:

- Format
- Standalone Shell (sash)

These standalone programs were not included in PROM because of space limitations. In addition, the sash program is frequently updated with additional device drivers and file system types. Both programs are run from the PROM Monitor command prompt using the Boot command.

The standalone Format program is a utility that allows you to initialize the disk drive, format the drive, and write the volume header.

The sash program is the MIPS Standalone Shell, and it is an extended version of the PROM Monitor. The sash program consists of all the PROM Monitor commands and additional device drivers and file system types. Sash also includes additional commands that are not available in the PROM Monitor.

Table F.1 lists the basic editing commands available for the Format program and for sash.

**Table F.1. Basic Editing Commands**

| Command | Description |
|---|---|
| **Control–H or DEL** | Erases the previous character. |
| **Control–U** | Erases the entire line. |
| **Control–C** | Aborts the program that is currently running and returns control to the PROM Monitor. |
| **Control–Z** | Causes the current program to execute a breakpoint instruction. This command is used in conjunction with the standalone program dbgmon. |
| **Control–D** | Causes the standalone program to exit normally. |

# Format

**CAUTION**

**SCSI disks are formatted at the factory and do not need to be formatted. The disk format that is performed by the factory is more rigorous and finds more defects than the following Format program is capable of detecting. Therefore, it is recommended that the M/120 SCSI disks are not formatted unless it is believed that there is something physically wrong with the disk.**

This section describes the standalone Format program and how it works for SCSI disk drives. The Format program can be used to modify a disk partition table, initialize the volume header, or to examine the volume header without formatting a disk.

The standalone version of Format is booted using the PROM Monitor **boot** command. The Format program can be booted from a cartridge tape, from a hard disk if the software has already been installed, or from the network. To boot the Format program from the network, a machine must be running the bootfile Server Daemon **bfsd(8)**.

To load the Format program from the cartridge tape containing the released software, type:

**boot –f tqis(,,2)format**

To load the Format program from SCSI disk, type:

**boot dkis()/stand/format**

To load the Format program from the network, type:

**boot –f bfs()/stand/format**

The parenthesis in the commands shown above indicate that the previous argument is a device. When booting over the network, if the command is entered as shown, then it will boot the Format program from the first machine that is found that has the program. You can also boot the format program from a specific machine by specifying the machine name and a path name as shown in the following example.

**boot –f bfs()machinename:/stand/format**

## Description

After the Format program has been loaded and is running, a series of questions is displayed on the screen. Some of the questions require a yes or a no answer, and some of the questions require numeric or typed–word answers. For questions that require a yes or a no answer, the program interprets any character other than *y* to be *no*.

The following pages contain actual screen output from the Format program along with a brief explanation. Refer to the *System Administrator's Guide* for examples on how to use this program. The questions that are displayed on the screen by the program appear one at a time. In the following examples, the screen output shows related and sequential questions grouped together. In the following examples of screen output, italic print indicates a variable.

When you first enter the Format program, the following program information and questions are asked.

```
MIPS Format Utility
Version 4.0 Thu June 16 08:42:14 PDT 1988 root

name of device?
LUN number?
target id?
```

If you enter a device name that the program does not recognize, then after you have entered a LUN number and target id number the program displays an error message, lists the known devices, and redisplays the "name of device" question. The known devices are shown below.

```
tty:        console uart
console:    pseudo console
dkis:       SCSI disk
bfs:        boot server/LANCE ethernet
tqis:       SCSI tape
mem:        memory pseudo–device
```

The device name for the SCSI disks is dkis. The LUN number must be zero, and the target id number is the SCSI device number. In the M/120, the SCSI devices are assigned as given in Table F.2.

**Table F.2. SCSI Device Assignments**

| Device # | Peripheral Device | Device Priority |
|---|---|---|
| 7 | Initiator, located on Motherboard | Highest |
| 6 | Tape Drive, Main Cabinet | |
| 5 | Disk in slot 5, Expansion Cabinet | |
| 4 | Disk in slot 4, Expansion Cabinet | |
| 3 | Disk in slot 3, Expansion Cabinet | |
| 2 | Disk in slot 2, Expansion Cabinet | |
| 1 | Disk in slot 1, Expansion Cabinet | |
| 0 | Disk Drive, Main Cabinet | Lowest |

After you have provided valid input for the first three questions, one of two things will happen. First, if the volume header is valid then the question shown below is displayed on the screen.

```
choose new drive parameters (y if yes)?
```

It is recommended that you answer this question with a no, unless you know that your volume header contains incorrect information. If you answer the question shown above with a yes, then the following information is displayed on the screen.

```
device parameters are known for:
            (9) fuji 2246sa (140Meg SCSI)
            (10) cdc 94161 (160Meg SCSI)
            (11) cdc 94171 (328Meg SCSI)
            (12) fuji 2249sa (325Meg SCSI)
enter number for one of the above?
```

Second, if the volume header is not valid, or if the disk is new and has never been formatted, then screen display shown above is immediately displayed on the screen, and you are asked to enter the number of the device.

In the display shown above, items 1–8 are not shown because they are for the SMD disk drives which are not used in the M/120. After selecting one of the disk drives from the displayed list, the following question is displayed on the screen.

```
The UNIX file system partitions may be either BSD or System V
do you desire BSD file system partitions (y if yes)
```

MIPS no longer supports System V file systems. Therefore, this question must be answered with a yes. The following question is displayed on the screen.

```
dump device parameters (y if yes)?
```

If you answer this question with a yes, then the following information is displayed on the screen. The device parameters for your disk will be displayed in place of the *number* in the following example.

```
            number cylinders = number
            number heads = number
            number sectors per track = number
            number bytes per sector = number
            sector interleave = number
modify device parameters (y if yes)?
```

If you answer the dump device parameters question with a no (N), then the following question appears on the screen. This is the same question that appears if you had indicated that you wanted to dump the device parameters first.

```
modify device parameters (y if yes)?
```

It is recommended that these device parameters should never be modified. If you think you need to change these parameters, then contact MIPS customer support first. When you answer no to the question shown above, the following question is displayed on the screen.

```
dump partition table (y if yes) ?
```

If you answer this question with a yes, then the following partition table information is displayed on the screen. The partition table for your disk will be displayed in place of the note below. After the table is displayed, you are asked if you want to modify the partition table. If you had answered no to the dump partition table question, then this same question (modify partition table) appears on the screen. In the following example, variables are indicated with italic printing.

Root partition is entry # *number*
Swap partition is entry # *number*
Default boot file is /vmunix

*Partition Table appears here*

modify partition table (y if yes) ?

If you choose to modify the partition table, then the following is displayed on the screen.

```
partition table manipulation
choose one of  (list, add, delete, quit, init, modify, replace)
command?
```

Depending on which item you select, the screen display is different. If the modify partition table question had been answered with a no, then the following message appears on the screen. This same message appears after you have modified the partition table, and exited the loop by pressing the **Enter** key or entering a no (N) answer to both the dump and modify questions.

```
formatting destroys ALL SCSI disk data, perform format (y if yes)?
```

If you answer yes to this question, then the first message shown below appears on the screen while the disk is being formatted. Formatting takes a while. When the disk has been formatted, then the second message shown below appears on the screen.

```
formatting
scanning destroys disk data, perform scan (y if yes)?
```

If you answer yes to the above question, then you are asked how many times you want to scan for bad blocks. Scanning is recommended after formatting because it is the scanning phase that detects errors on the disk. It is suggested that you scan three times.

number of scans for bad blocks (3 are suggested)?
starting cylinder is 0, ending cylinder is *number*

scanning for defects, pass 1 (*1 dot is printed for each cylinder that is checked*)

scanning for defects, pass 2 ...

scanning for defects, pass 3...

*continues for the number of passes you specified.*

If an error is found while scanning, the following error message is displayed, which indicates the cylinder number and track number of the error. After the error has been recorded, the dot printing is resumed for each cylinder that passes.

Error on cyl *number*, track *number*

If you did not want to format the SCSI disk and entered a no answer, then the following question appears on the screen.

formatting wasn't done, perform scan anyway (y if yes)?

If you answered no to the question above, or if you formatted the disk and finished the scanning, then the following messages appear on the screen.

SCSI defect list manipulation, when prompted,
choose one of (list, add, delete, quit)
command?

The **list** operation lists or displays the defect list on the screen. The **add** operation allows you to add a known defect to the list, and the **delete** operation allows you to remove a defect from the list. Selecting **quit**, exits the scanning phase, and displays the following question on the screen.

write new volume header? (y if yes)?

Entering a yes answer writes the volume header to disk and then exits the formatting program. If you enter a no answer, then the Format program is exited, and any changes you made to the device parameters or to the partition table are not saved.

## Additional Information

An example format session is contained in the section entitled **Disk Management Procedures** in the *Systems Administrator's Guide*. Also, additional information on how to create a volume header, can be found in the software installation instructions in the *Release Notes*.

# Standalone Shell (sash)

Sash is the MIPS standalone shell. The standalone shell is an extended version of the PROM Monitor that includes all the PROM Monitor commands. In addition to the PROM Monitor commands, sash includes additional commands and is configured with more device drivers and file system types. Sash exists so that the MIPS standalone programs and the PROM Monitor are not dependent on the operating system.

When the sash program requires a file name, the file name is constructed in different ways depending on the device. The different file name formats are shown below, and Table F.3 describes the different parts of the file names.

| | |
|---|---|
| **SCSI disk** | **dkis (LUN, target, partition) path** |
| **SCSI tape** | **tqis (LUN, target, partition) path** |
| **Console uart** | **tty (port #)** |
| **Pseudo console** | **console (port #)** |
| **Boot server** | **bfs ()** |

**Table F.3. File Name Syntax**

| File name Part | Description |
|---|---|
| **LUN** | Logical Unit Number. Each SCSI target can have up to 8 (0–7) logical units, but MIPS currently uses only embedded SCSI devices, which only support LUN 0 (zero). |
| **target** | The target number indicates the embedded SCSI device from 0–5, with 0 being the main cabinet device. Targets 1–5 are the SCSI devices in the Expansion Cabinet. The tape drive is hard–wired to device 6, but it is simply entered as 0 (zero) for the tqis device. If you do not specify a unit number, the default value of 0 is used. |
| **partition** | Disk devices are frequently broken down into logical subunits, called partitions. The partition field selects a disk partition within a unit. The partition's base cylinder and size is determined by accessing the disk volume header stored on the disk itself. If you do not specify the partition field, the default value of 0 is used. For Tape devices, this field specifies the number of the file on the tape. Files are numbered on the tape starting with zero. |
| **path** | The path indicates a particular file on the media specified by the device, controller, unit, and partition fields. The file referred to by path is located by consulting a directory located on the device itself. If you do not specify a path, the file name is assumed to refer to the raw device. |
| **port #** | The port # field indicates the serial I/O port number. This number can be either 0 (zero) or 1 (one). |

The sash program is booted using the PROM Monitor Boot command. The sash program can be booted from a cartridge tape, from a hard disk if the software has already been installed, or from the network. To boot the sash program from the network, a machine must be running the bootfile Server Daemon **bfsd(8)**.

To load the sash program from a cartridge tape, type:

**boot –f tqis()sash [ –a ] [ –r ] [ file [ args ]]**

To load the sash program from SCSI disk, type:

**boot dkis()/stand/sash [ –a ] [ –r ] [ file [ args ]]**

To load the sash program from the network, type:

**boot –f bfs()sash [ –a ] [ –r ] [ file [ args ]]**

The parenthesis in the commands shown above indicate that the previous argument is a device. When booting over the network, if the command is entered as shown, then it will boot sash from the first machine that is found that has the program. You can also boot sash from a specific machine by specifying the machine name and path name.

If sash is booted without arguments, then the sash command mode is entered. The sash command prompt is shown below.

**sash:**

If the **–a** argument is used as the first argument, then sash assumes that an *automatic* operating system boot is to be done. Sash examines the name by which it was booted and uses the same device, controller, and unit to look for an operating system to boot. Sash finds the correct operating system file to boot by examining the disk volume header on the specified device. The volume header specifies a root partition and an operating system file name. Once the appropriate operating system file is determined, sash boots the operating system and passes the **–a** argument and any other arguments following the **–a** to the operating system.

If the **–r** argument is specified as the first argument, then sash assumes that the next argument is the standalone program that is being booted by a remote debugger. Sash defines the environment variables "dbgmon" and "rdebug," boots the file specified by the argument after the **–r** flag, and passes any succeeding arguments. If the booted program was linked against the standalone library, then the start–up code provided will note the environment variables "dbgmon" and "rdebug" and load the debugging monitor co–resident with the program. This causes the program to enter the remote debugging mode.

If any other argument is passed to sash when it is booted, then sash interprets the argument as the file name of a program to be booted immediately. Any other arguments appearing on the command line to call sash will be passed through to the booted program. Therefore, if the PROM Monitor environment variable **bootfile** is set as "sash" and the command listed below is entered on the PROM Monitor command line, then the PROM Monitor loads the file indicated by the environment variable **bootfile**. The bootfile contains the sash program.

```
boot dkis()unix
```

## Extending the Standalone Shell

If you type a sash command on the sash command line that is not built–in, then sash uses the first word of the command as the name of a file. Sash then tries to boot that file by passing any other arguments on the command line to the booted program. This mechanism makes two–level boots possible.

If the environment variable **path** is not defined, then the first word of the command must be a complete file name specification consisting of a device name, controller, unit, partition, and a file path. If the environment variable **path** is defined, then the standalone shell attempts to boot the program file formed by prepending the contents of **path** to the original file name. If **path** is a list of prefixes separated by spaces, then the standalone shell will try each prefix from **path**, until the file is successfully booted or until all prefixes have been tried.

## Sash Commands

When sash is booted without arguments, the sash command mode is entered. From the command mode prompt, memory and environment variables can be displayed and altered, and other programs can be booted. The commands that can be used from the sash command prompt are listed in Table F.4 and Table F.5. The commands listed in Table F.4 are the PROM Monitor commands and a complete description of each command can be found in **Chapter 5, PROM Monitor**. Table F.5 lists the commands that are included in sash, but are not part of the PROM Monitor. A description for each sash command is given in the following pages.

### Table F.4. PROM Monitor Commands

| Command | Description |
|---|---|
| **auto** | Initiates the two–level operating system autoboot sequence. |
| **boot** | Loads the specified program. |
| **cat** | Displays the contents of the files listed on the console. |
| **disable** | Does not allow input from and output to the specified console device. |
| **dump** | Formats and displays the contents of memory. |
| **enable** | Allows input from and output to the specified console device. |
| **fill** | Fills the specified range of memory with the specified pattern. |
| **g** | Displays the contents of a single memory location in decimal, hexadecimal, and ASCII character formats. |
| **go** | Transfers control to code that is assumed to have been previously loaded. |
| **help** | Displays the syntax for all commands. |
| **init** | Reinitializes the PROM Monitor software state. |
| **init_tod** | Initializes the time–of–day chip. |
| **load** | Allows you to load memory over a serial line connection. |
| **p** | Puts or sets the contents of a single memory location to a specified value. |
| **printenv** | Displays the value of the PROM environment variables. |
| **pr_tod** | Prints the contents of the time–of–day register. |
| **setenv** | Used to create a new environment variable or to change an existing environment variable. |
| **sload** | Accepts a subset of the Motorola S–record protocol. |
| **spin** | Generates reference patterns for diagnostic use. |
| **unsetenv** | Used to delete an existing environment variable. |
| **warm** | Examines memory for a restart block. |

### Table F.5. Sash Commands

| Command | Description |
|---|---|
| **cp** | Copies the contents of one file to another file. |

# cp

## Synopsis

**cp** [–b blocksize] [–c count] file1 file2

## Description

The **cp** command copies the contents of **file1** to **file2**. The **–b** option specifies a blocksize for the transfer. If **–b** is not specified, then the blocksize defaults to 512 bytes. For raw devices, the blocksize should be an integral multiple of the device's physical record length. The **–c** option specifies a maximum byte count to be transferred. If you do not specify **–c**, the copy terminates at EOF on **file1**. While the **cp** command is being executed, a period ( . ) is printed on the screen for each record that is transferred.

## Example

The following example copies file 3 (the fourth file, zero is the first file) on the cartridge tape to partition 1 of the disk.

```
sash:cp -b 16k tqis(,,3) dkis(,,1)
```

# Index

## Numbers

8254 counter/timer, 3–18
  registers, 3–19

## A

access type, Fault ID Register, 3–9

adding drivers, 4–4
  compiling the driver, 4–5
  configuration files, 4–6
    kernel file, 4–6
    sysgen file, 4–6
  environment variable, 4–4
  master file, 4–5

addressing, 3–2—3–3

AT bus, 3–26—3–30, 4–7
  address space, 4–8
  AT Control Register, 3–28
  ATDAckEn Register, 3–30
  ATReqEn bit, 3–16
  byte swapping, 3–27
  control registers, 3–28
  DAckEnB bit, 3–16
  DMA flow control, 3–16
  FlowThruMode bit, 3–16
  interface, 3–26
  memory access, 3–26
  memory mapping, 3–26
  pin/signal assignments, A–11
  resetting, 3–13
  slots, 1–6

AT bus card slots, location of, 2–9

AT card slots, installing, 2–9

AT cards, location of, 2–9

AT Control register
  ATReqEn bit, 3–29
  DAckEnB bit, 3–29
  FlowThruMode bit, 3–29
  FlowToMbus bit, 3–29
  IAddr bits, 3–30

ATDackEn Register, AT bus, 3–30

ATReqEn bit
  AT control, 3–29
  DMA controller, 3–16

ATTCEn bit
  DMA controller, 3–16
  System Configuration Register, 3–13

auto, PROM Monitor command, 5–6, 5–7

## B

baud, environment variable, 5–3

baud rate, changing with BREAKs, 5–5

big–endian, 3–2

bit naming conventions, 3–2

block diagram, 3–1

boot, PROM Monitor command, 5–6, 5–8

bootfile, environment variable, 5–3

BootLockB bit, System Configuration Register, 3–13

bootmode, environment variable, 5–3

BREAKs, changing baud rate with, 5–5

building the kernel, 4–7

bus, AT bus, 3–26

byte count pointer, SCSI, 3–12

byte swapping, AT bus, 3–27

## C

c8.c driver program, E–1

cards, installing additional, 2–7

cartridge tape
  operation and maintenance, B–1—B–4
  write protection, B–2

cat, PROM Monitor command, 5–6, 5–9

CE_CONT, 4–16

CE_NOTE, 4–16

CE_PANIC, 4–16

CE_WARN, 4–16

clock. *See* real–time clock

cmn_err(), 4–15

ColdStart bit, System Configuration Register, 3–13

command set, PROM Monitor, 5–6

configuration. *See* System Configuration Register

configuration files, specifying address space, 4–6

connecting
  console device, 2–13
  serial I/I devices, 2–13

connectors
  Ethernet, 2–15
  SIO, 3–23

console, environment variable, 5–3

console device, connecting, 2–13

control register, AT bus, 3–28

control registers, AT bus, 3–28

controller, PROM Monitor, 5–2

controls, 1–7
  lock, 1–7
  reset, 1–7
  unlock, 1–7

CoProcB bit, System Configuration Register, 3–13

copy (cp) sash command, F–12

counter/timer, 3–18
  interrupt acknowledge, 3–19
  register summary, 3–19

cp (copy) sash command, F–12

CPU, central processing unit, 1–3

cpuid, environment variable, 5–4

## D

DAckEnB bit
  AT control, 3–29
  DMA controller, 3–16

data formats, 3–2—3–3

debugging drivers, 4–15
  error messages, 4–15
  halting the system, 4–15

Demand Dedicated with Bus Release, DMA operating mode, 3–15

device, PROM Monitor, 5–2

device assignments, SCSI, C–1

device driver, sample listing, E–1

device drivers, 4–1—4–16

devices
  dkis(), F–3, F–9
  tpqic(), F–3, F–9

diagnostics, power on, D–1—D–24

dimensions, 2–2

direct memory access. *See* DMA

disable, PROM Monitor command, 5–6, 5–10

disk drives, installing in expansion cabinet, C–1

DMA, 3–14—3–16
  slow registers, 3–13
  SoftEOP bit, 3–13
  TC enable bit, 3–13

DMA controller
  ATReqEn bit, 3–16
  ATTCEn bit, 3–16
  DAckEnB bit, 3–16
  FlowThruMode bit, 3–16
  FlowToMbus bit, 3–16
  interface registers, 3–15
  operating modes, 3–15
  SlowUDCEn bit, 3–16
  SoftEOP bit, 3–16
  software control, 3–16

DMA operating mode
  Demand Dedicated with Bus Release, 3–15
  Flowthru mode, 3–15
  Flyby mode, 3–15

DUART, 3–21—3–23
  registers, 3–21

dump, PROM Monitor command, 5–6, 5–11—5–13

## E

enable, PROM Monitor command, 5–6, 5–13

environment variables, PROM Monitor, 5–3

environmental requirements, 2–4

EOP bit
  DMA, 3–13
  DMA controller, 3–16

error, parity, 3–9

Ethernet interface. *See* Lance

Ethernet port, connector, 2–15

expansion cabinet, 1–8, C–1—C–10

extending PROM Monitor, 5–5

# F


FAR, *See* Fault Address Register

fault. *See* Fault Address Register; Fault ID Register

Fault Address Register, 3–7, 3–11

fault handling, 3–8

Fault ID Register, 3–7, 3–9—3–11
 access type, 3–9
 IBusMast bits, 3–10
 IBusValidB bit, 3–11
 MReadQ bit, 3–10
 OldAccType, 3–9
 parity error, 3–9
 ProcBd bit, 3–10
 TimeOut bit, 3–10

FID, *See* Fault ID Register

file name syntax, prom monitor, 5–2

file structure, 4–1
 bootarea directory, 4–3
 io directory, 4–2
 kernel subset, 4–1
 master.d directory, 4–2

fill, PROM Monitor command, 5–6

fill command, PROM Monitor command, 5–14

Flowthru, DMA operating mode, 3–15

FlowThruMode bit
 AT control, 3–29
 DMA controller, 3–16

FlowToMbus bit
 AT control, 3–29
 DMA controller, 3–16

Flyby, DMA operating mode, 3–15

ForceBadPar bit, System Configuration Register, 3–14

format program, 4–14, F–3—F–8

FPA
 *See also* Floating Point Coprocessor
 floating point accelertor, 1–5

fuse holder, 2–5


# G


g, PROM Monitor command, 5–6

g (get), PROM Monitor command, 5–15

go, PROM Monitor command, 5–6, 5–16


# H


help, PROM Monitor command, 5–6, 5–17


# I


IAddr bits, AT control, 3–30

IBusMast bits, Fault ID Register, 3–10

IBusValidB bit, Fault ID Register, 3–11

ID PROM, 3–17

IMR. *See* Interrupt Mask Register

indicators, 1–7

init, PROM Monitor command, 5–6, 5–18

init_tod, PROM Monitor command, 5–6, 5–19

input editing, PROM monitor, 5–4

installation, 2–1—2–16

installing additional cards, 2–7

installing AT cards, 2–9

installing memory cards, 2–8

interface
 Eterhnet, 3–25—3–26
 SCSI, 3–23—3–25

interrupt, level0, 3–6

interrupt acknowledge, counter/timer, 3–19

Interrupt Mask Register, 3–8

interrupt priority, 4–10
 changing levels of, 4–11

Interrupt Status Register, 3–7

interrupt system. *See* interrupts

Interrupts, level–0, 3–7—3–11

interrupts, 3–6
 level 0, 3–6
 level 1, 3–6
 level 2, 3–7
 level 3, 3–7
 level 4, 3–7
 level 5, 3–7

interval timer, 3–18

I/O devices, connecting, 2–11

I/O subsystems, 3–18—3–25

ISR. *See* Interrupt Status Register


# K


kernel
 building of, 4–7

support routines, 4–9
address translation, 4–9
delay(n) macro, 4–9

kernel file, 4–6

kernel subset, file structure of, 4–1

Key0 bit, System Configuration Register, 3–12

keyswitch, 1–7

## L

Lance
Local Area Network Controller for Ethernet. *See* Ethernet
registers, 3–25

LEDs, pon diagnostic patterns, D–2—D–24

LED register, 3–17

LEDs, 1–8

level 0 interrupt, 3–6

level 0 interupt, 3–7—3–11

level 1 interrupt, 3–6

level 2 interrupt, 3–7

level 3 interrupt, 3–7

level 4 interrupt, 3–7

level 5 interrupt, 3–7

load, PROM Monitor command, 5–6, 5–20

lock. *See* BootLock bit

lock keyswitch, 1–7

LUNS, 4–14

## M

map, memory. *See* memory map

mask register, interrupts, 3–8

memory, 1–5

memory access, 3–14
AT bus, 3–26

memory cards
installing, 2–8
location of, 2–8

memory faults, 3–8

memory management, 4–12

memory map, 3–3—3–5

memory mapping, AT bus, 3–26

memory usage, PROM Monitor, 5–1

memparity, environment variable, 5–4

modem adapter, connections for, 2–12

monitor. *See* PROM Monitor

motherboard, 1–3

MReadQ bit, Fault ID Register, 3–10

## N

naming conventions, 3–2

netaddr, environment variable, 5–3

non–volatile RAM
*See also* NVRAM
PROM monitor, 5–5

NVRAM, register summary, 3–20

## O

OldAccType, Fault ID Register, 3–9

operating modes, DMA controller, 3–15

operation details, SCSI, 3–25

optimizing compilers, 4–12

## P

p, PROM Monitor command, 5–6

p (put), PROM Monitor command, 5–21

parity error, Fault ID Register, 3–9

ParityEn bit, System Configuration Register, 3–14

partition, PROM Monitor, 5–2

path, PROM Monitor, 5–2

PC/AT bus. *See* AT bus

peripherals, 1–6
QIC tape drive, 1–7

pinouts
Ethernet port, 2–15
serial ports, 2–12

Pointer bits, System Configuration Register, 3–12

pon diagnostics, D–1—D–24

power on diagnostics, D–1—D–24

power requirements, 2–3

power up procedure, 2–16

pr_tod, PROM Monitor command, 5–6, 5–23

printenv, PROM Monitor command, 5–6, 5–22

ProcBd bit, Fault ID Register, 3–10

programming model, 3–1

PROM, ID, 3–17

PROM Monitor, 5–1—5–28
  environment variables, 5–3
    baud, 5–3
    bootfile, 5–3
    bootmode, 5–3
    console, 5–3
    cpuid, 5–4
    memparity, 5–4
    netaddr, 5–3
    rbaud, 5–3
    resetepc, 5–4
    resetra, 5–4
    version, 5–4
  file name syntax, 5–2
  input editing, 5–4
  memory usage, 5–1
  non–volatile RAM, 5–5
  time of day, 5–5

PROM monitor
  auto command, 5–7
  boot command, 5–8
  cat command, 5–9
  command set, 5–6
    auto, 5–6
    boot, 5–6
    cat, 5–6
    disable, 5–6
    dump, 5–6
    enable, 5–6
    fill, 5–6
    g, 5–6
    go, 5–6
    help, 5–6
    init, 5–6
    init_tod, 5–6
    load, 5–6
    p, 5–6
    pr_tod, 5–6
    printenv, 5–6
    setenv, 5–6
    sload, 5–6
    spin, 5–6
    unsetenv, 5–6
    warm, 5–6
  disable command, 5–10
  dump command, 5–11—5–13
  enable command, 5–13
  extending, 5–5
  fill command, 5–14
  g (get) command, 5–15
  go command, 5–16
  help command, 5–17
  init command, 5–18
  init_tod command, 5–19
  load command, 5–20
  p (put) command, 5–21
  pr_tod command, 5–23
  printenv command, 5–22
  setenv command, 5–24
  sload command, 5–25
  spin command, 5–26
  unsetenv command, 5–27
  warm command, 5–28

putbuf[], 4–16

# R

R2450 Memory Card, 1–5

R2450 memory cards, location of, 2–8

rbaud, environment variable, 5–3

real–time clock, register summary, 3–20

reconfiguration
  AT&T's reconfiguration process, 4–3
  MIPS' reconfiguration process, 4–3

register
  DUART, 3–21
  Fault Address, 3–7, 3–11
  Fault ID, 3–7, 3–9—3–11
  Interrupt Mask, 3–8
  Interrupt Status, 3–7
  LED, 3–17
  System Configuration, 3–12—3–14

registers
  AT bus, 3–28
  AT Control, 3–28
  ATDackEn, 3–30
  counter/timer, 3–19
  counter/timer interrupt acknowledge, 3–19
  DMA controller, 3–15
  Lance, 3–25
  real–time clock, 3–20
  SCSI, 3–24

removing side panel, 2–7

reset, AT bus, 3–13

reset keyswitch, 1–7

resetepc, environment variable, 5–4

ResetPC/ATB bit, System Configuration Register, 3–13

resetra, environment variable, 5–4

ResetSCSI bit, System Configuration Register, 3–13


# S


sash (standalone shell), F–9—F–12

sash commands, F–10—F–12

SCSI
  byte count pointer, 3–12
  device assignments, C–1
  direction control, 3–13
  operation details, 3–25
  registers, 3–24
  reset bit, 3–13

SCSI devices, 4–14
  hard disks, 4–14
  LUNS, 4–14
  tape, 4–14

SCSI interface, 3–23—3–25

SCSIHIN bit, System Configuration Register, 3–13

serial I/O devices, connecting, 2–11, 2–13

serial ports
  *See also* DUART
  pinouts, 2–12

setenv, PROM Monitor command, 5–6, 5–24

side panel
  reinstalling, 2–10
  removing, 2–7

signal naming conventions, 3–2

SIO connectors, 3–23

site selection, 2–1—2–4

sload, PROM Monitor command, 5–6, 5–25

slow UDC registers, DMA controller, 3–16

SlowUDCEn bit
  DMA controller, 3–16
  System Configuration Register, 3–13

SoftEOP bit
  DMA controller, 3–16
  System Configuration Register, 3–13

software control, DMA controller, 3–16

space requirements, 2–1

SPC, 4–14
  SCSI Protocol Controller. *See* SCSI

specifications, 1–9
  environmental, 2–4
  physical, 2–2
  power, 2–3

spin, PROM Monitor command, 5–6, 5–26

ss.h, E–22

standalone programs, F–1—F–12
  format, F–1
  sash, F–1

standalone shell, F–9—F–12

status register, interrupts, 3–7

swapping bytes, AT bus, 3–27

switches, 1–7

sysgen file, including a driver
  ATBUS, 4–6
  INCLUDE, 4–6
  VECTOR, 4–6

System Configuration Register, 3–12—3–14
  ATTCEn bit, 3–13
  BootLockB bit, 3–13
  ColdStart bit, 3–13
  CoProcB bit, 3–13
  ForceBadPar bit, 3–14
  Key0 bit, 3–12
  ParityEn bit, 3–14
  Pointer bits, 3–12
  ResetPC/ATB bit, 3–13
  ResetSCSI bit, 3–13
  SCSIHIN bit, 3–13
  SlowUDCEn bit, 3–13
  SoftEOP bit, 3–13


# T


tape drive, operation and maintenance, B–1—B–4

TC (terminal count) bit, DMA controller, 3–16

TC enable bit, 3–13

terminal adapter, connections for, 2–12

time of day, PROM monitor, 5–5

TimeOut bit, Fault ID Register, 3–10

timer, 3–18

transaction type, DMA operating modes, 3–15

transfer type, DMA operating mode, 3–15

troubleshooting, pon diagnostics, D–4

## U

UDC, 4–14
  Universal DMA Controller. *See* DMA
UDC (Universal DMA Controller). *See* DMA
unit, PROM Monitor, 5–2
unlock keyswitch, 1–7
unsetenv, PROM Monitor command, 5–6, 5–27

## V

version, environment variable, 5–4
volitile memory, 4–12
voltage requirments, 2–3
voltage selection, 2–5

## W

warm, PROM Monitor command, 5–6, 5–28
warm start. *See* ColdStart bit
write buffer, 4–13
write protection, cartridge tape, B–2

## Customer Response Card

Your comments, which can assist us in improving our products and our publications, are welcome.

If you wish to reply, be sure to include your name and address, *and the name and part number that appears on the first page of this manual.*

Thank you for your cooperation.

No postage necessary if mailed in the U. S. A.
After writing comments, detach this page and then fold, seal, and mail.

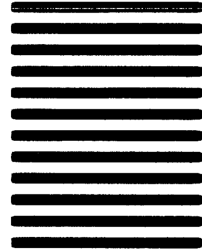
## Comments

**Name of manual:** ____________________

**Part number:** ____________

MIPS may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1659 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE:

**MIPS Computer Systems**
**928 Arques Avenue**
**Sunnyvale, CA 94086-9756**