AA-JP76A-TH

VAXmate™

Technical Reference Manual Volume 1

digital software

VAXmate™

Technical Reference Manual Volume 1

First Printing, February 1987

© Digital Equipment Corporation 1987. All Rights Reserved.

The material in this document is for informational purposes and is subject to change without notice; it should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

MS-DOS, MS-WINDOWS, and MS-NET are trademarks of Microsoft Corporation.
Topview is a trademark of International Business Corporation.
Motorola is a registered trademark of Motorola, Inc.
IBM PC AT is a trademark of International Business Machines Corporation.

The following are trademarks of Digital Equipment Corporation.

digital	IAS	Professional
DEC	MASSBUS	Rainbow
DECmate	MicroPDP	RSTS
DECnet	MicroVAX	RSX
DECsystem-10	MINC-11	ThinWire
DECSYSTEM-20	OMNIBUS	VAX
DECUS	OS/8	VAXmate
DECwriter	PDP	VMS
DIBOL	PDT	VT
EduSystem	P/OS	Work Processor

Printed in U.S.A.

Contents

Bibliography

Index

Tables

Figures

Preface

Audience

This manual provides reference material about the VAXmate workstation. It covers all programmable components, the firmware, and several MS-DOS related environments. The material and its presentation are directed to experienced programmers or software designers.

Manual Organization

This manual is divided into four parts and appendixes:

- Chapter 1 provides an overview of the VAXmate workstation and optional equipment.
- Chapters 2 through 13 introduce the VAXmate workstation programmable hardware devices. Each chapter discusses a single hardware programming task, such as video input/output (I/O), external interrupt processing, or serial communications and includes the following information:
  - A brief device description
  - A list of additional references
  - A description of the programmable hardware registers
  - A programming example
  - A discussion of the example

  The examples are written in the C programming language to reduce the size of the examples and focus on the task rather than the detail required by the language.
- Chapter 14 describes the power-up diagnostics and system startup.
- Chapter 15 describes the read-only memory basic input/output system (ROM BIOS).
- The appendixes contain additional information, including a bibliography of other useful publications.

Terminology

The following terms are used throughout this manual and are defined as follows:

Term	Definition	
Industry-standard	The computer industry recognizes two open architectures as industry standards, the IBM PC AT bus structure and the Microsoft disk operating system (MS-DOS). Moreover, supporting MS-DOS requires a defined set of ROM BIOS services. The term *industry-standard* refers to compatibility with these architectures.	
Reserved Available Unassigned	To avoid confusion and incompatibility, the use of certain items such as memory space, I/O space, interrupt vectors, and ROM BIOS parameters or return values must be clearly defined. These three categories define those items that do not have a specific use.	
	Reserved	In future hardware or software releases, DIGITAL may define a specific use for this item. Hardware or software applications that use this item may not work with future releases.
	Available	Hardware or software applications can use this item. DIGITAL has defined the specific use of this item as available for applications.
	Unassigned	Hardware or software applications can use this item. However, there remains some risk that DIGITAL may define a specific use for this item.

Federal Communications Commission Radio Frequency Interference

Class A Computing Devices

This equipment generates, uses, and may emit radio frequency energy. The equipment has been tested and found to comply with the limits for a Class A computing device pursuant to Sub-part J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such radio frequency interference when operated in a commercial environment. Operation of this equipment in a residential area may cause interference in which case the user at his own expense may be required to take measures to correct the interference.

If this equipment does cause interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following methods:

- re-orient the receiving antenna
- relocate the computer with respect to the receiver
- move the computer away from the receiver
- plug the computer into a different outlet so that computer and receiver are on different branch circuits.

If necessary, the user should consult the dealer or an experienced radio and television technician for additional suggestions. The user may find the booklet, *How to Identify and Resolve Radio/TV Interference Problems*, prepared by the Federal Communications Commission helpful. This booklet is available from the U.S. Government Printing Office, Washington, DC 20402, Stock No. 004-000-00398-5.

> **NOTE**
> Shielded cables are provided for use with this device. Should any cables be replaced or added for any reason, these cables should be the same as, or with higher shielding capabilities, than those provided by Digital Equipment Corporation.

Chapter 1
VAXmate Workstation Overview

This chapter describes the VAXmate workstations physical appearance, base configuration, optional components, and the logical relationship of the components.

The VAXmate workstation is a high-performance, standalone, desktop personal computer that executes industry-standard software. The integral Ethernet interface allows the VAXmate workstation to communicate on a network. The hard disk storage, provided in the optional expansion box, allows the VAXmate workstation to be a server on a network.

Base System

In the base configuration, the VAXmate workstation has three units:

- System unit
- Keyboard
- Mouse

Figure 1-1 shows a base configuration workstation.

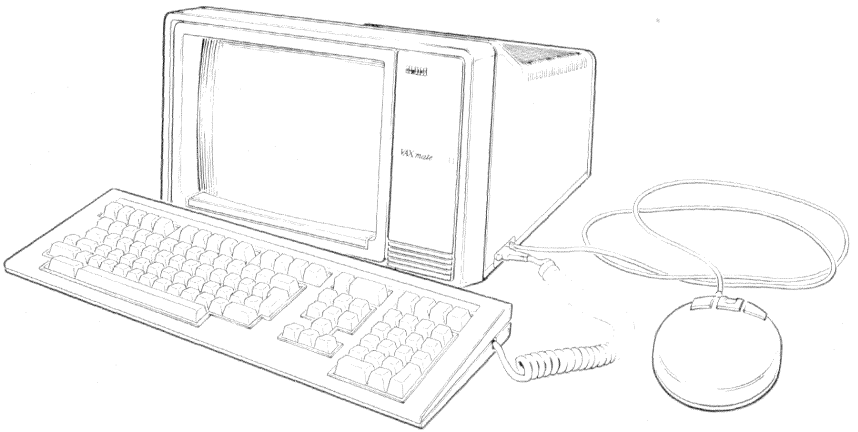


Figure 1-1 Base Configuration Workstation

In the base configuration, the system unit contains the following major components:

- 80286 microprocessor
- One megabyte of dynamic random-access memory (DRAM)
- Video monitor and controller
- Diskette drive and controller
- Ethernet controller
- Keyboard interface controller
- Event timer
- Real time clock and calendar
- Serial communications port
- Serial printer port
- Serial mouse port
- Speaker
- Power supply

Optional Components

The workstation provides for the following optional components:

- An expansion box, part number RCD31-EA, that attaches to the bottom of the system unit. Figure 1-2 shows the system unit with the expansion box attached. The expansion box contains an additional power supply, a battery, a 20 megabyte hard disk drive and controller, and two industry-standard expansion slots.
- An 80287 coprocessor, part number FP287, that installs in the system unit. Figure 1-3 shows the 80287 coprocessor.
- A two megabyte DRAM module, part number PC50X-AA, that installs in the system unit. Figure 1-4 shows the two megabyte DRAM module.
- A modem module, part number PC50X-MA, that installs in the system unit. Figure 1-5 shows the modem module.

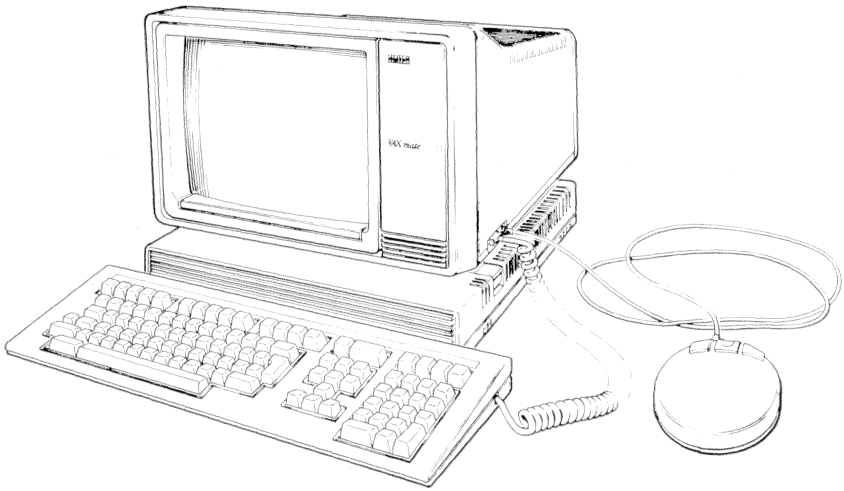


Figure 1-2 Workstation With Installed Expansion Box

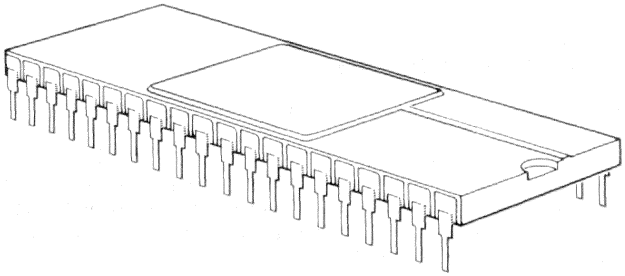


Figure 1-3 Optional 80287 Coprocessor

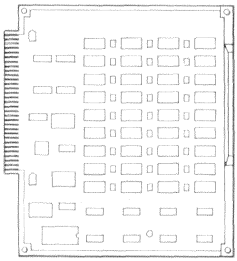


Figure 1-4 Optional Two Megabyte DRAM Module

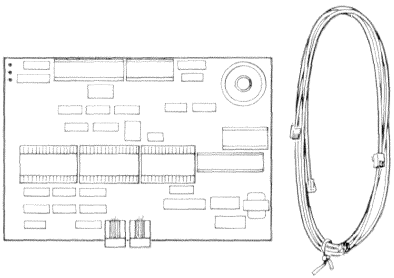


Figure 1-5 Optional Modem Module

Figure 1-6 shows the relationship of the workstation components. The battery is present only when an expansion box is installed.

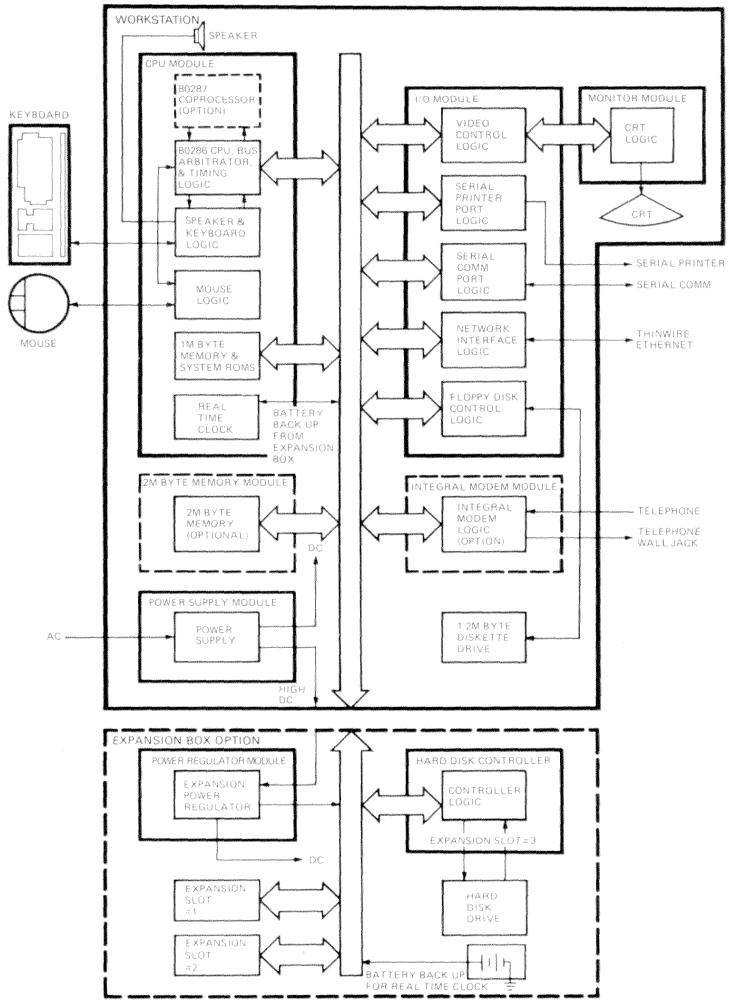


Figure 1-6 Block Diagram of Workstation Components

Chapter 2
VAXmate Microprocessor

Overview

The VAXmate microprocessor is an Intel 80286 central processing unit (CPU). The CPU is a high-performance, 8 MHz microprocessor with a 16-bit external data path and a 24-bit address path. The CPU provides two modes of operation, real address mode and protected virtual address mode.

Real Address Mode

On powerup, the CPU operates in real address mode. In real address mode, the 80286 CPU behaves as though it is a fast 8086 CPU. It is limited to the 1 Mbyte address range of the 8086 CPU. In real address mode, the ROM is accessed in the address range 0F0000H-0FFFFFH.

Protected Virtual Address Mode

In protected virtual address mode, the 80286 CPU can access 16 Mbytes of physical memory and 1 gigabyte of virtual memory.

The ROM is redundantly mapped to two physical address ranges, 0F0000H-0FFFFFH and FF0000H-FFFFFFH. Therefore, in protected virtual address mode, the 80286 CPU can access the ROM at either physical address range. However, the majority of the ROM BIOS code is not valid in protected virtual address mode.

The CPU uses the keyboard interface controller or a double exception fault to reset to real address mode from protected virtual address mode. The keyboard interface controller is discussed in Chapter 8.

Coprocessor

The optional coprocessor for the VAXmate workstation is an Intel 80287 processor extension chip. It is a high-performance, numeric processor that extends the CPU data types to include floating-point, extended-integer, and binary-coded decimal (BCD).

Additional Sources of Information

The following Intel Corporation documents provide additional information on the CPU and coprocessor:

- *Introduction to the iAPX 286* (Publication Number 210308)
- *iAPX 286 Hardware Reference Manual* (Publication Number 210760)
- *iAPX 286 Programmer's Reference Manual* (Publication Number 210498)
- *Microsystem Components Handbook* (Publication Number 230843)

Memory Map

The base configuration workstation has 1 Mbyte of RAM and 64 Kbytes of ROM. An optional memory module can be added without an expansion box.

Table 2-1 describes the VAXmate workstation's physical memory map. The 1 Mbyte of RAM is divided into three, noncontiguous blocks. In Table 2-1, these blocks are labeled BLOCK1, BLOCK2, and BLOCK3.

Table 2-1 Physical Memory Map

From	To	Size (Bytes)	Description
000000H	09FFFFH	640K	System RAM (BLOCK1)
0A0000H	0AFFFFH	64K	Reserved
0B0000H	0BFFFFH	64K	Video RAM
			During video mode setup, the video RAM is dynamically mapped. Only the video RAM required by the current video mode is accessible.
0C0000H	0CFFFFH	64K	Available for options with expansion ROM
0D0000H	0EFFFFH	128K	DIGITAL private RAM (BLOCK2)
0F0000H	0FFFFFH	64K	System ROM
100000H	EFFFFFH	14336K	Optional RAM space
F00000H	F1FFFFH	128K	Reserved RAM space
F20000H	F5FFFFH	256K	DIGITAL private RAM (BLOCK3)
F60000H	F7FFFFH	128K	Reserved RAM space
F80000H	FEFFFFH	448K	Reserved ROM space
FF0000H	FFFFFFH	64K	System ROM (redundantly mapped from 0F0000H)

Input/Output Address Map

Table 2-2 describes the VAXmate workstation's I/O address map. Many of the I/O ports have an industry-standard assignment. Recognition of that assignment does not indicate that the device is present in the workstation.

Table 2-2 Input/Output Address Map

From	To	Device	Description
0000H	001FH	8237A-5	DMA controller
0020H	003FH	8259A	Interrupt controller #1
0040H	005FH	8254-2	Timer
0060H	006FH	8042	Keyboard interface controller
0070H	———	———	Bit 7 controls the NMI mask register
0070H	0077H	MC146818	Real-time clock and CMOS RAM
0078H	007FH	———	Reserved
0080H	009FH	74LS670	DMA page registers
00A0H	00BFH	8259A	Interrupt controller #2
00C0H	00DFH	———	Reserved
00E0H	00EFH	———	Unassigned
00F0H	———	———	Clear math coprocessor busy
00F1H	———	———	Reset math coprocessor
00F2H	00F7H	———	Unassigned
00F8H	00FFH	80287	Math coprocessor
0100H	01EFH	———	Unassigned
01F0H	01F8H	WD2010	Hard disk controller
01F9H	01FFH	———	Unassigned
0200H	0207H	———	Game port I/O
0208H	0277H	———	Unassigned
0278H	027FH	———	Parallel printer port #2
0280H	02F7H	———	Unassigned
02F8H	02FFH	8250	Serial port #2 (Integral modem option)

Table 2-2 Input/Output Address Map (cont.)

From	To	Device	Description
0300H	031FH	———	Reserved
0320H	035FH	———	Unassigned
0360H	036FH	———	Reserved
0370H	0377H	———	Unassigned
0378H	037FH	———	Parallel printer port #1
0380H	038FH	———	Reserved
0390H	039FH	———	Unassigned
03A0H	03AFH	———	Reserved
03B0H	03BFH	———	Reserved
03C0H	03CFH	———	Reserved
03D0H	03DFH	6845	Graphics video controller
03E0H	03EFH	———	Unassigned
03F0H	03F5H	PD765A	Diskette controller
03F6H	03F7H	———	Hard disk and diskette controllers
03F8H	03FFH	8250	Serial port #1
0400H	0BFFH	———	Unassigned *
0C00H	0C1FH	———	System CSR 1
0C20H	0C3FH	———	Ethernet ROM
0C40H	0C5FH	2661	Universal Asynchronous Receiver/ Transmitter (UART) for mouse port
0C60H	0C7FH	———	Network Controller and Interface
0C80H	———	———	Special purpose register
0C81H	0C9FH	———	Reserved
0CA0H	0CA7H	8250	Integral serial printer port
0CA8H	0DFFH	———	Reserved
0E00H	FFFFH	———	Unassigned *

* Industry-standard, processor-board, I/O ports in the address range 0000H-00FFH respond to these I/O addresses. Therefore, I/O to the expansion box in this address range is undefined.

Interrupt Vector Map

Table 2-3 shows the VAXmate workstation's interrupts. The four columns in Table 2-3 provide the following information:

- The *interrupt* column identifies the interrupt number in hexadecimal.
- The *type* column is interpreted as follows:
  - The letter *E* indicates a processor exception interrupt.
  - The letter *H* indicates a hardware interrupt.
  - The letter *S* indicates a software interrupt.
  - The letter *P* indicates that the interrupt vector space contains a pointer to a parameter table or an application routine.
  - The letter *N* indicates that the vector has no assignment.
- The *description* column identifies the specific assignment of the interrupt vector.
- The *service* column indicates whether or not the ROM BIOS services the interrupt. During system startup, interrupt vectors that are not serviced by the ROM BIOS are initialized to point to an interrupt return (IRET) instruction, indicated by **IRET**. For information on ROM BIOS-serviced software interrupts, see Chapter 15.

Table 2-3 Interrupt Vector Map

Interrupt	Type	Description	Service
00H	E	Divide by zero	IRET
01H	E	Single step	IRET
02H	H	NMI	ROM BIOS
03H	S	Breakpoint (Used by DEBUG)	IRET
04H	E	Overflow	IRET
05H	S	Print Screen function	ROM BIOS
06H-07H	N	Reserved	IRET
08H	H	Timer interrupt service (IRQ0)	ROM BIOS
09H	H	Keyboard interrupt service (IRQ1)	ROM BIOS
0AH	H	Reserved (IRQ2 interrupt from controller #2)	IRET
0BH	H	Serial port #2 (Asynchronous) (modem option) (IRQ3)	IRET
0CH	H	Serial port #1 (Asynchronous) (IRQ4)	ROM BIOS
0DH	H	Unassigned (IRQ5)	IRET
0EH	H	Diskette interrupt service (IRQ6)	ROM BIOS
0FH	H	Parallel printer port #1 (IRQ7)	IRET
10H	S	Video I/O	ROM BIOS
11H	S	Return configuration	ROM BIOS
12H	S	Return memory size	ROM BIOS
13H	S	Diskette and hard disk I/O	ROM BIOS
14H	S	Asynchronous communications I/O	ROM BIOS
15H	S	Extended ROM BIOS functions	ROM BIOS
16H	S	Keyboard I/O	ROM BIOS
17H	S	Printer Output	ROM BIOS
18H	S	Invoke network boot/Maintenance Operation Protocol (MOP)	ROM BIOS
19H	S	Bootstrap	ROM BIOS

Table 2-3 Interrupt Vector Map (cont.)

Interrupt	Type	Description	Service
1AH	S	Time of day	ROM BIOS
1BH	P	Keyboard BREAK vector	IRET
1CH	P	Timer tick vector	IRET
1DH	P	Video parameter table	ROM BIOS
1EH	P	Diskette parameter table	ROM BIOS
1FH	P	Graphics character table (Character codes 80H-FFH)	ROM BIOS
20H-3FH	S	Reserved for MS-DOS	IRET
40H	P	INT 13H redirect when hard disk in use	ROM BIOS
41H	P	Parameter table pointer for hard disk 0	ROM BIOS
42H-45H	N	Reserved	IRET
46H	P	Parameter table pointer for hard disk 1	ROM BIOS
47H-5FH	N	Reserved	IRET
60H-67H	N	Available for application or user program interrupts	IRET
68H-6FH	N	Reserved for DECnet software	IRET
70H	H	Real time clock interrupt (IRQ8)	ROM BIOS
71H	H	Redirect to interrupt 0AH - Old IRQ2 (IRQ9)	IRET
72H	H	Ethernet controller (IRQ10)	ROM BIOS
73H	H	Serial printer port (IRQ11)	ROM BIOS
74H	H	Mouse port (IRQ12)	IRET
75H	H	80287 error (IRQ13)	ROM BIOS
76H	H	Hard disk controller (IRQ14)	ROM BIOS
77H	H	Unassigned (IRQ15)	IRET
78H-7FH	N	Unassigned	IRET
80H-F0H	N	Reserved	IRET
F1H-FFH	N	Unassigned	

Bus Timing and Structure

The 8 MHz clock rate results in a 125 ns processor cycle. Normal operation of the 80286 CPU requires two processor cycles. With zero wait states, a read or write cycle requires 250 ns.

There are three data bus structures:

- A 16-bit local bus
- An 8-bit expansion bus
- A 16-bit expansion bus

16-Bit Local Bus

The local bus connects the CPU to on-board memory and on-board peripherals. One wait state is added to local bus memory transfers, resulting in a 375 ns bus cycle. Two wait states are added to local bus input/output (I/O) transfers, resulting in a 500 ns bus cycle.

The RAM access time is 150 ns. The ROM access time is 250 ns.

> **NOTE**
> In assembly language programming, directing two or more contiguous I/O instructions at the same device may not provide enough time for the device to respond. This is possible because peripheral devices respond more slowly than the 80286 processor executes. A jump instruction consumes processor cycles and clears the processor pre-fetch queue. Thus, jumps to successive I/O instructions provide the required response time. The C language I/O functions, commonly named **in()** and **out()** or **inp()** and **outs()**, provide enough response time because they contain sufficient overhead in the calling sequence.

16-Bit Expansion Bus

The 16-bit expansion bus supports word transfers to memory and I/O. One wait state is automatically added, resulting in a 375 ns bus cycle.

8-Bit Expansion Bus

The 8-bit expansion bus supports byte and word transfers to memory and I/O. Word transfers are controlled by hardware, which issues two sequential byte transfers (low byte first and high byte second). Table 2-4 describes the bus cycle times for all transfer types on the 8-bit expansion bus.

Table 2-4 8-Bit Expansion Bus Transfer Times

Type	Size	Time	Wait States
Memory	Byte	750 ns	4
Memory	Word (two, 8-bit transfers)	1500 ns	8
I/O	Byte	1125 ns	7
I/O	Word (two, 8-bit transfers)	2250 ns	14

Expansion Box Technical Specifications

The VAXmate expansion box provides two expansion module slots. Each slot accommodates a single expansion module. If a mother-daughter board is used in one of the slots, then both slot areas will be used and it will not be possible to add a second module. Table 2-5 shows the amperage (current) and wattage values available for each slot. Each slot has an 8-bit and a 16-bit bus connector. Figure 2-1 shows the pin numbers and signal names for the 8-bit and 16-bit bus connector.

Table 2-5 Expansion Slot Power Ratings

Slot	+5.1V	+12.1V	-12.0V	-5.0V	Watts
OPT-1	1.300	0.100	0.100	0.100	9.540
OPT-2	1.300	0.100	0.100	0.100	9.540

Expansion Box Operating Ranges

Ambient Operating Temperature: 15 C (59 F) to 32 C (90 F)

Relative Humidity: 8% to 80%

8-Bit Bus Connector

GROUND	B1	A1	I/O CHK L
RESET H	B2	A2	SD7 H
+5V	B3	A3	SD6 H
IRQ9 H	B4	A4	SD5 H
-5V	B5	A5	SD4 H
DRQ2 H	B6	A6	SD3 H
-12V	B7	A7	SD2 H
OWS L	B8	A8	SD1 H
+12V	B9	A9	SD0 H
GROUND	B10	A10	I/O RDY H
MEMW L	B11	A11	AEN H
MEMR L	B12	A12	SA19 H
IOW L	B13	A13	SA18 H
IOR L	B14	A14	SA17 H
DACK3 L	B15	A15	SA16 H
DRQ3 H	B16	A16	SA15 H
DACK1 L	B17	A17	SA14 H
DRQ1 H	B18	A18	SA13 H
REFRESH L	B19	A19	SA12 H
CLOCK H	B20	A20	SA11 H
IRQ7 H	B21	A21	SA10 H
IRQ6 H	B22	A22	SA9 H
IRQ5 H	B23	A23	SA8 H
IRQ4 H	B24	A24	SA7 H
IRQ3 H	B25	A25	SA6 H
DACK2 L	B26	A26	SA5 H
T/C H	B27	A27	SA4 H
ALE H	B28	A28	SA3 H
+5V	B29	A29	SA2 H
OSC H	B30	A30	SA1 H
GROUND	B31	A31	SA0 H

16-Bit Bus Connector

MEM16 L	D1	C1	SBHE L
I/O16 L	D2	C2	UA23 H
IRQ10 H	D3	C3	UA22 H
IRQ11 H	D4	C4	UA21 H
IRQ12 H	D5	C5	UA20 H
IRQ15 H	D6	C6	UA19 H
IRQ14 H	D7	C7	UA18 H
DACK0 L	D8	C8	UA17 H
DRQ0 H	D9	C9	EMEMR L
DACK5 L **	D10	C10	EMEMW L
DRQ5 H **	D11	C11	SD08 H
DACK6 L **	D12	C12	SD09 H
DRQ6 H **	D13	C13	SD10 H
DACK7 L **	D14	C14	SD11 H
DRQ7 H **	D15	C15	SD12 H
+5V	D16	C16	SD13 H
MASTER L	D17	C17	SD14 H
GROUND	D18	C18	SD15 H

** Not Implemented

Figure 2-1 8-Bit and 16-Bit Bus Connectors

Chapter 3
Interrupt Controllers

Overview

The VAXmate 80286 central processing unit (CPU) has two interrupt input lines, the Non-Maskable Interrupt (NMI) and the Interrupt Request (INTR). When these hardware inputs are active, the CPU suspends execution of the current program and begins execution of an *interrupt handler*. An *interrupt handler* is a program or program segment that responds to a specific event. This allows an immediate response to asynchronous external events and the segregation of program responsibility for handling those events.

The interrupt input lines are assigned to different classes of events. The NMI is dedicated to two catastrophic events, memory parity errors and I/O bus errors. The INTR is assigned all other external interrupt sources, such as diskette and hard disk controllers, serial and parallel ports, and clocks. The reason for this division is the way the CPU implements the two interrupts:

- The NMI has a higher priority than the INTR.
- The CPU has no way to disable the NMI input.
- The CPU provides handshaking protocol during INTR processing, but not during NMI processing.
- The NMI generates only one interrupt vector, which is fixed.

Because the CPU does not provide handshaking during NMI processing, the CPU cannot communicate with an interrupt controller. Therefore, the NMI sources are connected directly to the NMI input. To determine the source of the interrupt, the NMI interrupt handler must read the status output of the sources.

The INTR input is buffered by two, 8259A interrupt controllers. The interrupt controllers reduce the CPU interrupt processing overhead in the following ways:

- They resolve the priority of simultaneous or overlapping interrupts.
- They concentrate multiple interrupts into one source.
- They provide the vector number of the interrupt handler.

Each interrupt controller is capable of handling eight interrupt requests. The 16 inputs are labeled IRQ0-IRQ15. Controller 1 buffers IRQ0-IRQ7 and controller 2 buffers IRQ8-IRQ15. Although they are physically identical, the interrupt controllers have a master/slave relationship. The output of controller 2 (the slave) is connected to the IRQ2 input of controller 1 (the master). The output of the master is connected to the INTR input of the CPU. Table 3-1 shows all of the IRQ inputs.

Table 3-1 Interrupt Request Lines

Priority	Controller #1 MASTER	Controller #2 SLAVE	Source
1	IRQ0		Event timer output 0
2	IRQ1		Keyboard controller
3	IRQ2		Slave interrupt controller
3.1		IRQ8	Real-time clock
3.2		IRQ9	Software redirection to IRQ2
3.3		IRQ10	LANCE (Ethernet)
3.4		IRQ11	Serial printer port
3.5		IRQ12	Mouse port
3.6		IRQ13	Coprocessor error
3.7		IRQ14	Hard disk drive controller
3.8		IRQ15	Available, 16-bit bus
4	IRQ3		Reserved, integral modem option
5	IRQ4		Asynchronous communications port
6	IRQ5		Available, 8-bit bus
7	IRQ6		Diskette drive controller
8	IRQ7		Available, 8-bit bus

Additional Source of Information

The following Intel Corporation document provides additional information:

- *Microsystem Components Handbook* (Publication Number 230843)

Read/Write Control

The 8259A interrupt controller has the following registers:

Initialization Command Words (ICW) - There are four initialization command words (ICW1-ICW4). They establish the operating conditions of the interrupt controller and are written only during system initialization.

Operation Command Words (OCW) - There are three operational command words (OCW1-OCW3). These registers select access to internal controller registers and control the run-time aspects of the interrupt controller.

Interrupt Mask Register (IMR) - The IMR selectively enables and disables the interrupt controller's interrupt input lines. In this manual, IMR refers to the physical register and OCW1 refers to the command to read or write the interrupt mask register.

Interrupt Request Register (IRR) - Following a CPU interrupt acknowledge, each bit in the IRR reflects the state of the corresponding interrupt input.

In-Service Register (ISR) - The ISR register indicates the interrupt input lines that the CPU is currently servicing.

Poll data - The poll data indicates whether any enabled interrupt inputs are active. If any enabled interrupt inputs are active, it also contains the interrupt input number of the highest priority input requesting service.

Although the 8259A interrupt controller has many registers, it has only two input/output (I/O) ports. Table 3-2 shows the master and slave I/O port addresses. Table 3-3 shows the registers and the requirements to access them.

Table 3-2 Master and Slave I/O Addresses

Port	Master	Slave
0	0020H	00A0H
1	0021H	00A1H

Table 3-3 Accessing the Interrupt Controller Registers

Register	R/W	Port	Access Method
ICW1	W	0	When bit 4 of the value written to port 0 equals 1, ICW1 is selected.
ICW2	W	1	Must be the next byte written after ICW1.
ICW3	W	1	The interrupt controller expects ICW3 only if ICW1, bit 1 equals 1. If written, ICW3 must be the next byte written after ICW2.
ICW4	W	1	The interrupt controller expects ICW4 only if ICW1, bit 0 equals 1. If ICW4 is written and ICW3 is not, ICW4 must be the next byte written after ICW2. If ICW3 and ICW4 are written, ICW4 must be the next byte written after ICW3.
OCW1	R/W	1	Reading or writing OCW1 requires only that the initialization process be complete. Reading or writing OCW1 accesses the interrupt mask register.
OCW2	W	0	Writing OCW2 requires that the initialization process be complete and OCW2 bits 4-3 are equal to 0.
OCW3	W	0	Writing OCW3 requires that the initialization process be complete, OCW3 bit 4 equals 0, and OCW3 bit 3 equals 1.
IRR	R	0	Reading the IRR is a two-step process. First, issue the read IRR command (write OCW3 with OCW3 bit 1 equals 1 and OCW3 bit 0 equals 0). Then, read the IRR through port 0. Until another command is written to OCW3, subsequent reads of port 0 return the IRR.
ISR	R	0	Reading the ISR is a two-step process. First, issue the read ISR command (write OCW3 with OCW3 bit 1 equals 1 and OCW3 bit 0 equals 1). Then, read the ISR through port 0. Until another command is written to OCW3, subsequent reads of port 0 return the ISR.
Poll Data	R	0	Reading the poll data is a two-step process. First, issue the read poll data command (write OCW3 with OCW3 bit 2 equals 1). Then, read the poll data through port 0. The OCW3 poll command must always be written prior to reading the poll data.

Initialization Command Words

The 8259A interrupt controllers do not have a hardware reset. After power is applied to the system and until they are initialized, the interrupt controllers are in an undefined state. The VAXmate startup code initializes the interrupt controllers.

Initializing the 8259A interrupt controller requires from two to four initialization command words written in sequence.

The interrupt controller recognizes ICW1 as the start of an initialization sequence. An ICW1 resets the interrupt controller as follows:

1. The trigger mode is cleared to edge-triggered mode and the edge sense circuit is reset. After initialization, an interrupt request input must make a low-to-high transition to generate an interrupt.
2. All bits in the IMR are cleared (enabled). Because the initialization sequence enables interrupt inputs, on completion of the initialization sequence, the interrupt controllers can immediately issue interrupt requests. Therefore, the interrupt vectors and handlers should be initialized prior to initializing the interrupt controllers.
3. The IRQ7 input is assigned priority 7.
4. The slave mode address is set to 7.
5. If ICW1 bit 0 equals 0, all bits in ICW4 are cleared (0).
6. In the OCW3 register, the special mask mode is cleared (disabled) and status read bits are set to read the IRR.
7. The interrupt controller enters fully nested mode. All other modes of operation are variations of this mode. In fully nested mode, the interrupt inputs have a fixed order of decreasing priority and the priority of an input corresponds to its input number 0 (highest) - 7 (lowest). While the CPU is servicing an interrupt (until the interrupt controller receives an end-of-interrupt command), the controller inhibits interrupts of equal or lower priority. However, the current interrupt service can be nested in favor of a higher priority interrupt as follows:
   - The higher priority interrupt input must be unmasked (enabled).
   - The CPU INTR input must be enabled (STI instruction).

NOTE
The 8259A interrupt controller is compatible with two microprocessor families, 8080/8085 and 8088/8086/80286. Because the VAXmate CPU is an 80286, this manual describes only the 80286 application. Those bits dedicated to the 8080/8085 family are unused and described only as belonging to the 8080/8085 family.

The 8259A interrupt controller has the following mutually exclusive methods of indicating whether an interrupt controller is a master or a slave:

- The initialization sequence selects nonbuffered mode in ICW4. In nonbuffered mode, a hardware connection to the SP/EN pin determines whether the controller is a master or a slave. In this mode, a high level at the SP/EN pin indicates a master and a low level at the SP/EN pin indicates a slave. The VAXmate workstation uses this method.
- The initialization sequence selects a buffered master or a buffered slave in ICW4.

Initialization Command Word 1 (0020H/00A0H)

7	6	5	4	3	2	1	0
0	0	0	1	TRIGGER MODE	0	SINGLE/ CASCADE	ICW4 REQUEST

Bit	R/W	Description
7-5	W	Always 0 (These bits are used only by the 8080/8085 CPU family.)
4	W	Always 1 For values written to port 0, this bit distinguishes an ICW1 from operational command words 2 and 3. For additional information, see Table 3-3.
3	W	TRIGGER MODE 0 = Edge-triggered mode 1 = Level-triggered mode For either trigger mode, a low-to-high transition at an interrupt input generates an interrupt request. In edge-triggered mode, to generate another interrupt request at the same input, the input must change from high to low and back to high. In level-triggered mode, while that interrupt input remains high, the controller can generate additional interrupt requests for that input. The VAXmate startup code initializes the interrupt controllers to edge-triggered mode.
2	W	Always 0 (This bit is used only by the 8080/8085 CPU family.)
1	W	SINGLE/CASCADE 0 = Cascade mode 1 = Single mode Single mode indicates that this is the only interrupt controller in the system. Therefore, it is neither a master nor a slave and ICW3 is not written. Cascade mode indicates that there is more than one interrupt controller in the system. Therefore, it is either a master or a slave and ICW3 is required. The VAXmate workstation uses cascade mode.
0	W	ICW4 REQUEST 0 = ICW4 is not required 1 = ICW4 is required This bit indicates whether ICW4 is required in the initialization sequence. The VAXmate workstation requires ICW4.

For the master ICW1 and the slave ICW1, use 11H.

Initialization Command Word 2 (0021H/00A1H)

7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	0	0	0

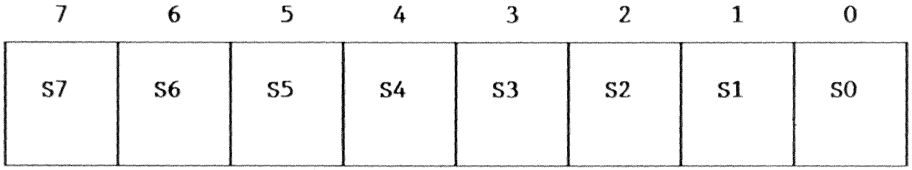
Bit	R/W	Description
7-3	W	Bits 7-3 of the interrupt number for interrupt input 0. This value corresponds to the address of the interrupt vector divided by four. The interrupt controller generates a sequential interrupt number for each of the interrupt inputs by *OR*ing the interrupt input number and ICW2. Because the interrupt input number is *OR*ed to the value in ICW2, there is no carry involved. Therefore, the value in ICW2 must be evenly divisible by 8 (modulo 8).
2-0	W	Always 0

For the master ICW2, use 08H. For the slave ICW2, use 70H.

Initialization Command Word 3 (0021H/00A1H)

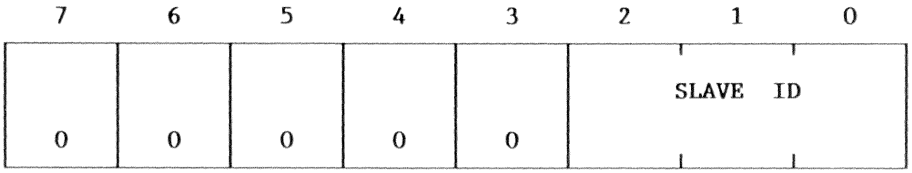
When there are two or more interrupt controllers in the system, an ICW3 is used in the initialization sequence. The VAXmate workstation has two interrupt controllers and requires ICW3. The meaning and use of ICW3 depends on whether the interrupt controller is a master or a slave.

ICW3 (Master)



Bit	R/W	Description
7-0	W	For each master interrupt input that is connected to a slave, the corresponding ICW3 bit is set (1). The master interrupt controller can then determine which interrupt inputs require a slave identification on the cascade lines. For the master ICW3, use 04H.

ICW3 (Slave)



Bit	R/W	Description
7-3	W	Always 0
2-0	W	SLAVE ID - Slave Identification The slave identification is the master interrupt input (7-0) to which the slave is connected. During the CPU interrupt acknowledge sequence, the slave compares its cascade input to these bits. If they are equal, the slave places the interrupt vector number on the I/O data bus. For the slave ICW3, use 02H.

Initialization Command Word 4 (0021H/00A1H)

7	6	5	4	3	2	1	0
0	0	0	SPECIAL FULLY NESTED MODE	BUFFER MODE	MASTER/ SLAVE	EOI MODE	CPU MODE

Bit	R/W	Description
7-5	W	Always 0
4	W	SPECIAL-FULLY-NESTED MODE * 0 = Disable special-fully-nested mode 1 = Enable special-fully-nested mode
3-2	W	BUFFERED MODE and MASTER/SLAVE 0X = Nonbuffered mode - In nonbuffered mode, a hardware connection to the SP/EN pin determines whether the controller is a master or a slave and bit 2, the master/slave selection, has no effect. In this mode, a high level at the SP/EN pin indicates a master and a low level at the SP/EN pin indicates a slave. The VAXmate workstation uses this mode. 10 = Buffered mode slave - The VAXmate workstation is incapable of operating in this mode. 11 = Buffered mode master - The VAXmate workstation is incapable of operating in this mode.
1	W	EOI MODE - End-of-interrupt Mode 0 = Normal EOI - In this mode, the CPU must write an EOI command to the interrupt controller. The VAXmate startup code initializes the interrupt controller to normal EOI mode. The EOI command is explained in the operation command word 2 description. 1 = Automatic EOI - In automatic EOI mode, the interrupt controller generates its own EOI on the second acknowledge pulse.
0	W	CPU MODE 0 = 8080/8085 microprocessor family 1 = 8088/8086/80286 microprocessor family The VAXmate workstation uses the 80286 CPU mode.

* Special-fully-nested mode is for master interrupt controllers. For the master interrupt controller, each slave controller is a single interrupt input. Thus, the master controller cannot resolve the priority of the slave controller interrupt inputs. If a slave controller has an active, low priority interrupt that is nested in favor of a higher priority interrupt, the master inhibits the new slave interrupt request. This effectively disables

nesting of slave interrupts. In special-fully-nested mode, the master interrupt controller acts on all slave interrupt requests, which allows the slave to nest interrupts. The VAXmate workstation startup code disables the special-fully-nested mode.

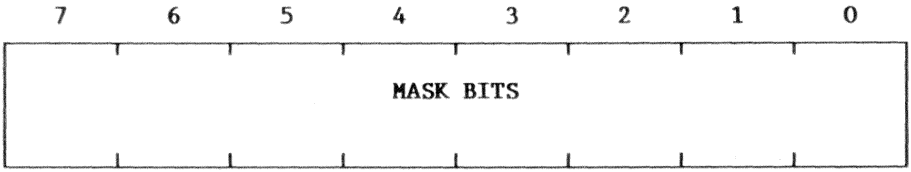
For the master ICW4 and the slave ICW4, use 01H.

Operation Command Words

The interrupt controller provides three operation command words (1-3) that are programmed after the initialization sequence is complete. The operation command words select various modes or operations as follows:

- Read or write the interrupt mask register
- Accept specific or nonspecific end-of-interrupt commands
- Enable or disable various automatic priority rotation schemes
- Set a specific priority level
- Set or reset the special mask
- Read poll data
- Read the interrupt request register
- Read the in-service register

Operation Command Word 1 (0021H/00A1H)



Bit	R/W	Description
7-0	R/W	Interrupt mask register bits 0 = Corresponding interrupt inputs are unmasked (enabled) 1 = Corresponding interrupt inputs are masked (disabled)

OCW1 reads or writes the interrupt mask register (IMR). Each bit in the IMR enables or disables the corresponding interrupt input.

Operation Command Word 2 (0020H/00A0H)

7	6	5	4	3	2	1	0
ROTATE	SL	EOI	0	0	INTERRUPT LEVEL		

Bit	R/W	Description
7-5	W	ROTATE/SL/EOI 000 = Disable rotation in automatic EOI mode 001 = Nonspecific end-of-interrupt (EOI) 010 = No operation 011 = Specific end-of-interrupt 100 = Enable priority rotation in automatic EOI mode 101 = Rotate priority on nonspecific EOI 110 = Rotate priority to specific interrupt input 111 = Rotate priority on specific EOI
4	W	Always 0 For values written to port 0, this bit distinguishes operational command words 2 and 3 from an ICW1. See Table 3-3.
3	W	Always 0 This bit distinguishes OCW2 from OCW3. See Table 3-3.
2-0	W	INTERRUPT LEVEL For interrupt-specific operations, these bits contain the interrupt input number (0-7) to act on. For nonspecific operations, these bits are ignored.

OCW2 issues an end-of-interrupt command or sets a priority rotation mode. Some OCW2 operations are nonspecific. (They act on the interrupt input that has the highest priority, whichever one that may be.) Non-specific commands do not use INTERRUPT LEVEL (OCW2 bits 2-0). Specific OCW2 operations require the interrupt input number (0-7) in INTERRUPT LEVEL.

For the master OCW2 and the slave OCW2, use 20H (nonspecific EOI command).

Priority Rotation

In nonspecific or automatic EOI mode, priority rotation has the effect of assigning equal priorities to all interrupt inputs. On receipt of an EOI command, the interrupt controller assumes that the active interrupt input with the highest priority is the interrupt just completed. The priority bits are rotated until the just completed interrupt has the lowest priority (7). If that interrupt input requires further service, it must wait until it is again the highest priority interrupt or until all interrupts of higher priority are inactive.

In Figure 3-1, interrupt inputs 2, 5, and 6 are requesting service and interrupt input 2 has a higher priority than interrupt inputs 5 and 6. After interrupt input 2 is serviced, the interrupt controller rotates the priority as shown in Figure 3-2. In Figure 3-2, interrupt input 3 has the highest priority, but it is inactive. Because interrupt input 5 has the highest priority of the active interrupts, it is the next interrupt input serviced.

Rotating priorities to a specific interrupt input is another method of priority rotation. In this method, the lowest priority is set, thereby fixing all other priorities. For example, if interrupt input 2 is programmed as the lowest priority, then interrupt input 3 becomes the highest. OCW2 bits 2-0 define the interrupt input number that is assigned the lowest priority. This method is not used in the VAXmate workstation startup code.

In-Service Bits

7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0

Priority Status

7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0

Figure 3-1 Priority Before Rotation

In-Service Bits

7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0

Priority Status

7	6	5	4	3	2	1	0
4	3	2	1	0	7	6	5

Figure 3-2 Priority After Rotation

Operation Command Word 3 (0020H/00A0H)

7	6	5	4	3	2	1	0
0	ENABLE SPECIAL MASK MODE	SPECIAL MASK MODE	0	1	POLL	READ IR REG	READ IS REG

Bit	R/W	Description
7	W	Always 0
6-5	W	ENABLE SPECIAL MASK MODE/SPECIAL MASK MODE 00 = No action 01 = No action 10 = Disable special mask mode 11 = Enable special mask mode Some operations require that an interrupt service routine dynamically change the priority structure. Masking an interrupt input in the special mask inhibits that priority level and enables all other priority levels (lower and higher) that are unmasked. After enabling special mask mode, the special mask is read or written to the IMR.
4	W	Always 0 For values written to port 0, this bit distinguishes operational command words 2 and 3 from an ICW1. See Table 3-3.
3	W	Always 1 This bit distinguishes OCW3 from OCW2. See Table 3-3.
2-0	W	POLL/READ IR REG/READ IS REG 000 = No action 001 = No action 010 = Read the IRR. * 011 = Read the ISR. * 100 = Read the poll data. ** 101 = Read the poll data. ** 110 = Read the poll data. ** 111 = Read the poll data. **

* See Table 3-3 and the IRR/ISR description.
** See Table 3-3 and the poll command description.

For standard operation of the VAXmate workstation, neither the master nor the slave use OCW3.

Interrupt Request and In-Service Registers

Interrupt Request Register

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

In-Service Register

7	6	5	4	3	2	1	0
IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0

The Interrupt Request Register (IRR) and the In-Service Register (ISR) maintain the state of the interrupt controller. During the first interrupt acknowledge of the CPU interrupt acknowledge sequence, the IRR latches the state of the interrupt input lines. The internal output of the Interrupt Mask Register (IMR) gates the output of the IRR to the priority encoder. Assuming that one or more IRR bits are set (active) and unmasked (enabled), the priority encoder determines which one has the highest priority. During the second interrupt acknowledge, that IRR bit is strobed into the corresponding ISR bit, the edge sense circuitry for that interrupt input is reset, and the interrupt vector number is placed on the I/O data bus.

Because the interrupt controller can nest interrupts, the ISR can contain one, two, or more bits that are set. This shows that another interrupt was acknowledged before other interrupt processing was completed. A specific end-of-interrupt (EOI) clears the indicated ISR bit. A nonspecific EOI clears the highest-priority ISR bit.

If an IRR bit is set (active) and masked (disabled), unmasking (enabling) that active IRR bit creates an interrupt.

Poll Command

When issued, the poll command performs steps similar to those described in the IRR/ISR description. The poll command replaces the function of the CPU interrupt acknowledge sequence. Instead of placing the interrupt vector number on the I/O data bus, the poll command connects the output of the poll data register to the port 0 output buffer. The polling interrupt handler then reads the poll data to determine if an interrupt input is active and, if so, which one. To complete the interrupt sequence, the polling interrupt handler must issue an EOI.

Poll Data Register

7	6	5	4	3	2	1	0
INT ACTIVE FLAG	0	0	0	0	INTERRUPT INPUT NUMBER		

Bit		Description
7	R	INT ACTIVE FLAG - Interrupt active flag 0 = No active interrupt inputs 1 = At least one interrupt input is active
6-3	R	Always 0
2-0	R	INTERRUPT INPUT NUMBER If bit 7 equals 1, these bits contain the interrupt input number (0-7) of the highest priority interrupt input that is active. If bit 7 equals 0, these bits have no meaning.

Interrupt Sequence

The following list describes interrupt processing. Each item in the list describes a system state or event. After a discussion of the state or event, the description indicates the next state or event. For the following interrupt processing description, it is assumed that the interrupt controllers are initialized as previously described. Later, Figure 3-3 shows the same logic in the form of a flow chart.

1. Until one or more interrupt controller input lines become active, the controller is idle. If one or more inputs are active, go to 2.
2. If any of the newly active inputs are unmasked (enabled), go to 4. Otherwise, go to 3.
3. If other interrupt inputs are pending, go to 5. Otherwise, go to 1.
4. If no other interrupt inputs are pending, go to 7. Otherwise, go to 6.
5. If the controller is waiting for an end-of-interrupt command, go to 6. Otherwise, go to 7.
6. If any interrupt has a priority higher than the one being processed by the CPU, nest the interrupts and go to 7. Otherwise, go to 8.
7. The controller activates its interrupt output line and waits for an acknowledge signal from the CPU.

   If the interrupt controller input is a slave input, then the slave interrupt output line activates the master interrupt controller IRQ2 interrupt input. The master interrupt process starts at step 2. Eventually, the master IRQ2 input becomes the highest priority master interrupt that is active and the master controller arrives at this step. At that time, both controllers are waiting for the CPU acknowledge signal.

   In either case, the master interrupt controller activates its interrupt output line, which triggers an external latch. The external latch drives the CPU INTR input.

NOTE
This external latch, between the master interrupt controller interrupt output and the CPU INTR input, was incorporated due to an advisory on an 80286 CPU design flaw.

Disabling the CPU INTR input before disabling an interrupt controller input or initializing the interrupt controllers can leave the latch set. On reenabling the CPU INTR input, the latch could indicate an interrupt request when none exists.

To avoid this situation, disable the interrupt controller input or write the first master interrupt controller initialization command before disabling the CPU INTR input.

If the CPU INTR input is disabled, the interrupt controller continues to wait. If other interrupt controller inputs become active during this waiting period, go to 2. When the CPU INTR input is enabled, the CPU recognizes the interrupt request and responds with an acknowledge signal.

On receiving the acknowledge, the interrupt controller sets the highest priority bit in the in-service register and resets the corresponding bit in the interrupt request register. This allows the controller to recognize another interrupt request at that interrupt controller input.

The CPU issues a second acknowledge signal. When the master interrupt controller recognizes the second acknowledge signal, it determines whether the interrupt input source is a slave interrupt controller. If the interrupt input source is not a slave, the master controller places a preprogrammed 8-bit interrupt vector on the input/output (I/O) data bus. If the interrupt input source is a slave, the master controller places the slave address (master interrupt input number 0-7) on the cascade lines. When enabled by the slave address on the cascade lines, the slave places the preprogrammed 8-bit interrupt vector on the input/output (I/O) data bus. In either case, the CPU reads the 8-bit interrupt vector, stacks the current state and begins executing the interrupt handler that is pointed to by the contents of the interrupt vector.

8. The interrupt controller(s) are waiting for an end-of-interrupt (EOI) command. When a slave interrupt is processed, an EOI command is required by both the slave and the master.

   If an interrupt occurs during this waiting period, go to step 2. When the CPU writes the end-of-interrupt command, go to step 1.

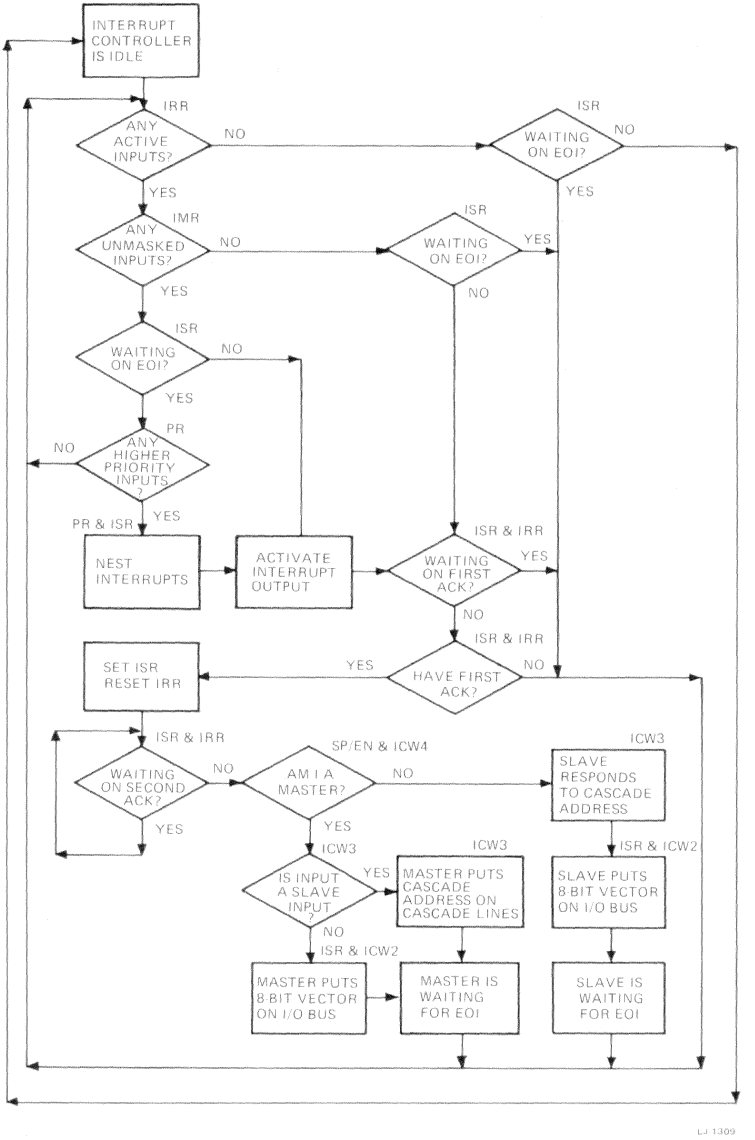


Figure 3-3 Interrupt Sequence

Programming Example

The following programming examples demonstrate:

- Initializing a master or slave 8259A peripheral interrupt controller (PIC)
- Programming the PIC interrupt mask register
- Issuing end-of-interrupt commands to a master or slave PIC

The example provides routines as described in the following:

pic_init	Initializes the master and slave PICs.
imask	Masks or unmasks the specified bit in the interrupt mask register.
eoi	Issues an end-of-interrupt command to the appropriate PICs.

CAUTION
Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

Constant Values and Data Structures

The constant value *NPIC* defines the number of peripheral interrupt controllers in the VAXmate workstation.

The constant value *EOI* defines the bit value that must be issued to OCW2 to establish an end-of-interrupt condition.

The structure type *PIC* defines the input/output ports of the 8259A peripheral interrupt controller (PIC). These two ports access the PIC registers. The bit values written to registers and the read or write sequence determine the accessed register.

The structure type *PIC_DAT* defines the type of data required to initialize a PIC.

Initialization Data

The array of structures *allpics* provides the actual data for initializing the master and slave PICs. Later, references to the PIC number refer to the position of an element in the array *allpics*.

```
/*************************************************************************/
/* define constants and structures used in 8259 PIC example              */
/*************************************************************************/

#define NPIC 2                                  /* number of pics in system */
#define EOI  0x20                               /* bit value of EOI command */

typedef struct                                  /* define pic I/O structure */
{
  unsigned char port0;                          /* when address line A0 = 0 */
  unsigned char port1;                          /* when address line A0 = 1 */
} PIC;

typedef struct
{
  PIC  *base;                                    /* base I/O address of pic */
  char icw2;                                    /* modulo 8 base int vector */
  char icw3;                                  /* ir has a slave or slave id */
  char icw4;                                          /* icw4 mode data */
} PIC_DAT;



/*************************************************************************/
/* define pic initialization data                                        */
/*************************************************************************/

PIC_DAT allpics[NPIC] =                           /* device data tables */
{
  { (PIC *)0x0020, 0x08, 0x04, 0x01 },          /* pic 0 is the master  */
  { (PIC *)0x00a0, 0x70, 0x02, 0x01 },            /* pic 1 is the slave */
};
```

Initializing the Peripheral Interrupt Controller

The function *pic_init* initializes the master and slave PICs. Because the ROM BIOS startup sequence initializes the peripheral interrupt controllers (PIC), initialization of the PICs is not normally required.

Because the initialization sequence clears the interrupt mask register, the CPU interrupt flag is cleared after the initialization is started and before the initialization is complete. Thus, no interrupts are pending when the initialization is started and the CPU will not respond to any interrupts that become active during the initialization sequence.

The first two instructions write a value to port 0 of the indicated PIC. The value of bit 4 is key to this operation. Writing a value to port 0, with bit 4 set, selects the ICW1 register and indicates a reset sequence. The PIC stores the remaining ICW1 bits in the ICW1 register, which starts the initialization sequence.

- Bit 0 is set, indicating that the initialization sequence includes ICW4.
- Bit 1 is clear, indicating that the addressed PIC is involved in a cascade (master/slave) arrangement and that ICW3 must be written during the initialization sequence. That is, the PIC is not operating in a standalone environment.
- Bit 3 is clear, indicating that the PIC is operating in edge-triggered mode.
- All other ICW1 bits apply to the 8085 mode of operation and are not used.

Because two PICs must be initialized, the rest of the initialization sequence is performed within a *for* loop as follows:

1. The first instruction initializes a pointer to the required data.
2. The second instruction initializes a pointer to the port 0 I/O address.
3. The third instruction writes the base interrupt vector to port 1. Because this is the second value written in the sequence, the PIC routes the value to ICW2.

   This base interrupt vector refers to the interrupt vector for interrupt input zero. To generate a unique interrupt vector number for each interrupt input, the PIC ORs the interrupt input number (0-7) and the base interrupt vector number. Therefore, the base interrupt vector must be modulo 8.
4. The fourth instruction is dependent upon whether the PIC is a master or a slave. Because this is the third value written in the sequence and ICW1 indicated a cascade mode, the PIC routes the value to ICW3.

- If the PIC is a master, each bit set in the value indicates that the corresponding interrupt input is connected to a slave PIC. For the VAXmate, only bit 2 is set.
- If the PIC is a slave, the value is the slave identification. The slave identification is a value between 0 and 7 inclusive, and corresponds to the master interrupt input to which it is connected.

```
/*************************************************************************/
/* pic_init() - initialize master and slave pics                         */
/*************************************************************************/

void pic_init()                                    /* initialize all pics */
{

int i;                                          /* variable for loop control */
int intr_flg;                                        /* to hold CPU IF state */
register PIC_DAT *ppd;                               /* pointer to PIC data */
register PIC *pps;                              /* pointer PIC I/O structure */

  outp(allpics[0].base, 0x11);              /* write master ICW1 is cascade */
  outp(allpics[1].base, 0x11);              /* write slave  ICW1 is cascade */
  intr_flg = int_off();                          /* turn CPU interrupts off */
  for(i = 0; i < NPIC; i++)
  {
    ppd = &allpics[i];                          /* assign pointer to PIC data */
    pps = ppd->base;                           /* assign pointer to I/O ports */
    outp(&pps->port1, ppd->icw2);                                /* write ICW2 */
    outp(&pps->port1, ppd->icw3);                                /* write ICW3 */
    outp(&pps->port1, ppd->icw4);                                /* write ICW4 */
    outp(&pps->port1, 0xff);                          /* mask all interrupts */
  }
  int_on(intr_flg);                                /* turn CPU interrupts on */
}
```

5. The fifth instruction writes the ICW4 value to port 1.
   - Bit 0 determines the microprocessor family. In this case, it is set and indicates the 8086/80286 mode.
   - Bit 1 is clear, indicating that the interrupt handling routine issues an end-of-interrupt command after the interrupt is processed.
   - Bits 2 and 3 are clear, indicating the nonbuffered mode of operation. For the VAXmate, a permanent hardware connection determines the master/slave relationship.
   - Bit 4 is clear, indicating that the PIC is not in special-fully-nested mode.
   - All other bits are not used and are clear.

6. The sixth instruction masks (disables) all interrupts. The interrupt inputs must be unmasked before the PIC can generate an interrupt to the CPU.

To complete the initialization, the last instruction enables CPU interrupts. Because the PIC interrupt mask is cleared during initialization, it is possible that the PIC will recognize an active interrupt input between instructions 5 and 6. Before a PIC interrupt input is unmasked, an interrupt handler must be available and the appropriate interrupt vector initialized.

A PIC interrupt input that is not active long enough to be latched is considered a glitch. If a glitch occurs, the PIC generates an interrupt for IRQ7 (master) or IRQ15 (slave). To determine whether an interrupt for IRQ7 or IRQ15 is a glitch, test ISR bit 7 of the appropriate controller. If the ISR bit 7 is set, the interrupt is a valid interrupt. If the ISR bit 7 is clear, the interrupt is a glitch.

Issuing an End-of-Interrupt Command

In fully nested mode (default mode), the PIC processes the highest priority interrupt that is pending. When the PIC receives a nonspecific end-of-interrupt (EOI), it clears the highest priority bit that is set in the in-service register. Until no interrupts are pending, the PIC continues by processing the highest priority interrupt that is pending.

To allow the PIC to process the same interrupt or an interrupt of lower priority, the *eoi* function is called at the end of an interrupt handling sequence. The calling parameter indicates which PIC issued the interrupt. If an interrupt is issued by the slave PIC, an EOI must be issued to the slave and then to the master.

During interrupt processing, it is possible for a higher priority interrupt to become active. If this happens, the PIC attempts nesting the interrupts. For the PIC to nest interrupts, the CPU interrupt request input must be enabled. Otherwise, the CPU will not issue the required acknowledge sequence. During the interrupt processing, the CPU automatically stacks its current state and clears the interrupt enable flag. Because none of the interrupt handlers, in these examples, enable the CPU interrupt request input, nesting of interrupts is effectively disabled.

Masking Interrupts

The function *imask* masks or unmasks a bit in the interrupt mask register (OCW1). The calling parameters indicate the PIC number, the bit number (0-7), and whether the bit should be masked or unmasked.

```
/*************************************************************************/
/* eoi() - establish End-Of-Interrupt for pic(s)                         */
/*************************************************************************/

void eoi(pic)                                   /* send nonspecific EOI */

int pic;                                    /* which pic handled interrupt */

{

  outp(&(allpics[pic].base)->port0, EOI);     /* write eoi as indicated */
  if(pic)                                     /* was it the slave pic ? */
    outp(&(allpics[0].base)->port0, EOI);        /* write eoi to master */
}



/*************************************************************************/
/* imask() - mask or unmask desired bit in pic mask register             */
/*************************************************************************/

void imask(pic, bitno, enable)              /* set or clear bit in mask */
                                             /* register of desired pic */
int pic;                                                /* which pic ? */
int bitno;                                              /* which bit ? */
int enable;                                    /* enable or disable ? */

{

unsigned char current;                          /* current contents of MR */
unsigned char mask;                                /* the mask to write */
register PIC *pps;                           /* pointer PIC I/O structure */

  pps = allpics[pic].base;                 /* assign pointer to I/O ports */
  current = inp(&pps->port1);                        /* read current mask */
  mask = 1 << bitno;                                /* set up correct bit */
  if(enable) current &= ~mask;                           /* clear the bit */
  else current |= mask;                                     /* or set it */
  outp(&pps->port1, current);                /* write the resulting mask */
}
```

Chapter 4
DMA Controller

Overview

The direct-memory-access (DMA) controller is an Intel 8237A-5, programmable, DMA controller operating at 4 MHz. The DMA controller allows the direct transfer of 8-bit data between DMA-capable, input/output (I/O) devices and memory. The DMA controller has four, independent DMA channels. Table 4-1 lists the assignment of the four DMA request lines. Each channel has 16 address lines and an external 8-bit page register. Thus, each channel can transfer a maximum of 64 Kbytes anywhere in the 16 Mbyte address range.

The following list shows the operational modes and restrictions of the DMA controller.

Single transfer	This is the suggested mode of operation.
Block transfer	To prevent interference with DRAM refresh cycles, limit block transfers to 8 transfers per block.
Demand transfer	To prevent interference with the DRAM refresh cycle, limit demand transfers to 8 transfers per demand.
Cascade	As bus master, the slave DMA controller should release the bus after 10 μs.
Compressed timing	Compressed timing is not supported by the processor board hardware.
Memory to memory	Memory-to-memory transfers are not supported by the processor hardware.
Extended write cycle	The extended-write cycle does not provide sufficient data setup time. Use the normal DMA write cycle.

Table 4-1 DMA Request Line Assignments

Channel	Request Line
0	Available
1	Available
2	Diskette Controller
3	Available

Additional Source of Information

The following Intel Corporation document provides additional information:

- *Microsystem Components Handbook* (Publication Number 230843)

Operation

When the DMA controller receives a DMA request from a peripheral device, the DMA controller sends a hold request signal to the CPU. When the CPU responds with a hold acknowledge signal, the DMA controller takes control of the I/O data bus, the system address bus, and the control bus. The controller then generates a 16-bit memory address and activates the corresponding DMA acknowledge line, the I/O read or write line, and the memory read or write line. On seeing the DMA acknowledge, the DMA-capable I/O device transfers (reads or writes) the data on the data bus. Thus, the data is transferred directly between the I/O device and memory.

The DMA controller operates in two major cycles, idle and active. Each DMA cycle can assume seven, separate states. Each state is composed of one full, clock period. Table 4-2 describes the various controller states.

Table 4-2 DMA Controller States

State	Description
SI	This is the inactive state. No valid DMA requests are pending and the CPU can program the DMA controller.
S0	This is the first active state of DMA service. The controller has requested a CPU hold, but the CPU has not acknowledged a hold. Programming of the DMA controller can continue until the acknowledge is received.
S1-S4	These are the DMA working states.
SW	When more time is required to complete a transfer, wait states are inserted between S2 and S3, or S3 and S4.

Idle Cycle

When none of the I/O channels is requesting DMA service, the DMA controller enters the idle cycle and performs SI states. At each clock cycle in the idle cycle, the DMA controller samples the DMA request lines and the chip select line.

If a DMA request line becomes active, the DMA controller goes to the active state. Otherwise, if CPU has selected the DMA controller and the CPU has control of the bus, the CPU can read or write the DMA controller internal registers.

Active Cycle

When the DMA controller is in the idle cycle and a nonmasked channel requests DMA service, the controller issues a hold request to the CPU and enters the active cycle. The DMA service will then occur in one of the four following modes.

Single Transfer Mode

The DMA controller is programmed to perform only one transfer in this mode. After the transfer, the word count is decremented and the address is either decremented or incremented. When the word count goes from 0000H to FFFFH, a terminal count (TC) signal is generated, and will auto-initialize the channel to its original condition if it had been programmed to do so.

The ROM BIOS uses this mode for data transfers between the diskette controller and memory.

Block Transfer Mode

In this mode, the DMA controller is activated by a DMA request to continue making transfers until a TC (word count has reached FFFFH) or an external end-of-process (EOP) signal occurs. If the channel has been programmed for auto-initialization, the auto-initialization occurs at TC or EOP. This mode should be limited to eight transfers (assuming no additional wait states) to prevent interference with refresh cycles.

Demand Transfer Mode

The DMA controller performs transfers until a TC or external EOP occurs, or until there is no DMA request. Transfers may continue until the I/O device has exhausted its data capacity. Once the I/O device has caught up, DMA service is reestablished by means of a DMA request. The intermediate values of address and word count are stored in DMA controller internal registers between services while the CPU is running. At the end of the service, only an EOP can cause auto-initialization to occur. This mode should be limited to eight transfers per demand to prevent interference with refresh cycles.

Cascade Mode

This mode is used when DMA controllers are cascaded for system expansion. In this configuration, the initial controller determines the priority of the additional controllers. Each of the additional controllers establish priority within themselves and make the DMA request to the initial controller. The initial controller does not output any address or control signals, since they could conflict with the outputs of the added controller.

Data Transfers

The DMA controller can perform read, write, or verify operations in each transfer mode. Read transfers move data from memory to an I/O device; write transfers move data from an I/O device to memory; and verify transfers are pseudo data transfers. In verify mode, the controller operates as if in read or write mode, however the memory and I/O control lines are not active.

Memory-to-memory transfers are a special case of DMA transfer. Channel 0 is the source and channel 1 is the target. In memory-to-memory transfers, channel 0 uses one cycle to read the data byte and store it in the temporary register. On the following cycle, channel 1 writes the value in the temporary register to the target location.

Auto-Initialize

Restores the DMA channel to its original condition following an EOP. Auto-initialization is accomplished by restoring the original values of the Current Address and Current Word Count registers from the Base Address and Base Word Count registers. The CPU loads the current registers and base registers which do not change during the DMA service. When the channel is in auto-initialize mode, the mask bit is not set. After auto-initialization and a receipt of a DMA request, the channel can perform DMA service without CPU intervention.

Priority

The two types of priority, fixed and rotating, are defined as follows:

Fixed Priority	In fixed priority, the channels are placed in order based on the descending value of their assigned number. The assigned number range is from zero to three (0-3), with zero as the highest priority.
Rotating Priority	The channel being serviced is assigned lowest priority value, and all others rotate to the next higher value.

Address Generation

The eight, high-order address bits (15-8) are multiplexed on the I/O data lines. At the S1 state, the high-order 8-bits are output to an external latch and placed on the system address bus. The low-order bits are output directly from the DMA controller to the system address bus. For multiple transfers, such as block and demand transfers, the addresses are generated sequentially. The data in the external latch (high-order byte) can remain the same for many transfers, and have to be changed only when a borrow or carry takes place in the normal sequence of addresses. The controller executes S1 states only when updating of the high-order byte is required.

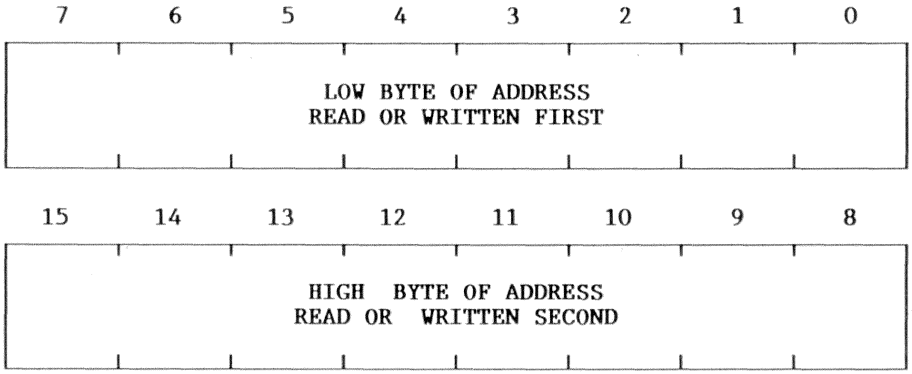
Table 4-3 DMA Controller and Page Register Address Map

Port	R/W	Channel	Register
0000H	W	0	Base and Current address
	R	0	Current address
0001H	W	0	Base and Current word count
	R	0	Current word count
0002H	W	1	Base and Current address
	R	1	Current address
0003H	W	1	Base and Current word count
	R	1	Current word count
0004H	W	2	Base and Current address
	R	2	Current address
0005H	W	2	Base and Current word count
	R	2	Current word count
0006H	W	3	Base and Current address
	R	3	Current address
0007H	W	3	Base and Current word count
	R	3	Current word count
0008H	W	-	Command
	R	-	Status
0009H	W	-	Request
000AH	W	-	Write single mask register bit
000BH	W	-	Mode register
000CH	W	-	Clear byte pointer flip/flop
	R	-	Temporary
000DH	W	-	Master clear
000EH	W	-	Clear mask register
000FH	W	-	Write all mask register bits
0080H	W	1	Channel 1 page register
0081H	W	2	Channel 2 page register
0082H	W	3	Channel 3 page register
0083H	W	0	Channel 0 page register

Registers

The DMA controller has 16 I/O ports to access 26 internal registers. Additionally, the DMA circuitry has four I/O ports to access four page registers. Table 4-3 lists the I/O ports and the registers accessed.

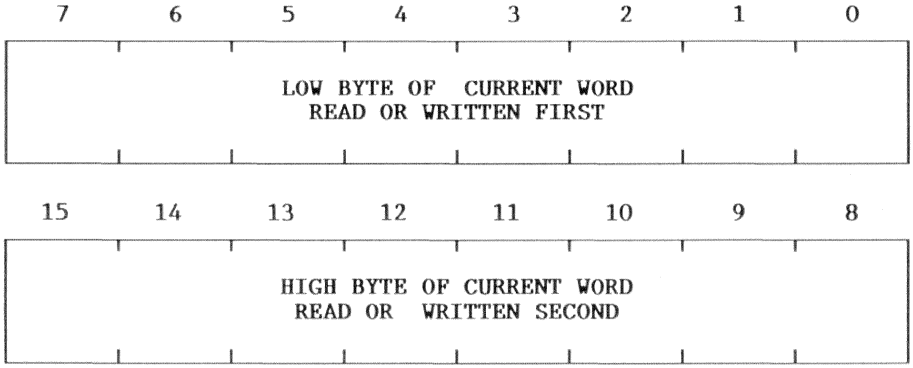
Base and Current Address Register (0000H/0002H/0004H/0006H)



Each DMA channel has a 16-bit base address register and a 16-bit current address register. The base address register contains the initial value. Writing a value to the base address register initializes the current address register to the same value. The current address register is incremented or decremented after each transfer. When the required number of transfers have occurred and if auto-initialize (see the mode register) is enabled, the current register is initialized from the base register.

Before performing a 16-bit read or write, clear the byte pointer flip/flop. To write a base register, write two, 8-bit bytes in succession to the same port. To read a current register, read two, 8-bit bytes in succession to the same port. In either case, the low byte is accessed first and then the high byte.

Base and Current Word Register (0001H/0003H/0005H/0007H)



Each DMA channel has a 16-bit base word count register and a 16-bit current word count register. The value written to this register determines the number of transfers performed. The number of transfers is the programmed value plus one. The current word count is decremented after each transfer. When the current word count is decremented below zero (FFFFH), a terminal count is generated. When the required number of transfers have occurred and if auto-initialize (see the mode register) is enabled, the current register is initialized from the base register.

Before performing a 16-bit read or write, clear the byte pointer flip/flop.
To write a base register, write two, 8-bit bytes in succession to the same port.
To read a current register, read two, 8-bit bytes in succession to the same port.
In either case, the low byte is accessed first and then the high byte.

Command Register (0008H)

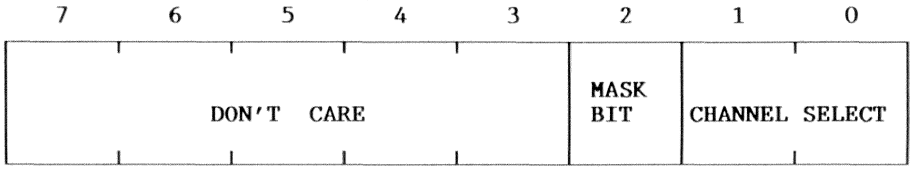
7	6	5	4	3	2	1	0
DACK SENSE	DREQ SENSE	WRITE SELECT	PR	TIMING	CE	CHANNEL 0 ADDRESS HOLD	MEMORY TO MEMORY

Bit	R/W	Description
7	W	DACK SENSE - DMA Acknowledge Sense 0 = DACK sense active low 1 = DACK sense active high
6	W	DREQ SENSE - DMA Request Sense 0 = DREQ sense active high 1 = DREQ sense active low
5	W	WRITE SELECT 0 = Late write selected 1 = Extended write selected For the VAXmate workstation, the extended write mode does not provide an adequate write cycle. Use only the late write mode. If bit 3 equals 1 (compressed mode), bit 5 is a don't care value. However, the VAXmate workstation is not capable of using compressed mode.
4	W	PR - Priority 0 = Fixed priority 0 (highest), 1, 2, and 3 (lowest) 1 = Rotating priority Initially, the priority is the same order as in fixed priority. In the rotating priority scheme, the currently serviced DMA channel becomes the lowest priority channel. However, the channels always maintain their priority in numeric order. That is, the priority decreases as the channel number increases and wraps between channels 3 and 0.

Bit	R/W	Description (Command Register - cont.)
3	W	TIMING 0 = Normal read/write timing - A read/write cycle requires a minimum of three clock cycles and is subject to wait states. The VAXmate workstation uses this mode. 1 = Compressed read/write timing - A read/write cycle occurs in two clock cycles. The VAXmate workstation is not capable of using compressed mode. If bit 0 equals 1 (memory-to-memory enabled), bit 3 (timing) is a don't care value.
2	W	CE - Controller Enable 0 = Controller disabled 1 = Controller enabled
1	W	CHANNEL 0 ADDRESS HOLD 0 = Disable channel 0 address hold 1 = Enable channel 0 address hold Channel 0 address hold causes the DMA controller to copy a single byte to the specified number of destination bytes. If bit 0 equals 0 (memory-to-memory disabled), bit 1 (channel 0 address hold) is a don't care value.
0	W	MEMORY-TO-MEMORY 0 = Memory-to-memory transfers disabled 1 = Memory-to-memory transfers enabled The VAXmate workstation does not support memory-to-memory transfers.

This 8-bit register controls the operation of the DMA controller. It is cleared by a hardware reset or a master clear instruction.

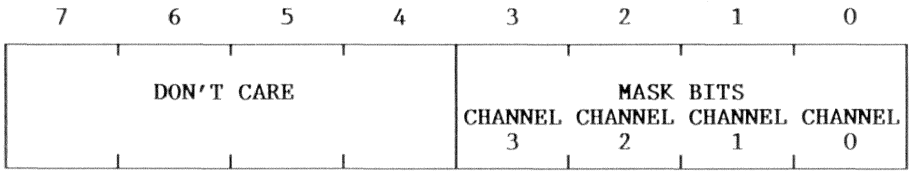
Write Single Mask Bit (000AH)



Bit	R/W	Description
7-3	W	DON'T CARE (any value)
2	W	MASK BIT 0 = Enable the selected channel 1 = Disable the selected channel
1-0	W	CHANNEL SELECT 00 = Select channel 0 mask bit 01 = Select channel 1 mask bit 10 = Select channel 2 mask bit 11 = Select channel 3 mask bit

Each channel has a mask bit, which can be set to disable the incoming DMA request. These bits are set if their associated channel produces an EOP and auto-initialize is not enabled.

Write All Mask Bits (000FH)



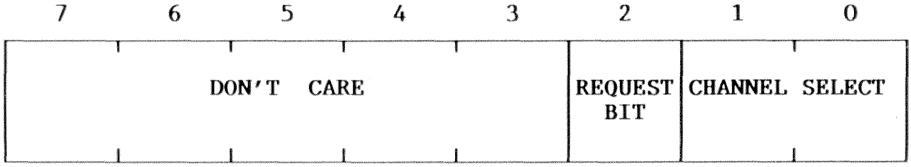
Bit	R/W	Description
7-4	W	DON'T CARE (any value)
3-0	W	MASK BITS 0 = Enable the indicated channel (CHANNEL 3-0) 1 = Disable the indicated channel (CHANNEL 3-0)

Mode Register (000BH)

7	6	5	4	3	2	1	0
OPERATION MODE		INCR/ DECR SELECT	AUTO INIT	TRANSFER TYPE		CHANNEL SELECT	

Bit	R/W	Description
7-6	W	OPERATION MODE 00 = Demand mode 01 = Single mode 10 = Block mode 11 = Cascade mode
5	W	INCR/DECR SELECT - Increment/Decrement selection 0 = Increment selected 1 = Decrement selected
4	W	AUTO INIT - Auto-initialization enable 0 = Disable auto-initialization 1 = Enable auto-initialization
3-2	W	TRANSFER TYPE 00 = Verify 01 = Write 10 = Read 11 = Invalid value A read transfer moves data from memory to the I/O device. A write transfer moves data from the I/O device to memory. That is, the orientation is from the I/O device, not the CPU. If bits 7-6 equal 11, then the transfer type is a don't care value.
1-0	W	CHANNEL SELECT 00 = Channel 0 selected 01 = Channel 1 selected 10 = Channel 2 selected 11 = Channel 3 selected Each DMA channel has a 6-bit mode register. Register selection is determined by bits 1 and 0.

Request Register (0009H)



Bit	R/W	Description
7-3	W	DON'T CARE (any value)
2	W	REQUEST BIT 0 = Reset the indicated request bit 1 = Set the indicated request bit
1-0	W	CHANNEL SELECT 00 = Channel 0 01 = Channel 1 10 = Channel 2 11 = Channel 3

The DMA controller responds to requests for DMA service from both software and the DMA request signal. Each channel has a request bit that can be set or reset as determined by the Request register. These bits are not maskable and are subject to prioritization.

Status Register (0008H)

7	6	5	4	3	2	1	0
DMA REQUEST PENDING CHANNEL 3	DMA REQUEST PENDING CHANNEL 2	DMA REQUEST PENDING CHANNEL 1	DMA REQUEST PENDING CHANNEL 0	TERMINAL COUNT REACHED CHANNEL 3	TERMINAL COUNT REACHED CHANNEL 2	TERMINAL COUNT REACHED CHANNEL 1	TERMINAL COUNT REACHED CHANNEL 0

Bit	R/W	Description
7-4	R	DMA REQUEST PENDING 0 = Indicated channel does not have a request pending (CHANNEL 3-0) 1 = Indicated channel has a request pending (CHANNEL 3-0)
3-0	R	TERMINAL COUNT REACHED 0 = Indicated channel has not reached the terminal count (CHANNEL 3-0) 1 = Indicated channel has reached the terminal count or external EOP applied (CHANNEL 3-0)

Temporary Register (000CH)

7	6	5	4	3	2	1	0
DATA							

Bit	R/W	Description
7-0	R	Last data byte transferred in a memory-to-memory transfer

Between the read and write cycles of a memory-to-memory transfer, the DMA controller stores the source byte in this register. This register is cleared by a hardware reset or a master clear.

Programming Example

The following programming example demonstrates:

- Initializing the 8237A DMA controller
- Enabling and disabling a channel
- Preparing a channel for data transfer

The example provides routines as described in the following list:

dma_init	Resets the DMA controller.
dma_open	Enables the indicated DMA channel.
dma_transfer	Prepares the indicated channel for data transfer.
dma_close	Disables the indicated channel.

> **CAUTION**
> Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

Constant Values

The constant values *DMA_PAGE0* through *DMA_PAGE3* define the I/O address of the indicated page register. The values *CHANNEL0* through *CHANNEL1* define the channel select bit values for the mode, mask, and request registers. The values *MTM_ENA* through *BIT_SET* define the bit values for various conditions of the command, status, mode, and request registers.

```
/*************************************************************************/
/* define constants used in 8237 DMA example                             */
/*************************************************************************/

#define DMA_PAGE0 0x83              /* DMA page register 0 I/O address */
#define DMA_PAGE1 0x80              /* DMA page register 1 I/O address */
#define DMA_PAGE2 0x81              /* DMA page register 2 I/O address */
#define DMA_PAGE3 0x82              /* DMA page register 3 I/O address */

#define CHANNEL0 0x00                   /* select channel 0 bit value */
#define CHANNEL1 0x01                   /* select channel 1 bit value */
#define CHANNEL2 0x02                   /* select channel 2 bit value */
#define CHANNEL3 0x03                   /* select channel 3 bit value */
```

```
/* command register bit definitions */
#define MTM_ENA   0x01                    /* Memory-to-memory enable */
#define HOLD_ENA  0x02                /* channel 0 hold address enable */
#define DMA_DIS   0x04                     /* dma controller disable */
#define C_TIME    0x08                          /* compressed timing */
#define ROT_PRI   0x10                          /* rotating priority */
#define EXTD_WR   0x20                             /* extended write */
#define DREQ_LO   0x40                       /* DREQ active when low */
#define DACK_HI   0x80                      /* DACK active when high */

/* status register bit definitions */
#define CH0_TC    0x01          /* channel 0 has reached terminal count */
#define CH1_TC    0x02          /* channel 2 has reached terminal count */
#define CH2_TC    0x04          /* channel 3 has reached terminal count */
#define CH3_TC    0x08          /* channel 4 has reached terminal count */
#define CH0_REQ   0x10                /* channel 0 requesting service */
#define CH1_REQ   0x20                /* channel 1 requesting service */
#define CH2_REQ   0x40                /* channel 2 requesting service */
#define CH3_REQ   0x80                /* channel 3 requesting service */

/* mode register bit definitions */
#define TRAN_VR   0x00                             /* verify transfer */
#define TRAN_WR   0x04                              /* write transfer */
#define TRAN_RD   0x08                               /* read transfer */
#define AUTO_IE   0x10                     /* auto-initialize enable */
#define ADR_DEC   0x20                          /* address decrement */
#define MODE_DM   0x00                                /* demand mode */
#define MODE_SI   0x40                                /* single mode */
#define MODE_BK   0x80                                 /* block mode */
#define MODE_CS   0xC0                               /* cascade mode */

/* request register bits */
#define BIT_SET 0x04                        /* set selected mask bit */
```

Data Structures

The structure *DMA_CHANNEL* resembles the I/O space of the address and word count registers for a channel. Multiple instances of this structure are used in the declaration of the *DMA_CONTROLLER* structure. The *DMA_CONTROLLER* structure defines the I/O space of DMA controller internal registers. The value *DMA_BASE* defines the base address for referencing the structure *DMA_CONTROLLER*.

```
/*************************************************************************/
/* declare structures used in 8237 DMA example                           */
/*************************************************************************/

typedef struct                      /* define dma channel I/O structure */
{
  unsigned char bc_addr;    /* write base address, read current address */
  unsigned char bc_word;         /* write base word, read current word */
} DMA_CHANNEL;

typedef struct                    /* define dma controller I/O structure */
{
  DMA_CHANNEL ch0;                              /* channel zero registers */
  DMA_CHANNEL ch1;                               /* channel one registers */
  DMA_CHANNEL ch2;                               /* channel two registers */
  DMA_CHANNEL ch3;                             /* channel three registers */
  unsigned char csr;                        /* write control, read status */
  unsigned char req;                             /* write request register */
  unsigned char wsmb;                             /* write single mask bit */
  unsigned char mode;                               /* write mode register */
  unsigned char temp;/* write clears byte pointer flip-flop, read temp */
  unsigned char master_clr;                    /* write master clear/reset */
  unsigned char clr_mask;                           /* clear all mask bits */
  unsigned char wr_mask;                            /* write all mask bits */
} DMA_CONTROLLER;

#define DMA_BASE (DMA_CONTROLLER *)0x0000                /* base address */
```

Initializing the DMA Controller

The DMA controller is initialized by issuing a MASTER CLEAR instruction. This clears all bits in the command register and effectively disables the controller. The second instruction, which explicitly clears the control register, ensures that the controller is disabled.

```
/*********************************************************************/
/* dma_init() - initialize the 8237 DMA controller                   */
/*********************************************************************/

dma_init()
{
DMA_CONTROLLER *pdc = DMA_BASE;              /* point to DMA controller */

  outp(&pdc->master_clr, 0);                    /* reset DMA controller */
  outp(&pdc->csr, 0);                 /* all command register bits to 0 */
}
```

Opening a DMA Channel

The *dma_open* function assumes that the channel is currently disabled. It writes valid values to the registers that control the indicated channel.

For this C compiler, offset 0 in the data segment is used only for monitoring NULL pointers. With a zero word count, an inadvertent data transfer can move only one byte before expiring.

The last instruction enables the indicated channel.

```
/*************************************************************************/
/* dma_open() - open a DMA channel                                       */
/*************************************************************************/

dma_open(channel)

int channel;                                /* which DMA channel to open */

{
DMA_CONTROLLER *pdc = DMA_BASE;              /* point to DMA controller */
DMA_CHANNEL *pch;                      /* pointer to a channel structure */
int i;                                                  /* loop control */

  for(i = channel, pch = &pdc->ch0; i; i--)  /* discover which channel */
    pch++;                                    /* point to next channel */
  outp(&pdc->mode, channel);    /* clear mode register for this channel */
  outp(&pdc->req, channel);  /* clear channel request for this channel */
  outp(&pdc->temp, 0);                    /* clear first/last flip-flop */
  outp(&pch->bc_addr, 0);                          /* write 0 to low byte */
  outp(&pch->bc_addr, 0);                         /* write 0 to high byte */
  outp(&pdc->temp, 0);                    /* clear first/last flip-flop */
  outp(&pch->bc_word, 0);                          /* write 0 to low byte */
  outp(&pch->bc_word, 0);                         /* write 0 to high byte */
  outp(&pdc->wsmb, channel);          /* clear mask bit for this channel */
}
```

Preparing a Channel for Data Transfer

The *dma_transfer* function prepares a channel for data transfer. Next, the function disables the channel. It then initializes the page, address, word count, and mode registers.

> **NOTE**
> Before writing a 16-bit register, the byte pointer flip/flop must be cleared. This sequence loads the two sequential bytes in the correct locations. Because interrupt processing could disrupt the process, the *dma_transfer* function disables CPU interrupts before clearing the byte pointer flip/flop. Interrupts are not enabled until after the 16-bit registers have been written.

```
/*************************************************************************/
/* dma_transfer() - set parameters for a DMA transfer                    */
/*************************************************************************/

dma_transfer(channel, page_val, addr, count, ttype)

int channel;                          /* transfer on which DMA channel ? */
int page_val;                               /* page register contents */
unsigned char *addr;                              /* transfer address */
unsigned int count;                             /* count to transfer */
int ttype;                                         /* transfer type */

{
DMA_CONTROLLER *pdc = DMA_BASE;              /* point to DMA controller */
DMA_CHANNEL *pch;                     /* pointer to a channel structure */
unsigned int page_reg;                  /* which page register to write */
int ch_mode;                                             /* channels mode */
int intr_flag;                                  /* to hold CPU IF state */

  switch(channel)                                        /* which channel ? */
  {
    case 0:                                                 /* channel 0 ? */
      pch = &pdc->ch0;                    /* point to channel 0 registers */
      page_reg = DMA_PAGE0;                 /* set page register address */
      ch_mode = 0;                /* auto-initialize & increment/decrement */
      break;
```

```
    case 1:                                              /* channel 1 ? */
      pch = &pdc->ch1;                  /* point to channel 1 registers */
      page_reg = DMA_PAGE1;               /* set page register address */
      ch_mode = 0;              /* auto-initialize & increment/decrement */
      break;

    case 2:                                              /* channel 2 ? */
      pch = &pdc->ch2;                  /* point to channel 2 registers */
      page_reg = DMA_PAGE2;               /* set page register address */
      ch_mode = MODE_SI;        /* auto-initialize & increment/decrement */
      break;

    case 3:                                              /* channel 3 ? */
      pch = &pdc->ch3;                  /* point to channel 3 registers */
      page_reg = DMA_PAGE3;               /* set page register address */
      ch_mode = 0;              /* auto-initialize & increment/decrement */
      break;
  }
  outp(&pdc->wsmb, BIT_SET | channel);/* set mask bit for this channel */
  outp(&pdc->req, channel);  /* clear channel request for this channel */
  outp(&pdc->mode, ttype | ch_mode | channel);     /* set mode register */
  intr_flag = int_off();                          /* no interrupts please */
  outp(&pdc->temp, 0);                      /* clear first/last flip-flop */
  outp(&pch->bc_addr, (unsigned int)addr & 0xff);     /* write low byte */
  outp(&pch->bc_addr, (unsigned int)addr >> 8);      /* write high byte */
  outp(&pch->bc_word, count & 0xff);                  /* write low byte */
  outp(&pch->bc_word, count >> 8);                   /* write high byte */
  int_on(intr_flag);                                /* allow interrupts */
  outp(page_reg, page_val);                     /* write the page register */
  outp(&pdc->wsmb, channel);          /* clear mask bit for this channel */
}
```

Disabling a DMA Channel

The *dma_close* function closes the channel by masking (disabling) that channel's request input line.

```
/*************************************************************************/
/* dma_close() - close a DMA channel                                     */
/*************************************************************************/

dma_close(channel)

unsigned char channel;                    /* which DMA channel to close */

{
DMA_CONTROLLER *pdc = DMA_BASE;             /* point to DMA controller */

  outp(&pdc->wsmb, BIT_SET | channel);/* set mask bit for this channel */
}
```

Chapter 5
Real-Time Clock and CMOS RAM

Overview

The VAXmate processor board contains an MC146818 real-time clock. The real-time clock has the following features:

- Time-of-day clock with alarm and 100-year calendar
- Counts seconds, minutes, and hours of the day
- Counts days of the week, days of the month, month, and year with automatic end-of-month and leap year recognition
- Binary or binary-coded-decimal (BCD) representation of date, time, and alarm (the ROM BIOS and MS-DOS use BCD).
- 24-hour clock or 12-hour clock with a.m./p.m. indication
- Daylight savings time option
- Internal time base and oscillator
- External time base 32.768 KHz crystal
- 64 byte, low-power, static RAM (14 bytes of registers and 50 bytes of general purpose RAM)
- Square wave generator
- Programmable interrupts
  - Time-of-day alarm, once-per-second to once-per-day
  - Periodic interrupt rates from 30.5 μs to 500 ms
  - End-of-update interrupt

Additional Source of Information

The following Motorola Inc. document provides additional information on programming the real-time clock.

- *8-Bit Microprocessor & Peripheral Data*

Battery-Backup Considerations

To keep time and maintain RAM when system power is off, the real-time clock requires a battery-backup source. The two lithium batteries in the VAXmate expansion box provide the only battery power source.

> NOTE
> The lithium battery used in the VAXmate expansion box has an operational life expectancy of 6 years and a shelf life of 10 years.

Addressing the Real-Time Clock

The real-time clock (RTC) is addressed by the contents of an 8-bit latch at I/O port 0070H and the RTC data is read or written through I/O port 0071H.

> NOTE
> The RTC address latch is write only. Bit 7 of the RTC address latch (I/O port 0070H) is the nonmaskable interrupt (NMI) mask register. If bit 7 equals zero, the NMI is enabled. Otherwise, the NMI is disabled. For more information about the nonmaskable interrupt, see Chapters 3 and 15.

The RTC dedicates the first 14 bytes of RAM (00H through 0DH) as registers for the real-time clock functions. The remaining 50 bytes of RAM (0EH through 3FH) are not dedicated to the RTC. Table 5-1 describes the RTC address map.

Table 5-1 Real-Time Clock Address Map

Latch Value	R/W	Location Accessed
00H	R/W	Seconds register
01H	R/W	Seconds alarm register
02H	R/W	Minutes register
03H	R/W	Minutes alarm register
04H	R/W	Hours register
05H	R/W	Hours alarm
06H	R/W	Day-of-week register
07H	R/W	Day-of-month register
08H	R/W	Month register
09H	R/W	Year register
0AH	R/W	Register A
0BH	R/W	Register B
0CH	R/W	Register C
0DH	R/W	Register D
0EH-3FH	R/W	Remaining 50 bytes of RTC RAM *

* See the definition of the structure RTC in the programming example.

Real-Time Clock Registers

The real-time clock (RTC) has two types of registers:

- Data (locations 00H through 09H)
- Control and status (locations 0AH through 0DH)

Data registers are valid only when the RTC is not updating. During clock updates, the RTC disconnects the data registers from the RTC bus. The specifics of data register processing are discussed later.

The control and status registers are available at all times.

Register A

Addressing - Write 0AH to address latch at 0070H.
Data - Read or write data at address 0071H.

7	6	5	4	3	2	1	0
UIP	DIVIDER SELECTION BITS			RATE SELECTION BITS			
	DS2	DS1	DS0	RS3	RS2	RS1	RS0

Bit	R/W	Description
7	R/W	UIP - Update In Progress 0 = For all time bases, at least 244 μs remain before the update cycle begins. The data registers are available for reading. 1 = Update cycle is in progress or begins in less then 244 μs. The UIP bit is a read-only bit. For the 32.768 KHz time base, the update cycle time is 1984 μs. Writing a 1 to the Register B SET bit inhibits the update cycle and clears the UIP status bit. A hardware reset does not modify the UIP bit.
6-4	R/W	DIVIDER SELECTION BITS These bits identify the time base to use. Writing 111 to these bits resets the divider. One second after removing the divider reset, the first update cycle begins. For the VAXmate workstation time base of 32.768 KHz, set these bits to 010. A hardware reset does not modify the DIVIDER SELECTION bits.
3-0	R/W	RATE SELECTION BITS These bits select one of 15 taps on a 22-stage divider or disable the divider. Table 5-2 shows the bit values for the possible interrupt rates. A hardware reset does not modify the RATE SELECTION bits. On powerup, the ROM BIOS sets these bits to 0.

Table 5-2 Rate Selection Bits

RS3	RS2	RS1	RS0	Periodic Interrupt Rate
0	0	0	0	None (divider disabled)
0	0	0	1	3.90625 μs
0	0	1	0	7.8125 μs
0	0	1	1	122.070 μs
0	1	0	0	244.141 μs
0	1	0	1	488.281 μs
0	1	1	0	976.562 μs
0	1	1	1	1.953125 ms
1	0	0	0	3.90625 ms
1	0	0	1	7.8125 ms
1	0	1	0	15.625 ms
1	0	1	1	31.250 ms
1	1	0	0	62.5 ms
1	1	0	1	125.0 ms
1	1	1	0	250.0 ms
1	1	1	1	500.0 ms

Register B

Addressing - Write 0BH to address latch at 0070H.
Data - Read or write data at address 0071H.

7	6	5	4	3	2	1	0
SET	PIE	AIE	UIE	SQWE	DM	HM	DSE

Bit	R/W	Description
7	R/W	SET 0 = Allow update cycles to occur once per second 1 = Abort any update cycle in progress and inhibit update cycles until cleared. (This allows initialization of the date, time, and alarm registers.) A hardware reset does not modify the SET bit.
6	R/W	PIE - Periodic Interrupt Enable 0 = Disable periodic interrupts (default value) 1 = Enable periodic interrupts at the rate specified by RS3-RS0 in Register A A hardware reset clears the PIE bit to 0.
5	R/W	AIE - Alarm Interrupt Enable 0 = Disable alarm interrupts (default value) 1 = Enable alarm interrupts. (The interrupt frequency depends on the contents of the alarm registers.) A hardware reset clears the AIE bit to 0.
4	R/W	UIE - Update-ended Interrupt Enable 0 = Disable the update-ended interrupt (default value) 1 = Enable the update-ended interrupt A hardware reset or setting the register B SET bit clears the UIE bit to 0.
3	R/W	SQWE - Square Wave Enable 0 = Disable the square-wave output (default value) 1 = Enable the square-wave output The square-wave output is not connected to anything, so the SQWE bit should always be written as 0. A hardware reset clears the SQWE bit to 0.

Bit	R/W	Description (Register B - cont.)
2	R/W	DM - Data Mode 0 = Binary-coded-decimal (BCD) data format used for date, time, and alarm registers 1 = Binary data format used for date, time, and alarm registers A hardware reset does not modify the DM bit. However, the ROM BIOS clears DM to 0.
1	R/W	HM - Hour Mode 0 = Hours register and hours alarm register use a 12-hour clock with a.m. or p.m. indication 1 = Hours register and hours alarm register use a 24-hour clock A hardware reset does not modify the HM bit. However, the ROM BIOS sets HM equal to 1.
0	R/W	DSE - Daylight Savings Enable 0 = Disable daylight savings 1 = Enable daylight savings. Daylight savings changes occur at 2 a.m. on the last Sunday in April and the last Sunday in October. A hardware reset does not modify the DSE bit. However, the ROM BIOS clears DSE to 0.

Register C

Addressing - Write 0CH to address latch at 0070H.
Data - Read or write data at address 0071H.

7	6	5	4	3	2	1	0
IRQF	PIF	AIF	UIF	0	0	0	0

Bit	R/W	Description
7	R/W	IRQF - Interrupt Request Flag When one or more of the following conditions are true, the RTC sets the IRQF bit to 1: PIF = PIE = 1 AIF = AIE = 1 UIF = UIE = 1
6	R/W	PIF - Periodic Interrupt Flag When register B, bit PIE equals 1, the PIF bit indicates the state of the periodic interrupt. If PIF equals 1, the RTC sets IRQF. Register A bits RS3-RS0 establish the rate of this interrupt.
5	R/W	AIF - Alarm Interrupt Flag When register B, bit AIE equals 1, the AIF indicates the state of the alarm interrupt. When AIF equals 1, the current time matches the alarm time and the RTC sets IRQF.
4	R/W	UIF - Update-ended Interrupt Flag When register B, bit AIE equals 1, the UIF bit indicates the state of the update-ended interrupt. At the end of each update cycle, the RTC sets this bit to 1 and sets IRQF.
3-0	R/W	Always 0

Resetting hardware or reading register C clears all bits in register C. Writing to Register B does not modify the bits in Register C.

Register D

Addressing - Write 0DH to address latch at 0070H.
Data - Read or write data at address 0071H.

7	6	5	4	3	2	1	0
VRT	0	0	0	0	0	0	0

Bit	R/W	Description
7	R/W	VRT - Valid RAM and Time 0 = Since the last time this register was read, the power-sense circuitry detected a loss of power to the RTC. The RTC registers and RAM contain invalid data. 1 = Since the last time this register was read, power to the RTC has remained stable. Reading this register sets the VRT bit. It is the only way to set the VRT bit. After setting the date, time, or alarm, read this register so that the VRT bit indicates that the registers are valid. A hardware reset does not modify the VRT bit.
6-0	R/W	Always 0

Real-Time Clock Data Registers

The real-time clock (RTC) formats the date and time in either binary or binary-coded-decimal (BCD). All data registers (00H through 09H) must use the same format. If the data format is changed, the data registers must be initialized in the new format. The ROM BIOS uses the BCD data format. Bit 2 of register B controls the format.

The HOUR MODE bit in Register B controls the range of the hour and hour alarm registers. When the HOUR MODE bit is set (1), the hour and hour alarm registers have the range 0-23. When the HOUR MODE bit is clear (0), the hour and hour alarm registers have the ranges 1-12 (a.m.) and 129-140 (p.m.).

The hours, minutes, and seconds alarm registers have an additional range of C0H-FFH. This is an alarm register don't care code. For more information, see the alarm description. Table 5-3 shows the format and ranges of the data registers.

Table 5-3 RTC Data Register Ranges

Latch Value	Register	Function	Binary Range	BCD Range
00H	Seconds	All modes	0-59	00H-59H
01H	Seconds Alarm	Specific time	0-59	00H-59H
		Each second	192-255	C0H-FFH
02H	Minutes	All modes	0-59	00H-59H
03H	Minutes Alarm	Specific time	0-59	00H-59H
		Each minute	192-255	C0H-FFH
04H	Hours	24-hour mode	0-23	00H-23H
		12-hour mode a.m.	1-12	01H-12H
		12-hour mode p.m.	129-140	81H-92H
05H	Hours Alarm	Specific time (24-hour mode)	0-23	00H-23H
		Specific time (12-hour mode a.m.)	1-12	01H-12H
		Specific time (12-hour mode p.m.)	129-140	81H-92H
		Each hour (all modes)	192-255	C0H-FFH
06H	Day-of-Week		1-7	01H-07H
07H	Day-of-Month		1-31	01H-31H
08H	Month		1-12	01H-12H
09H	Year		0-99	00H-99H

Alarms

During each real-time clock (RTC) update cycle, the RTC compares the hour, minute, and second registers to the corresponding alarm registers. If all of the time registers match all of the alarm registers, the RTC sets the register C AIF. If, when this occurs, the register B alarm interrupt enable (AIE) bit is enabled, the alarm interrupt triggers IRQ8.

An alarm register value in the range C0H-FFH is a don't care code. When an alarm register contains a don't care code, that alarm register matches any value in the corresponding time register.

Table 5-4 shows the eight different types of automatic alarm cycles provided by the real-time clock (RTC).

Table 5-4 RTC Automatic Alarm Cycles

Cycle Description	Hour Alarm	Minute Alarm	Second Alarm
Once per second every second	C0H-FFH	C0H-FFH	C0H-FFH
Once per second for a one-minute span every hour	C0H-FFH	Specified	C0H-FFH
Once per second for a one-minute span every 24 hours	Specified	Specified	C0H-FFH
Once per second for a one-hour span every 24 hours	Specified	C0H-FFH	C0H-FFH
Once per minute every minute	C0H-FFH	C0H-FFH	Specified
Once per minute for a one-hour span every 24 hours	Specified	C0H-FFH	Specified
Once per hour every hour	C0H-FFH	Specified	Specified
Once every 24 hours	Specified	Specified	Specified

Also, there is a nonautomatic way to use the alarm function. To use this method, set a specific alarm time. At each subsequent alarm interrupt, set the next specific alarm time.

Update Cycle

Once per second, the real-time clock (RTC) performs an update cycle. With a 32.768 KHz time base, the update cycle requires 1948 μs. The update cycle comprises the following steps:

- The RTC sets (1) the register A UIP bit.
- After 244 μs, the RTC disconnects the data registers from the external bus and connects them to the internal bus.
- The RTC increments the seconds register.
- The RTC checks for an overflow condition. If no overflow condition exists, the RTC goes to the next step. Otherwise, the RTC zeros the register and increments the next register in the series.
- The RTC compares the hour, minute, and seconds registers to the corresponding alarm registers. If a match occurs for all three registers, the RTC sets (1) the register C alarm flag (AIF).
- The RTC disconnects the data registers from the internal bus and connects them to the external bus.
- The RTC clears (0) the register A UIP bit.
- The RTC sets (1) the register C update-ended interrupt flag (UIF)

During an update cycle, the data registers are disconnected from the external bus. Therefore, while an update is in progress, reading or writing a data register produces invalid results. Use one of the following methods to avoid update cycles:

- Monitor the register A update-in-progress (UIP) bit. The update cycle begins 244 μs after the RTC sets the UIP bit. Thus, if the UIP bit is clear, the data registers will remain valid for at least 244 μs.
- Enable the update-ended interrupt. This interrupt occurs after every update cycle. The date and time registers remain valid for over 999 ms after the RTC sets the UIF. If the processor must handle an excessive amount of interrupts, the interrupt handler for the RTC should also monitor the UIP bit.
- Monitor the register C periodic interrupt flag (PIF). The periodic interrupt is synchronized with the update cycle. For any given periodic interrupt, there is a time after the interrupt when the data registers are valid. For a 32.768 KHz time base, use only the rates between 3.90625 ms and 500 ms. Use the following formula to calculate the valid time span:

  Time Span = 244 μs + (RATE / 2)

Interrupts

Periodic Interrupt

If the PIE bit is set (1), the periodic interrupt triggers IRQ8 at the rate specified by the RATE SELECT bits in register A.

Update-Ended Interrupt

If the UIE bit is set (1), the update-ended interrupt triggers IRQ8 once per second. The next RTC update cycle starts 1000 ms after the update-ended interrupt.

Alarm Interrupt

During each RTC update cycle, the RTC compares the hour, minute, and second registers to the corresponding alarm register. If the time and alarm registers match and if the AIE bit is set (1), the alarm interrupt triggers IRQ8.

Programming Example

The real-time clock (RTC) programming example demonstrates:

- Reading and writing the RTC registers and RAM
- Handling RTC interrupts
- Interpreting the data stored in the RTC RAM
- Calculating the checksum that ensures data integrity

The next programming example provides the following routines:

rd_rtc	Reads the indicated RTC register or RAM location
wr_rtc	Writes the indicated RTC register or RAM location
rtc_cksum	Returns the calculated RTC RAM checksum
btb	Returns the binary equivalent of a binary-coded decimal (BCD) value
bcd	Returns the binary-coded decimal (BCD) equivalent of a binary value
rd_date	Reads the date-related registers and stores the results in the indicated structure
rd_time	Reads the time-related registers and stores the results in the indicated structure
shw_date	Displays the current date at location 0,0
shw_time	Displays the current time at location 0,72
shw_ddtyp	Displays the diskette drive types
shw_hdtyp	Displays the hard disk drive types
rtc_int_hand	Handles hardware interrupts from the RTC
shw_hdw	Displays the hardware setup from RTC RAM
rtc_init	Initializes the RTC interrupt vector (70H) and the RTC alarm registers
rtc_rest	Restores the RTC interrupt vector (70H) and disables clock interrupts
rtc	Provides menu selection of the examples and executes the examples

CAUTION
Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the next programming example is limited to the context provided in this manual. No other use is intended.

Constant Values

The file (*kyb.h*) that is included defines constant values for function keys. See Chapter 8 for information about keyboard programming.

The file (*example.h*) that is included defines the structure type MESSAGE that is used to display the menu.

The constant values CKSUM_START and CKSUM_END define the start and end offsets of the RTC RAM that is under checksum control. If any value in this range is changed, a new checksum must be written to reflect this change.

The constant values UIP through VRT define bit values for registers A through D. The value DIVIDE_SEL defines the divider that divides the base input frequency for the internal RTC operation. This value is related to the VAXmate hardware design and should be considered a fixed value.

```
#include "kyb.h"                    /* reference function key constants */
#include "example.h"                        /* reference menu structure */


/*************************************************************************/
/* define constants used in RTC example                                  */
/*************************************************************************/

#define CKSUM_START 0x10          /* offset of start of checksum area */
#define CKSUM_END   0x20          /* offset of end   of checksum area */

/* define register A bit values */

#define UIP         0x80                    /* update in progress bit */
#define DIVIDE_SEL  0x20            /* FIXED VALUE - Hardware related */
#define RATE_SEL    0x0d          /* Programmer defined interrupt rate */

/* define register B bit values */

#define SET_UPD     0x80           /* disable updating of date & time */
#define PIE         0x40                 /* Periodic Interrupt Enable */
#define AIE         0x20                   /* Alarm Interrupt Enable */
#define UIE         0x10             /* Update-Ended Interrupt Enable */
#define SQWE        0x08                        /* Square Wave Enable */
#define DAT_MOD     0x04           /* Data mode (BCD = 0, Binary = 1) */
#define CLK24       0x02            /* 12-hour clock = 0, 24 hour = 1 */
#define DSE         0x01                  /* Daylight Savings Enable */

/* define register C bit values */

#define IRQF        0x80                    /* Interrupt Request Flag */
#define PIF         0x40                  /* Periodic Interrupt Flag */
#define AIF         0x20                    /* Alarm Interrupt Flag */
#define UIF         0x10                      /* Update-Ended Flag */

/* define register D bit values */

#define VRT         0x80                      /* Valid Ram & Time bit */
```

Data Structures

The structure type RTC defines the I/O space that accesses the real-time clock. The offset within the real-time clock (00H-3FH) is written to an 8-bit latch that addresses the real-time clock. This latch is located at I/O address 0070H. Data is read or written through I/O address 0071H.

The structure type CMOS defines how each RAM location is used within the real- time clock. The real-time clock dedicates the first 14 locations as registers. The remaining 50 bytes of RAM are defined according to industry-standard usage.

The structure type DATIM provides a consistent format for moving date and time information.

```
/*************************************************************************/
/* declare structures used in RTC example                                */
/*************************************************************************/

typedef struct
{
    unsigned char addr_port;     /* write RTC/CMOS address to this port */
    unsigned char data_port;       /* read/write data through this port */
} RTC;

typedef struct
{
    unsigned char seconds;              /* seconds (0-59) current time */
    unsigned char alr_sec;                          /* seconds alarm */
    unsigned char minutes;              /* minutes (0-59) current time */
    unsigned char alr_min;                          /* minutes alarm */
    unsigned char hours;               /* hours (0-11/23) current time */
    unsigned char alr_hr;                            /* hours alarm */
    unsigned char dow;                        /* day-of-week (1-7)  */
    unsigned char dom;                /* day-of-month (1-28/29/30/31) */
    unsigned char month;               /* month  (1-12) current date */
    unsigned char year;                 /* year (0-99) current date */
    unsigned char rega;                                /* register A */
    unsigned char regb;                                /* register B */
    unsigned char regc;                                /* register C */
    unsigned char regd;                                /* register D */
    unsigned char diag;                          /* diagnostics byte */
    unsigned char reset;                         /* reason for reset */
    unsigned char ddtyp;                    /* diskette drive type */
    unsigned char reserv1;                        /* reserved byte */
    unsigned char hdtyp;                        /* hard disk type */
    unsigned char reserv2;                        /* reserved byte */
```

```
    unsigned char syscfg;                   /* system configuration byte */
    unsigned char bmeml;                    /* base memory size low byte */
    unsigned char bmemh;                   /* base memory size high byte */
    unsigned char lememl;              /* expansion memory size low byte */
    unsigned char lememh;             /* expansion memory size high byte */
    unsigned char reserv3[0x2e - 0x19];                    /* reserved */
    unsigned char cksumh;            /* high byte of checksum (always 0) */
    unsigned char cksuml;                          /* low byte of checksum */
    unsigned char hememl;              /* expansion memory size low byte */
    unsigned char hememh;             /* expansion memory size high byte */
    unsigned char century;       /* century byte of date (19 from 1986) */
    unsigned char info;                              /* information flag */
    unsigned char reserv4[0x40 - 0x34];                    /* reserved */
} CMOS;

typedef struct
{
    int seconds;                                        /* seconds (0-59) */
    int minutes;                                        /* minutes (0-59) */
    int hours;                                         /* hours (0-11/23) */
    int dow;                                        /* day-of-week (1-7)  */
    int dom;                              /* day-of-month (1-28/29/30/31) */
    int month;                                          /* month  (1-12) */
    int year;                    /* year (century * 100 + year register) */
} DATIM;
```

Reading the Registers and RAM

The function *rd_rtc* reads the indicated byte of real-time clock RAM. Before accessing the byte, the offset is compared to the range 00H-09H. This is the offset range of the data registers, which are invalid during update cycles. If the offset falls within this range, the read is synchronized with the update-in-progress bit.

```
/*************************************************************************/
/* rd_rtc() - read an RTC byte (if date or time, monitor UIP bit)        */
/*************************************************************************/

int rd_rtc(offset)                                        /* read RTC byte */

int offset;                                          /* byte offset to read */

{

RTC *prtc;                                    /* ptr to address & data ports */
CMOS *pcmos;                                    /* ptr to RTC/CMOS structure */
unsigned int intr_flg;                                      /* CPU IF state */
unsigned char retval;                                     /* value to return */

  prtc = (RTC *)0x70;                                  /* assign I/O address */
  pcmos = 0;                                    /* structure offset is zero */
  if(offset < (int)(&pcmos->rega))                    /* need to monitor UIP ? */
  {
    while(1)                                           /* break out when ready */
    {
      intr_flg = int_off();                          /* no interrupts allowed */
      outp(&prtc->addr_port, &pcmos->rega);                  /* set to reg A */
      if(inp(&prtc->data_port) & UIP)                        /* test UIP bit */
        int_on(intr_flg);                               /* allow interrupts */
      else break;
    }
  }
  else intr_flg = int_off();                         /* no interrupts allowed */
  outp(&prtc->addr_port, offset);                    /* set to desired offset */
  retval = inp(&prtc->data_port);                                /* read data */
  int_on(intr_flg);                                  /* allow interrupts */
  return(retval);                                    /* return data byte */
}
```

Writing the Registers and RAM

The function *wr_rtc* writes the indicated byte of real-time clock RAM. Before accessing the byte, the offset is compared to the range 00H-09H. This is the offset range of the data registers, which are invalid during update cycles. If the offset falls within this range, the write is synchronized with the update-in-progress bit.

```
/*************************************************************************/
/* wr_rtc() - write an RTC byte                                          */
/*************************************************************************/

void wr_rtc(offset, value)                          /* write RTC byte */

int offset;                                        /* offset to write */
unsigned char value;                             /* byte value to write */

{

RTC *prtc;                                 /* ptr to address & data ports */
CMOS *pcmos;                                 /* ptr to RTC/CMOS structure */
unsigned int intr_flg;                                    /* CPU IF state */

  prtc = (RTC *)0x70;                                 /* assign I/O address */
  pcmos = 0;                                    /* structure offset is zero */
  if(offset < (int)(&pcmos->rega))                 /* need to monitor UIP ? */
  {
    while(1)                                        /* break out when ready */
    {
      intr_flg = int_off();                       /* no interrupts allowed */
      outp(&prtc->addr_port, &pcmos->rega);                /* set to reg A */
      if(inp(&prtc->data_port) & UIP)                      /* test UIP bit */
        int_on(intr_flg);                          /* allow interrupts */
      else break;
    }
  }
  else intr_flg = int_off();                      /* no interrupts allowed */
  outp(&prtc->addr_port, offset);                 /* set to desired offset */
  outp(&prtc->data_port, value);                                /* write data */
  int_on(intr_flg);                              /* allow interrupts */
}
```

Calculating the Checksum

The function *rtc_cksum* calculates the checksum and returns the result to the caller. The checksum is the sum, modulo 256, of all bytes in the range CKSUM_START through CKSUM_END.

```
/*************************************************************************/
/* rtc_cksum() - calculate the CMOS checksum and return its value        */
/*************************************************************************/

unsigned char rtc_cksum()                  /* calculate the CMOS checksum */
{

int i;                                                  /* loop control */
unsigned char sum;                        /* accumulates the checksum */

  sum = 0;                                          /* sum starts out zero */
  for(i = CKSUM_START; i <= CKSUM_END; i++)          /* all checksum bytes */
    sum += rd_rtc(i);                             /* read data and add to sum */
  return(sum);                                  /* return calculated checksum */
}
```

Converting Binary-Coded Data

The ROM BIOS defines the data mode as binary-coded decimal (BCD) and therefore, so does this example. This requires converting between BCD and binary. The function *btb* converts a BCD value to its binary equivalent. The function *bcd* converts a binary value to its BCD equivalent.

```
/*************************************************************************/
/* btb  - convert bcd value to binary integer value                      */
/*************************************************************************/

btb(bcd_val)                                    /* bcd to binary integer */

unsigned char bcd_val;                                   /* bcd value */

{                                              /* assume valid bcd value */

  return(((bcd_val >> 4) * 10) + (bcd_val & 0x0f));
}


/*************************************************************************/
/* bcd  - convert binary value to bcd value                              */
/*************************************************************************/

unsigned char bcd(val)                                /* binary to bcd */

unsigned char val;                                     /* binary value */

{                                              /* assume valid bcd value */

unsigned char tmp;

  tmp = (val / 10) << 4;                        /* tens in upper nibble */
  tmp |= val % 10;                              /* ones in lower nibble */
  return(tmp);                                     /* return BCD value */
}
```

Reading the Date

The function *rd_date* reads all of the date-related registers and stores the results in the DATIM structure pointed to by the calling parameter. It can be called at any time without restriction.

The century byte is not a real-time clock register. The century byte is an industry-standard location that overcomes the 100-year calendar limitation. It must be updated manually or by software.

```
/*************************************************************************/
/* rd_date() - read date and write to DATIM structure                    */
/*************************************************************************/

void rd_date(pd)                                          /* read the date */

DATIM *pd;                                          /* where to store data */

{

CMOS *pcmos;                                  /* ptr to RTC/CMOS structure */

  pcmos = 0;                                   /* structure offset is zero */
  pd->dow      = btb(rd_rtc(&pcmos->dow));                 /* day-of-week */
  pd->dom      = btb(rd_rtc(&pcmos->dom));                /* day-of-month */
  pd->month    = btb(rd_rtc(&pcmos->month));                     /* month */
  pd->year     = btb(rd_rtc(&pcmos->century)) * 100;           /* century */
  pd->year    += btb(rd_rtc(&pcmos->year));                      /* year */
}
```

Reading the Time

The function *rd_time* reads all of the time-related registers and stores the results in the DATIM structure pointed to by the calling parameter. This function assumes the use of the 24-hour clock mode. It can be called at any time without restriction.

```
/*************************************************************************/
/* rd_time() - read time and write to DATIM structure                    */
/*************************************************************************/

void rd_time(pd)                                        /* read the time */

DATIM *pd;                                          /* where to store data */

{

CMOS *pcmos;                                  /* ptr to RTC/CMOS structure */

  pcmos = 0;                                   /* structure offset is zero */
  pd->seconds = btb(rd_rtc(&pcmos->seconds));                  /* seconds */
  pd->minutes = btb(rd_rtc(&pcmos->minutes));                  /* minutes */
  pd->hours   = btb(rd_rtc(&pcmos->hours));                     /* hours */
}
```

Displaying the Date

The function *shw_date* displays the current date starting at row 0 and column 0. It can be called at any time without restriction.

```
/*************************************************************************/
/* define some day and month names                                       */
/*************************************************************************/
char day_name[8][10] =
{
  "Invalid" ,                                    /* rtc is 1 based */
  "Sunday"  , "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday"
};

char month_name[13][10] =
{
  "Invalid",                                      /* rtc is 1 based */
  "January"  , "February", "March"    ,   "April",
  "May"      , "June"    , "July"     ,   "August",
  "September", "October" , "November",   "December"
};


/*************************************************************************/
/* shw_date - show date starting at row 0 column 0                       */
/*************************************************************************/

shw_date()
{

DATIM dt;                                    /* date and time structure */
char sdate[50];                               /* place to store output */

  rd_date(&dt);                                        /* read current date */
  sprintf(sdate, "%9s %9s %2d, %04d", &day_name[dt.dow][0],
          &month_name[dt.month][0], dt.dom, dt.year);
  disp_str(0, 0, sdate);                                  /* display it */
}
```

Displaying the Time

The function *shw_time* displays the current time starting at row 0 and column 72. It can be called at any time without restriction.

```
/*********************************************************************/
/* shw_time - show time on row 0 column (last_column - 7)            */
/*********************************************************************/

shw_time()
{

DATIM dt;                                    /* date and time structure */
char stime[50];                               /* place to store output */

  rd_time(&dt);                                      /* read current time */
  sprintf(stime, "%2d:%02d:%02d", dt.hours, dt.minutes, dt.seconds);
  disp_str(0, 72, stime);                                 /* display it */
}
```

Displaying the Diskette Drive Type

The function *shw_ddtyp* is a text-formatting subroutine that generates the diskette drive type according to the calling parameters. It is only called by the function shw_hdw.

```
/************************************************************************/
/* shw_ddtyp - show diskette drive type                                 */
/************************************************************************/

void shw_ddtyp(pc, ddtyp, drive)          /* show diskette drive type */

char *pc;                                    /* buffer to write to */
char ddtyp;                                 /* diskette drive type */
char drive;                                       /* drive letter */

{

int i;                                                /* temp for index */

  i = sprintf(pc, "Diskette drive %c is ", drive);  /* general opening */
  switch(ddtyp)
  {
    case 0:                                                 /* no drive */
      sprintf(&pc[i], "non-existent");
      break;

    case 1:                                                 /* 48 tpi dsdd */
      sprintf(&pc[i], "48-TPI double sided");
      break;

    case 2:                                               /* 96 tpi dsdd hc */
      sprintf(&pc[i], "an RX33 96-TPI double-sided, high-capacity");
      break;

    default:                                                /* unknown type */
      sprintf(&pc[i], "an unknown type");
      break;
  }
}
```

Displaying the Hard Disk Type

The function *shw_hdtyp* is a text-formatting subroutine that generates the hard disk drive type according to the calling parameters. It is only called by the function shw_hdw.

```
/*************************************************************************/
/* shw_hdtyp - show hard disk type                                       */
/*************************************************************************/

void shw_hdtyp(pc, hdtyp, drive)                /* show hard disk type */

char *pc;
char hdtyp;                                          /* hard disk type */
char drive;                                            /* drive letter */

{
int i;

  i = sprintf(pc, "Hard disk drive %c is ", drive);
  if(hdtyp) sprintf(&pc[i], "type %d", hdtyp);          /* drive type */
  else sprintf(&pc[i], "non-existent");                   /* no drive */
}
```

Handling the Clock Interrupts

The function *rtc_int_hand* is the real-time clock interrupt handler. It checks for all of the three possible interrupts, update-ended flag (UIF), alarm flag (AF), and periodic interrupt flag (PIF). After handling the interrupts, the interrupt handler notifies the interrupt controller.

The update-ended interrupt occurs once per second. At each interrupt, the interrupt handler increments the global flag, time_flag, to indicate that at least one second has elapsed.

The alarm interrupt is initialized to the first second of every day. At each interrupt, the interrupt handler increments the global flag, date_flag, to indicate that at least one day has elapsed.

The periodic interrupt is initialized to a rate of 125 ms. At each interrupt, the interrupt handler increments the global counter, metronome. The 8254 timer and speaker example in Chapter 6 uses this counter for output timing. Also, the periodic interrupt handler calls *unbeep*. If required, unbeep turns off the bell (beep sound). The example programs use the speaker to generate a bell (beep sound).

```
/*************************************************************************/
/* rtc_int_hand() - real-time clock interrupt handler                    */
/*************************************************************************/
int time_flag;                                    /* 1 second update flag */
int date_flag;                                      /* 1 day update flag */
unsigned int metronome;                          /* timer for sound output */
int motor_flag;                        /* timer for diskette drive motors */
int head_settle;                /* head settle timer for diskette drives */

void rtc_int_hand()                                      /* rtc int handler */
{

CMOS *pcmos;                                  /* ptr to RTC/CMOS structure */
unsigned char tmp;                              /* temp to read in reg C */

  pcmos = 0;                                   /* structure offset is zero */
  tmp = rd_rtc(&pcmos->regc);          /* read current interrupt requests */
  if(tmp & UIF) time_flag++;             /* time updates once per second */
  if(tmp & AIF) date_flag++; /* alarm set for once per day at 00:00:00 */
  if(tmp & PIF)                                  /* periodic interrupt ? */
  {
    metronome++;                     /* increment timing for speaker demo */
    unbeep();                         /* unbeep turns off speaker if bell */
    if(motor_flag)                     /* if timing diskette drive motors */
      if(--motor_flag == 0)                                 /* if timed out */
        motor_off();                   /* call routine to turn motors off */
    if(head_settle)                              /* if timing head settle */
      head_settle--;                                       /* reduce count */
  }
  eoi(1);                  /* end of interrupt for interrupt controller */
}
```

Interpreting the RAM Contents

The function *shw_hdw* interprets the industry-standard locations in the real-time clock RAM and displays the results. The ROM BIOS interprets this data in the same manner.

```
/*************************************************************************/
/* sh_hdw() - show hardware setup in CMOS                                */
/*************************************************************************/

sh_hdw()
{

unsigned char tmp;                          /* to hold CMOS byte read */
unsigned int ui;                              /* to hold memory size */
CMOS *pcmos;                              /* ptr to RTC/CMOS structure */
char *pc;
char line[512];

#define ROW 16
#define COL 17

  pcmos = 0;                               /* structure offset is zero */
  tmp = rd_rtc(&pcmos->syscfg);                 /* read system config */
  sprintf(line, "%d diskette drive(s) present", (tmp >> 6) + 1);
  disp_str(ROW, COL, line);
  switch((tmp & 0x30) >> 4)                       /* check video type */
  {
    case 0:                                       /* not a valid type */
      pc = "Invalid video type";
      break;

    case 1:
      pc = "40 column color graphics";
      break;

    case 2:
      pc = "80 column color graphics";
      break;

    case 3:
      pc = "Monochrome adapter with parallel port";
      break;
  }
  disp_str(ROW + 1, COL, pc);                   /* display video type */
  tmp = rd_rtc(&pcmos->ddtyp);            /* read diskette drive types */
```

```
    shw_ddtyp(line, tmp >> 4, 'A');                    /* get drive a type */
    disp_str(ROW + 2, COL, line);                   /* display drive a type */
    shw_ddtyp(line, tmp & 0x0f, 'B');                  /* get drive b type */
    disp_str(ROW + 3, COL, line);                   /* display drive b type */
    tmp = rd_rtc(&pcmos->hdtyp);                    /* read hard disk types */
    shw_hdtyp(line, tmp >> 4, 'C');                    /* get drive c type */
    disp_str(ROW + 4, COL, line);                   /* display drive c type */
    shw_hdtyp(line, tmp & 0x0f, 'D');                  /* get drive d type */
    disp_str(ROW + 5, COL, line);                   /* display drive d type */
    tmp = rd_rtc(&pcmos->bmemh);                   /* base memory high byte */
    ui = (unsigned int)tmp << 8;                       /* shift and assign */
    tmp = rd_rtc(&pcmos->bmeml);                    /* base memory low byte */
    ui |= (unsigned int)tmp;                                /* add low byte */
    sprintf(line, "Base memory = %dK bytes", ui);
    disp_str(ROW + 5, COL, line);                    /* display base memory */
    tmp = rd_rtc(&pcmos->lememh);              /* expanded memory high byte */
    ui = (unsigned int)tmp << 8;                       /* shift and assign */
    tmp = rd_rtc(&pcmos->lememl);               /* expanded memory low byte */
    ui |= (unsigned int)tmp;                                /* add low byte */
    sprintf(line, "Expansion memory = %dK bytes", ui);
    disp_str(ROW + 6, COL, line);                /* display expanded memory */
    tmp = rd_rtc(&pcmos->hememh);              /* expanded memory high byte */
    ui = (unsigned int)tmp << 8;                       /* shift and assign */
    tmp = rd_rtc(&pcmos->hememl);               /* expanded memory low byte */
    ui |= (unsigned int)tmp;                                /* add low byte */
    sprintf(line, "Expansion memory = %dK bytes", ui);
    disp_str(ROW + 7, COL, line);                /* display expanded memory */
}
```

Initializing the Real-Time Clock

To start up real-time clock interrupt processing, the *rtc_init* function:

- Disables real-time clock interrupts and update cycles
- Initializes the processor interrupt vector, the real-time clock control, and alarm registers; and unmasks the interrupt controller input
- Enables the real-time clock interrupts and update cycles

```
/*************************************************************************/
   /* rtc_init() - initialize alarms and vectors                          */
   /*************************************************************************/

   rtc_init()
   {
   CMOS *pcmos;                              /* ptr to RTC/CMOS structure */

     pcmos = 0;                               /* structure offset is zero */
     wr_rtc(&pcmos->regb, SET_UPD | CLK24);          /* prepare to init */
     imask(1, 0, 0);                        /* disable PIC interrupt */
     iv_init(0x70);                    /* initialize RTC interrupt vector */
     wr_rtc(&pcmos->rega, DIVIDE_SEL | RATE_SEL);        /* set pi rate */
     wr_rtc(&pcmos->alr_hr, 0x00);                /* write hours alarm */
     wr_rtc(&pcmos->alr_min, 0x00);              /* write minutes alarm */
     wr_rtc(&pcmos->alr_sec, 0x00);              /* write seconds alarm */
     wr_rtc(&pcmos->regb, AIE | UIE | PIE | CLK24   );    /* enable clock */
     imask(1, 0, 1);                          /* enable PIC interrupt */
   }
```

Restoring the Interrupt Vectors

To shut down real-time clock interrupt processing, the *rtc_rest* function:

- Disables the real-time clock interrupts
- Masks the interrupt controller input
- Restores the interrupt vector to its previous condition

NOTE
Update cycles remain enabled. If update cycles are disabled, the clock stops.

```
/*************************************************************************/
/* rtc_rest() - disable interrupts and restore vectors                   */
/*************************************************************************/

rtc_rest()
{
CMOS *pcmos;                                  /* ptr to RTC/CMOS structure */

  pcmos = 0;                                   /* structure offset is zero */
  wr_rtc(&pcmos->regb, CLK24);                 /* disable clock interrupts */
  imask(1, 0, 0);                                 /* disable PIC interrupt */
  iv_rest(0x70);                               /* restore interrupt vector */
}
```

Real-Time Clock Example

The function *rtc* displays the menu, accepts input, and executes the examples.

```
/*************************************************************************/
/* rtc() - execute RTC examples                                          */
/*************************************************************************/

rtc()
{
static MESSAGE mrtc[] =                                    /* rtc menu */
{
  {  3, 24, "Real-time Clock and CMOS Example" },
  {  5, 24, "F1.  Display CMOS hardware setup" },
  {  6, 24, "F2.  Display CMOS checksum" },
  {  7, 24, "F3.  Display calculated CMOS checksum" },
  {  8, 24, "F4.  Set CMOS checksum" },
  {  9, 24, "F5.  Set date" },
  { 10, 24, "F6.  Set time" },
  { 11, 24, "F7.  Set day-of-week" },
  { 12, 24, "F10. Return to Main menu" },
  {  0,  0,  0 },
};

unsigned char tmp;                          /* to hold CMOS byte read */
unsigned char sum;                       /* to hold calculated checksum */
char line[512];                             /* to hold input line */
int i;                                     /* to hold menu selection */
int r;                                                  /* temp value */
DATIM dt;                                /* place to store date & time */
CMOS *pcmos;                              /* ptr to RTC/CMOS structure */


#define ROW 16
#define COL 17

  pcmos = 0;                              /* structure offset is zero */
  line[0] = 0;                                    /* null terminated */
  while(1)                                       /* forever (see F10) */
  {
    disp_menu(mrtc);                          /* display the rtc menu */
    switch(line[0])                        /* determine menu selection */
    {
      case F1:                                    /* show CMOS hardware */
        sh_hdw();
        break;
```

```
case F2:                                        /* get current checksum */
  sprintf(line, "CMOS checksum = %02x",
  rd_rtc(&pcmos->cksuml));
  disp_str(ROW, COL, line);
  break;

case F3:                                        /* calculate checksum */
  sprintf(line, "Calculated checksum = %02x", rtc_cksum());
  disp_str(ROW, COL, line);
  break;

case F4:
  sum = rtc_cksum();                    /* write calculated checksum */
  wr_rtc(&pcmos->cksuml, sum);
  sprintf(line, "Checksum byte set to %02xH", sum);
  disp_str(ROW, COL, line);
  break;

case F5:                                                /* set new date */
  while(1)
  {
    disp_str(ROW    , COL,"Enter date as MM/DD/YYYY");
    disp_str(ROW + 1, COL, "Where MM represents the month (1 - 12)");
    disp_str(ROW + 2, COL, "Where DD represents the day (1 - 31)");
    disp_str(ROW + 3, COL, "Where YYYY represents the year (0000 -
    9999)");
    disp_str(ROW + 4, COL, "Date: ");
    get_keys(ROW + 4, COL + 6, line);
    r = sscanf(line, "%2d/%2d/%4d", &dt.month,
               &dt.dom, &dt.year);
    if(r != 3) continue;
    else break;
  }                                     /* note: no limit check */
  wr_rtc(&pcmos->month, bcd(dt.month));           /* write month */
  wr_rtc(&pcmos->dom, bcd(dt.dom));         /* write day-of-month */
  wr_rtc(&pcmos->year, bcd(dt.year % 100));         /* write year */
  wr_rtc(&pcmos->century, bcd(dt.year / 100));  /* write century */
  rd_rtc(&pcmos->regd);       /* make date valid, set the VRT bit */
  shw_date();
  disp_menu(mrtc);
  break;

case F6:                                                /* set new time */
  while(1)
  {
    disp_str(ROW    , COL, "Enter time as HH:MM:SS");
```

```
          disp_str(ROW + 1, COL, "Where HH represents the hour (0 - 23)");
          disp_str(ROW + 2, COL, "Where MM represents the minutes (0 - 59)");
          disp_str(ROW + 3, COL, "Where SS represents the seconds (0 - 59)");
          disp_str(ROW + 4, COL, "Time: ");
          get_keys(ROW + 4, COL + 6, line);
          r = sscanf(line, "%2d:%2d:%2d", &dt.hours, &dt.minutes,
&dt.seconds);
          if(r != 3) continue;
          else break;
        }                                        /* note: no limit check */
        wr_rtc(&pcmos->hours, bcd(dt.hours));          /* write hours */
        wr_rtc(&pcmos->minutes, bcd(dt.minutes));     /* write minutes */
        wr_rtc(&pcmos->seconds, bcd(dt.seconds));     /* write seconds */
        rd_rtc(&pcmos->regd);      /* make time valid, set the VRT bit */
        shw_time();
        disp_menu(mrtc);
        break;

      case F7:
        while(1)
        {
          disp_str(ROW, COL, "Enter day-of-week (1 - 7): ");
          get_keys(ROW, COL + 27, line);
          r = sscanf(line, "%d", &dt.dow);
          if(r != 1) continue;
          else break;
        }                                        /* note: no limit check */
        wr_rtc(&pcmos->dow, bcd(dt.dow));         /* write day-of-week */
        rd_rtc(&pcmos->regd);       /* make DOW valid, set the VRT bit */
        shw_date();
        disp_menu(mrtc);
        break;

      case F10:                          /* return to caller (main menu) */
        return;
    }
    line[0] = get_fkey();      /* get a function key for menu selection */
  }
}
```

Chapter 6
Three-Channel Counter and Speaker

Overview

The VAXmate processor board has an 8254 programmable interval timer that provides three independent 16-bit counters for counting or timing. All three counters have a 1.1931816 MHz clock input. The counters are programmable and are used by the ROM BIOS as follows:

- Counter 0 is a general purpose timer to provide:
  - A time-of-day clock
  - A diskette drive motor timer
  - A screen blanking timer

  Its output goes to IRQ0 of the interrupt logic.

  CAUTION: Reprogramming this counter can destroy the timing.

- Counter 1 provides the dynamic RAM refresh timing. The ROM BIOS programs it for a 15 μs cycle time. Its output is connected to the refresh counter.

  CAUTION: Reprogramming this counter can destroy the refresh cycle.

- Counter 2 provides a frequency-modulated square wave output for the speaker interface.

Additional Source of Information

The following Intel Corporation document provides additional information on the 8254 three-channel counter/timer.

- *Microsystems Components Handbook* (Publication Number 230843)

Block Diagram

Figure 6-1 shows the block diagram of an 8254. The data bus buffer interfaces the I/O data bus, and the read/write logic interfaces the address bus on the CPU module.

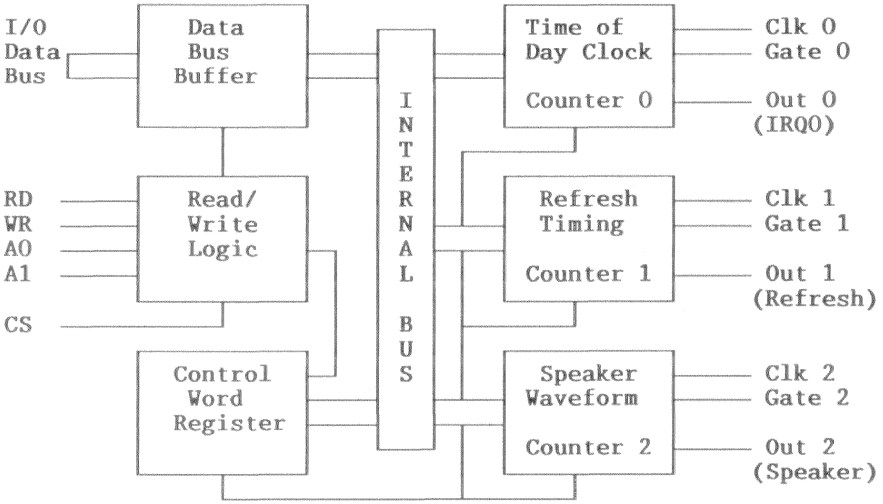


Figure 6-1 Three-Channel Counter/Timer Block Diagram

Counter Description

The three 16-bit synchronous down counters are identical in operation, but fully independent. Each counter has two 8-bit input latches (count registers), two 8-bit output latches, and a counting element. The counter control logic enables only one latch at a time. Therefore, writing a 16-bit count requires two 8-bit writes to the same register and reading a 16-bit count requires two 8-bit reads from the same register.

The control word register provides for 8-bit and 16-bit counts. The 8-bit count can be written to either the least significant byte (LSB) or the most significant byte (MSB). Loading one 8-bit count register of a 16-bit pair clears the other count register. That is, writing an 8-bit count to the LSB clears the MSB and writing an 8-bit count to the MSB clears the LSB.

The signals CLK, GATE, and OUT are all connected to control logic on the CPU module. A 14.31818 MHz signal divided by 12 provides a 1.1931816 MHz clock to all three counters.

A high level (1) at the GATE input enables counting and a low level (0) at the GATE input disables counting.

Table 6-1 shows the CLK input frequency, the GATE source, and the destination of the OUT signal.

Table 6-1 Counter Signals

Counter	CLK Frequency	GATE Source	OUT Destination
0	1.1931816 MHz	Tied high	IRQ0
1	1.1931816 MHz	Tied high	Refresh timer
2	1.1931816 MHz	System CSR (bit 0)	Speaker driver

Mode Definitions

The three-channel counter/timer has six modes of operation:

Mode 0 Interrupt on Terminal Count
Mode 1 Hardware Retriggerable One-Shot (not used)
Mode 2 Rate Generator
Mode 3 Square Wave Mode
Mode 4 Software Triggered Strobe
Mode 5 Hardware Triggered Strobe (retriggerable)

Table 6-2 lists the default mode and function of each counter in the VAXmate workstation (as established by the ROM BIOS).

Table 6-2 Modes Used by the Three Counters

Counter	Function	Mode	Description	Output
0	Time-of-day clock	5	Hardware triggered strobe	IRQ0
1	Refresh timing	2	Rate generator	Refresh counter
2	Speaker waveform	3	Square wave	Speaker driver

Mode 0 (Interrupt on Terminal Count)

Mode 0 is used for one-shot event counting.

Initializing Mode 0

Programming the control word for mode 0 causes OUT to go low. The GATE input has no effect on the OUT signal.

If a new count is written during counting, the new count is loaded on the next CLK pulse and counting continues from the new count.

Mode 0 Cycle

Writing a new count starts the cycle. Where n equals the count, the mode 0 cycle is $n + 1$ CLK pulses long. During the start of the cycle, the OUT signal is low for n CLK pulses (while the counter decrements from n to 0.) On the next CLK pulse, the OUT signal makes a transition from low-to-high. The OUT signal remains high until the control word is written or until a new count is written.

Mode 1 (Hardware Retriggerable One-Shot)

Mode 1 is used for one-shot event counting. Because the GATE input is the trigger, mode 1 is viable only on counter 2 (the GATE input of counters 0 and 1 are tied high.) Mode 1 could be used for sound generation, but it is not normally used on the VAXmate workstation.

Initializing Mode 1

Programming the control word for mode 1 and writing a new count causes OUT to go high and arms the trigger. The GATE input has no effect on the OUT signal.

Writing a new count during counting has no effect on the current count. However, if the GATE input is triggered, the cycle restarts with the new count.

Mode 1 Cycle

With the trigger armed, a low-to-high transition at the GATE input triggers the cycle. On the next CLK pulse, the count is loaded and the OUT signal makes a high-to-low transition. Where n equals the count, the OUT signal remains low for n CLK pulses. That is, when the count decrements to 0, the OUT signal goes high.

After the trigger is armed for the first time, the trigger remains armed until the control word is reprogrammed. Thus, after a count has decremented to 0, triggering the GATE input restarts the cycle. The count is reloaded

automatically.

Triggering the GATE input before a count decrements to 0 restarts the cycle on the next CLK pulse. The count is reloaded automatically. Because the count did not expire, the OUT signal remains low.

Mode 2 (Rate Generator)

Where *n* equals the initial count, mode 2 functions like a divide by *n* counter. It generates pulses at a rate equal to the *input frequency* divided by the initial count (*n*). Mode 2 is periodic, repeating the cycle every *n* CLK pulses.

The ROM BIOS uses counter 1 in mode 2 to provide the refresh timing signal.

Initializing Mode 2

Programming the control word for mode 2 causes OUT to go high. Providing that the GATE input is high, the cycle starts 1 CLK pulse after the initial count is written.

If the GATE input goes low, the OUT signal goes high immediately. On the CLK pulse following a low-to-high transition at the GATE input, the counter reloads the initial count. Thus, the GATE input can synchronize the count to an external event.

Writing a new count during counting has no effect on the count for the current cycle. When the cycle repeats, the count is reloaded with the new count. However, if the GATE input is triggered, the new count is loaded on the next CLK pulse and cycle restarts.

Mode 2 Cycle

Where *n* is the initial count, the mode 2 cycle is *n* CLK pulses long. During the start of the cycle, the OUT signal is high. The OUT signal remains high until the count decrements to 1. When the count decrements to 1, the OUT signal makes a high-to-low transition and the counter reloads the initial count. The OUT signal is low only for that CLK pulse. On the next CLK pulse, the OUT signal goes high and the cycle repeats.

NOTE
In mode 2, a count of 1 is invalid.

Mode 3 (Square Wave Mode)

Mode 3 generates a square wave at the OUT signal. Where *n* is the count, the OUT signal has a frequency equal to CLK / *n*. When *n* is an even number, the OUT signal is high for *n* / 2 CLK pulses and then low for *n* / 2 CLK pulses. When *n* is an odd number, the OUT signal is high for (*n* + 1) / 2 CLK pulses and then low for (*n* - 1) / 2 CLK pulses.

Initializing Mode 3

Programming the control word for mode 3 causes OUT to go high. Providing that the GATE input is high, the cycle starts 1 clock pulse after the initial count is written.

If the GATE input goes low, the OUT signal goes high immediately. On the CLK pulse following a low-to-high transition at the GATE input, the counter reloads the initial count. Thus, the GATE input can synchronize the count to an external event.

Writing a new count during counting has no effect on the count for the current cycle. When the cycle repeats, the count is reloaded with the new count. However, if the GATE input is triggered, the new count is loaded on the next CLK pulse and cycle restarts.

Mode 3 Cycle

In the first CLK pulse, the initial count is loaded. If the count is odd, then count - 1 is loaded. The cycle starts with the next CLK pulse.

With each succeeding CLK pulse, the count is decremented by two. When the count decrements to 0, the initial count is tested for an odd or even value. If the initial count was even, the OUT signal makes an immediate transition from high-to-low. If the initial count was odd, the counter waits one more CLK pulse and then makes a high-to-low transition. The initial count is reloaded and decremented by two, which starts the second half of the cycle. With each succeeding CLK pulse, the count is decremented by two. When the count decrements to 0, the OUT signal makes an immediate transition from low-to-high; the initial count is reloaded and decremented by two, which starts a new cycle.

Mode 4 (Software Triggered Strobe)

In mode 4, the count cycle is triggered by writing a new count. Where n equals the initial count, the OUT signal is high for $n + 1$ CLK pulses, low for 1 CLK pulse, and then high until a new count is written.

Initializing Mode 4

Programming the control word for mode 4 causes OUT to go high. The GATE input has no effect on the OUT signal.

If a new count is written during counting, the new count is loaded on the next CLK pulse and counting continues from the new count.

For a 2-byte count, writing the first byte has no effect on counting. Writing the second byte allows the count to be loaded on the next CLK pulse.

Writing a new count while the original count is counting allows the new count to be loaded on the next CLK pulse.

Mode 4 Cycle

The mode 4 cycle is triggered by writing a new count. The new count is loaded on the next CLK pulse, but not decremented. With each successive CLK pulse, the count is decremented. When the count decrements to 0, the OUT signal goes low. It remains low for 1 CLK pulse. When the count decrements to FFFFH, the OUT signal goes high. Until a new count is written, the OUT signal remains high, which restarts the cycle.

Mode 5 (Hardware Triggered Strobe)

Because the GATE input is the trigger, mode 5 is viable only on counter 2 (the GATE input of counters 0 and 1 are tied high.) Where n equals the initial count, the OUT signal is high for $n + 1$ CLK pulses, low for 1 CLK pulse, and then high until the GATE input triggers another cycle.

Initializing Mode 5

Programming the control word for mode 1 and writing a new count causes OUT to go high and arms the trigger. The GATE input has no effect on the OUT signal.

Writing a new count during counting has no effect on the current count. However, if the GATE input is triggered, the cycle restarts with the new count.

Mode 5 Cycle

With the trigger armed, a low-to-high transition at the GATE input triggers the cycle. The new count is loaded on the next CLK pulse, but not decremented. With each successive CLK pulse, the count is decremented. When the count decrements to 0, the OUT signal goes low. It remains low for 1 CLK pulse. When the count decrements to FFFFH, the OUT signal goes high.

After the trigger is armed for the first time, the trigger remains armed until the control word is reprogrammed. Thus, after a count has decremented to 0, triggering the GATE input restarts the cycle.

Triggering the GATE input before a count decrements to 0 restarts the cycle on the next CLK pulse. The count is reloaded automatically. Because the count did not expire, the OUT signal remains high.

Registers

This section discusses the 8254 registers. Because bit 0 controls the GATE input of counter 2 and bit 1 controls the output to the speaker, the system register is also discussed here. Table 6-3 shows the addresses of the 8254 registers and the system register.

Table 6-3 8254 and System Register Addresses

Register	R/W	Address
8254 - Counter 0	R/W	0040H
8254 - Counter 1	R/W	0041H
8254 - Counter 2	R/W	0042H
8254 - Command Word	W	0043H
System	R/W	0061H

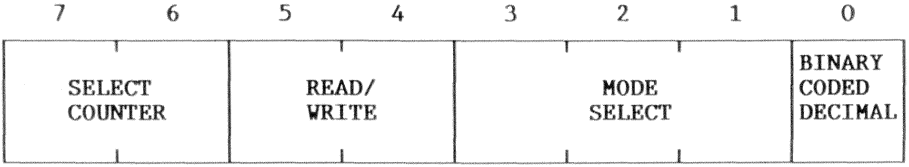
System Register (0061H)

7	6	5	4	3	2	1	0
RAM PARITY CHECK	I/O CHECK	COUNTER 2 OUT SIGNAL	REFRESH REQUEST	ENABLE I/O CHECK	ENABLE RAM PARITY	SPEAKER DATA	COUNTER 2 GATE INPUT

Bit	R/W	Description
7	R	RAM PARITY CHECK 0 = Processor board RAM parity good 1 = Processor board RAM parity error
	W	Always 0
6	R	I/O CHECK 0 = No bus I/O error or option RAM parity error 1 = Bus I/O or option RAM parity error exists
	W	Always 0
5	R	COUNTER 2 OUT SIGNAL 0 = Counter 2 OUT signal is low 1 = Counter 2 OUT signal is high
	W	Always 0
4	R	REFRESH REQUEST 0 = Refresh request not active 1 = Refresh request active The diagnostic software uses this bit to check the operation of the DRAM refresh circuitry.
	W	Always 0
3	R/W	ENABLE I/O CHECK 0 = Enables checking of the bus I/O check line and option RAM parity (enabled by ROM BIOS) 1 = Disable bus I/O error checking
2	R/W	ENABLE RAM PARITY CHECK 0 = Enable processor board RAM parity checking (enabled by ROM BIOS) 1 = Disable processor board RAM parity checking

Bit	R/W	Description (System Register - cont.)
1	R/W	SPEAKER DATA 0 = No sound output from speaker 1 = Sound output from speaker (Counter 2 OUT signal must be high or generating a frequency) The output of this bit is *AND*ed with the Counter 2 OUT SIGNAL. Assuming that the counter 2 OUT signal is high, toggling this bit generates a pulse train to the speaker driver. Otherwise, to enable sound output to the speaker, this bit must equal 1.
0	R/W	COUNTER 2 GATE INPUT 0 = Counter 2 GATE input is low 1 = Counter 2 GATE input is high

Control Word Register (0043H)



Bit	R/W	Description
7-6	W	SELECT COUNTER 00 = Select counter 0 01 = Select counter 1 10 = Select counter 2 11 = Read-back command
5-4	W	READ/WRITE 00 = Counter-latch command 01 = Read/Write LSB 10 = Read/Write MSB 11 = Read/Write LSB first, then MSB *
3-1	W	MODE SELECT 000 = Mode 0 001 = Mode 1 X10 = Mode 2 X11 = Mode 3 100 = Mode 4 101 = Mode 5
0	W	BINARY CODED DECIMAL 0 = Binary counter 16 bits 1 = Binary-coded-decimal (BCD) counter (4 decades)

* The counter does not start counting until the second byte of the 2-byte pair is written to the counter latch.

Counter-Latch Command (Control Word Register)

7	6	5	4	3	2	1	0
SELECT COUNTER		0	0	0	0	0	0

Bit	R/W	Description
7-6	W	SELECT COUNTER 00 = Select Counter 0 01 = Select Counter 1 10 = Select Counter 2 11 = Undefined
5-0	W	Always 0 for counter-latch command

Counter-latch commands do not affect the programmed mode of the counter. The counter-latch command latches the contents of the counters without affecting the count in progress. When the 8254 receives a counter-latch command, it latches the selected counter into the counters output latch. The latched count is held until read by the CPU (or until the counter is reprogrammed). After the latched count is read, the output latch follows the count in the counter.

When a counter-latch command is issued for more than one counter, each counter output latch holds the count until read. When any given counter is latched two or more times without an intervening read, only the first latch command is effective. When read, the count is the count latched by the first counter-latch command.

The latched count must be read according to the programmed format (LSB, MSB, or LSB and MSB).

Read-Back Command (Control Word Register)

7	6	5	4	3	2	1	0
1	1	LATCH COUNT	LATCH STATUS	COUNTER 2 SELECT	COUNTER 1 SELECT	COUNTER 0 SELECT	0

Bit	R/W	Description
7-6	W	Always 11
5-4	W	LATCH COUNT and LATCH STATUS 00 = Latch status and count of selected counter(s) 01 = Latch count of selected counter(s) 10 = Latch status of selected counter(s) 11 = Undefined
3	W	COUNTER 2 SELECT 0 = Counter 2 not selected 1 = Counter 2 selected
2	W	COUNTER 1 SELECT 0 = Counter 1 not selected 1 = Counter 1 selected
1	W	COUNTER 0 SELECT 0 = Counter 0 not selected 1 = Counter 0 selected
0	W	Always 0

The read-back command is written to the control word register. For the selected counters, the read-back command latches a status byte and/or the current count.

The status byte format is described under *Status Response*. The status byte is read from the indicated counter register as a single 8-bit byte. When the read-back command latches both status and count, the status byte is read first and then the count. Thereafter, any read returns an unlatched count.

The latched count follows the format described under *Counter-Latch Command*.

If multiple read-back commands are issued without intervening reads, all but the first are ignored. The status read is the status at the time of the first read-back command.

Status Response (Read-back Command)

7	6	5 4	3 2	1 0
OUT PIN	NULL COUNT	READ/ WRITE	SELECTED MODE	BINARY CODED DECIMAL

Bit	R/W	Description
7	R	OUT PIN 1 = OUT pin is high (1) 0 = OUT pin is low (0)
6	R	NULL COUNT 0 = New count is loaded and is available for reading. 1 = Null count A write to the control word register has occurred, which sets the null count bit of specified counter. If the counter is programmed for 2-byte counts, when the second byte is written, the null count goes to 1.
5-4	R	READ/WRITE 00 = Counter-latch command 01 = Read/Write LSB 10 = Read/Write MSB 11 = Read/Write LSB first, then MSB.
3-1	R	SELECTED MODE 000 = Mode 0 001 = Mode 1 X10 = Mode 2 X11 = Mode 3 100 = Mode 4 101 = Mode 5
0	R	BINARY CODED DECIMAL 0 = Binary counter 16 bits 1 = Binary-coded-decimal (BCD) counter (4 decades)

This page is intentionally blank.

Programming Example

The three channel counter/timer and speaker programming example demonstrates:

- Writing the counter/timer registers
- Enabling and disabling the output to the speaker
- Setting the output frequency to the speaker

The example provides routines as described in the following list:

wr_cnt16	Writes a 16-bit value to the indicated counter
beep	Enables the bell (beep) at the speaker
unbeep	Disables the bell (beep) at the speaker
tim_spk	Initializes the counter, displays the menu, and executes the example program

> CAUTION
> Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

Constant Values

The included file *kyb.h* defines constant values for function keys. For information about keyboard programming, see Chapter 8. For a listing of the file *kyb.h*, see Appendix A.

The included file *example.h* defines the structure type MESSAGE that is used to display the menu. For a listing of the file *example.h*, see Appendix A.

The constant value *systat* defines the offset of the system status register in I/O space.

The constant values *cwrdreg* through *count2* define the offset of the 8254 counter/timer registers in I/O space.

The constant values *selcnt0* through *rbcnt2* define the bit values of various 8254 counter/timer commands.

The constant value *inpfreq* defines the input frequency to all three counter/timers.

```
#include "kyb.h"                    /* reference function key constants */
#include "example.h"                        /* reference menu structure */

/*************************************************************************/
/* define constants used to program 8254 timer                           */
/*************************************************************************/

#define SYSTAT  0x61            /* system status register in I/O space */

#define CWRDREG 0x43             /* control word register in I/O space */
#define COUNT0  0x40               /* counter 0 register in I/O space */
#define COUNT1  0x41               /* counter 1 register in I/O space */
#define COUNT2  0x42               /* counter 2 register in I/O space */

#define SELCNT0 0x00                                /* select counter 0 */
#define SELCNT1 0x40                                /* select counter 1 */
#define SELCNT2 0x80                                /* select counter 2 */
#define SELRDBK 0xC0                                /* select read back */

#define LATCOM  0x00                          /* select latch command */
#define RWLSB   0x10                               /* read/write LSB */
#define RWMSB   0x20                               /* read/write MSB */
#define RWLSMS  0x30                       /* read/write LSB then MSB */

#define TMODE0  0x00                          /* select timer mode 0 */
#define TMODE1  0x02                          /* select timer mode 1 */
#define TMODE2  0x04                          /* select timer mode 2 */
#define TMODE3  0x06                          /* select timer mode 3 */
#define TMODE4  0x08                          /* select timer mode 4 */
#define TMODE5  0x09                          /* select timer mode 5 */

#define BINDAT  0x00                            /* binary count data */
#define BCDDAT  0x01                /* binary coded decimal count data */

#define LATCNT  0x20                     /* read back cmd latch count */
#define LATSTA  0x10                    /* read back cmd latch status */
#define RBCNT0  0x02                          /* read back counter 0 */
#define RBCNT1  0x04                          /* read back counter 1 */
#define RBCNT2  0x08                          /* read back counter 2 */

#define INPFREQ 1193181L          /* 14.31818 Mhz / 12 = 1.1931816 Mhz */
```

Writing a Counter

The function *wr_cnt16* writes a 16-bit value to the indicated counter. A 16-bit value is written 8-bits at a time (low byte first) to the same port.

Making a Bell Sound

The function *beep* enables the speaker output at 1000 Hz. It provides the bell (beep sound) for the ASCII character BEL (07H). This function can be called at any time. The speaker output is automatically disabled by the function *unbeep*.

The function *unbeep* monitors the variable beep_flag. If required, it disables the speaker. This function is called from within the real time clock interrupt handler. It tracks the number of 125 ms periods that the speaker has been on for a bell (beep sound). After 500 ms total, the speaker output is disabled. If the real time clock interrupts are not enabled, the speaker output will not be disabled automatically.

```
/*************************************************************************/
/* wr_cnt16() - write 16-bit value to counter                            */
/*************************************************************************/

wr_cnt16(counter, value)

unsigned char counter;                          /* which counter to set */
unsigned int value;                                    /* 16-bit value */

{
unsigned int intr_flag;                     /* to hold current IF state */

  intr_flag = int_off();                            /* disable interrupts */
  outp(counter | COUNTO, value & 0xff);       /* write counter low byte */
  outp(counter | COUNTO, value >> 8);        /* write counter high byte */
  int_on(intr_flag);                             /* enable interrupts */
}

/*************************************************************************/
/* beep() - start up beep sound at speaker                               */
/*************************************************************************/
int beep_flag;                                    /* true while beeping */

beep()
{
  wr_cnt16(2, (int)(INPFREQ / 1000L));         /* set desired frequency */
  outp(SYSTAT, 0x03);                               /* turn speaker on */
  beep_flag = 1;                               /* set flag, speaker is on */
}

/*************************************************************************/
/* unbeep() - time to stop beep sound at speaker ?                       */
/*************************************************************************/

unbeep()
{
  if(beep_flag)                          /* are we making a beep sound */
    if(++beep_flag > 3)                  /* has it been on long enough */
    {
      outp(SYSTAT, 0x00);                                /* turn it off */
      beep_flag = 0;                                      /* reset flag */
    }
}
```

Counter and Speaker Example

The function *tim_spk* initializes the counter, displays the menu, and executes the example.

```
/*************************************************************************/
/* tim_spk() - execute timer and speaker examples                        */
/*************************************************************************/

tim_spk()
{
static MESSAGE mtim_spk[] =          /* menu for timer/speaker example */
{
  {  3, 24, "8254 Timer and Speaker Example" },
  {  5, 24, "F1.  Set frequency to speaker" },
  {  6, 24, "F2.  Speaker on" },
  {  7, 24, "F3.  Speaker off" },
  {  8, 24, "F4.  DO-RE-MI" },
  {  9, 24, "F10. Return to Main menu" },
  {  0,  0,  0 },
};

static int tone[8] =            /* frequencies for notes to do-re-mi */
{   2281, 2032, 1810, 1709, 1524, 1366, 1209, 1140 };

char line[512];                                  /* to hold input line */
unsigned int freq;                             /* to remember frequency */
unsigned int tval;                              /* general temporary */
unsigned int i;                                 /* iteration control */

extern unsigned int metronome;             /* defined in clock example */
                                        /* maintains beat of do-re-mi */
#define ROW 16
#define COL 17

  line[0] = 0;
  freq = 1000;                                      /* default frequency */
                                                /* initialize counter mode */
  outp(CWRDREG, SELCNT2 | RWLSMS | TMODE3 | BINDAT);
  while(1)
  {
    disp_menu(mtim_spk);
    switch(line[0])
    {
      case F1:                                  /* set output frequency */
        disp_str(ROW, COL, "Enter new frequency (19Hz - 20000Hz):");
```

```
        get_keys(ROW, COL + 37, line);
        sscanf(line, "%d", &freq);
        if(freq < 18) freq = 19;
        else if(freq > 20000) freq = 20000;
        tval = (int)(INPFREQ / (long)freq);
        wr_cnt16(2, tval);
        disp_menu(mtim_spk);
        break;

      case F2:                                    /* turn speaker on */
        outp(SYSTAT, 0x03);
        break;

      case F3:                                   /* turn speaker off */
        outp(SYSTAT, 0x00);
        break;

      case F4:                                     /* play do-re-mi */
        i = 0;                                  /* iteration count = 0 */
        metronome = 0xffff;                /* prepare counter to overflow */
        while(metronome) ;                    /* wait until it overflows */
        wr_cnt16(2, tone[i++]);                      /* start first note */
        outp(SYSTAT, 0x03);                            /* enable speaker */
        while(i < 9)                                     /* do all notes */
        {
          if(metronome > 3)                      /* hold note for 500 ms */
          {
            wr_cnt16(2, tone[i++]);                          /* next note */
            metronome = 0;                                /* reset counter */
          }
          chk_dt();                          /* redisplay time for menu ? */
        }
        outp(SYSTAT, 0x00);                              /* turn speaker off */
        tval = (int)(INPFREQ / (long)freq);               /* reset frequency */
        wr_cnt16(2, tval);
        break;

      case F10:                                          /* return to caller */
        return;
    }
    line[0] = get_fkey();                                /* get function key */
  }
}
```

Chapter 7
Video Controller

Introduction

The VAXmate video controller is on the I/O board and drives a monochrome monitor. The video controller can process 16 colors or shades of gray. In this chapter, the term color also means shades of gray or intensity levels.

Industry-Standard Text and Graphics Features

The VAXmate video controller has the following industry-standard text and graphics features:

- 80 x 25 and 40 x 25 text display
- 8 x 8 graphics character cell
- character attributes:
  - 16 foreground colors
  - 16 background colors or 8 background colors plus blink
- bit map graphics with industry-standard color palettes
  - 320 x 200 4 colors
  - 640 x 200 2 colors

Enhancements to Industry-Standard Features

The video controller has the following enhancements to industry-standard features:

- The screen resolution is 640 horizontal pixels by 400 scan lines. Industry-standard graphics (200 scan lines) is accomplished by displaying each scan line twice.
- The character pattern is 8 horizontal pixels by 16 scan lines, resulting in higher quality characters in text modes.
- The 256-character font RAM provides flexibility in terminal emulation and multilingual applications.
- The dual-port video memory eliminates annoying screen flicker (disabling the screen before accessing video memory is unnecessary).
- The 16-bit data path to video memory, coupled with the dual-port access results in faster screen updates.

Industry-Standard Features Not Available

The video controller does not support these features:

- 160 x 100 16-color graphics mode
- 15.75 KHz monitor support
- Border color support
- Light pen support

Extra Features

The video controller has the following additional graphic features:

- 640 x 400 2-color graphics
- 640 x 400 4-color graphics
- 640 x 200 4-color graphics
- 800 x 252 4-color graphics
- 320 x 200 16-color graphics
- 256-character soft font

Block Diagram

The video controller consists of a display processor and video memory that reside on the I/O board. As shown in Figure 7-1, the display processor includes a translation ROM, a 6845 CRT controller, text video logic, graphics video logic, a video look-up table, and status and control registers.

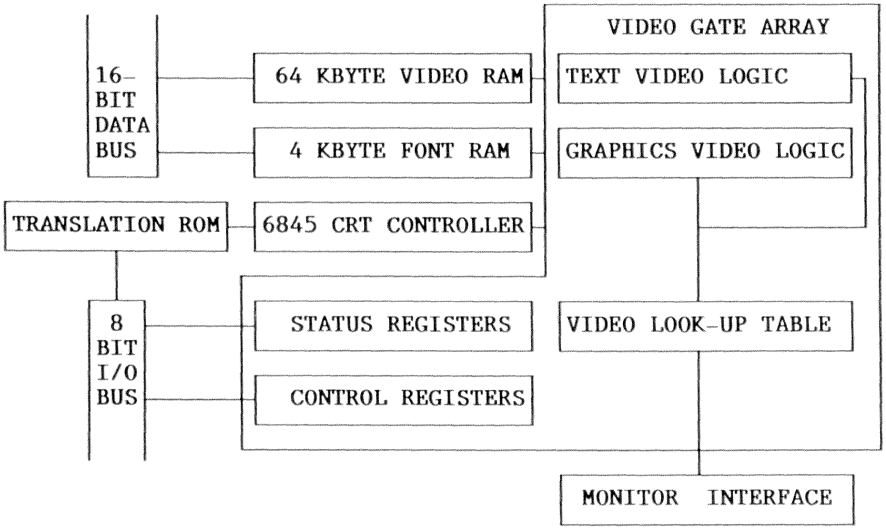


Figure 7-1 Block Diagram of the VAXmate Video Controller

The translation ROM translates industry-standard color graphic adapter data to data that is correct for the DIGITAL video controller.

The 6845 CRT Controller (CRTC) internal registers control horizontal and vertical positioning, synchronization, video and cursor starting addresses, and width of video display.

Two status registers monitor vertical synchronization, video blanking time, and various modes in the control registers.

The two control registers enable the various text and graphics modes, enable and disable the display, select the font RAM, select the video look-up table (VLT), and provide screen saver support.

64K bytes of dual-ported memory, which maps into the address space of the VAXmate CPU.

The display processor converts memory data into various raster formats. The display processor generates IRGB outputs that drive the monochrome monitor. The VAXmate monitor displays the color information as different levels of intensity (shades of gray).

Additional Sources of Information

The following documents provide additional information on the video controller:

Device	Company	Document
6845-1	Motorola	*8-Bit Microprocessor & Peripheral Data*
HD46505S	Hitachi	*8/16-Bit Multi-Chip Microcomputer Data Book*

Video Modes

The video controller has several modes, some of which have a mode number assigned indicating that the ROM BIOS supports these modes. For modes not supported by the ROM BIOS, the hardware must be programmed directly. Table 7-1 shows the available video modes.

For industry-standard color graphic adapters, the difference between a color and a monochrome mode is the presence (color) or absence (monochrome) of the color burst signal in the composite video output. Because the VAXmate video controller does not provide a composite video output, there is no difference between the color and the monochrome modes.

On powerup or system reset, the video system is initialized to mode 03H.

Table 7-1 Available Video Modes

Mode	Size	Description
00H	40 x 25	text mode monochrome (industry-standard)
01H	40 x 25	text mode color (industry-standard)
02h	80 x 25	text mode monochrome (industry-standard)
03h	80 x 25	Text mode color (industry-standard)
04H	320 x 200	4-color graphics mode (industry-standard)
05h	320 x 200	monochrome graphics (industry-standard)
06h	640 x 200	monochrome graphics mode (industry-standard)
-	320 x 200	16-color graphics mode (digital extended) *
d0h	640 x 400	2-color graphics mode (digital extended)
d1h	640 x 400	4-color graphics mode (digital extended)
d2h	800 x 252	4-color graphics mode (digital extended) **
-	640 x 200	4-color graphics mode (DIGITAL extended) *

* No ROM BIOS support
** Limited ROM BIOS support

Text Modes

The video controller has a 16 Kbyte text buffer in the address range B8000H-BBFFFH. Video modes 00H, 01H, 02H, and 03H use the text buffer. For modes 00H and 01H, the text buffer provides 8 display pages of 2048 bytes each. For modes 02H and 03H, the text buffer provides 4 display pages of 4096 bytes each.

Character Buffer Format

A displayed character is represented by two consecutive bytes. The first byte, of the 2-byte pair, is the character code. The character code is stored at an even address. The second byte, of the 2-byte pair, is the attribute byte. The attribute byte is stored at the odd address following the character code. Figure 7-2 shows the character code and attribute byte addressing. Table 7-2 defines the meaning of each bit within the attribute byte.

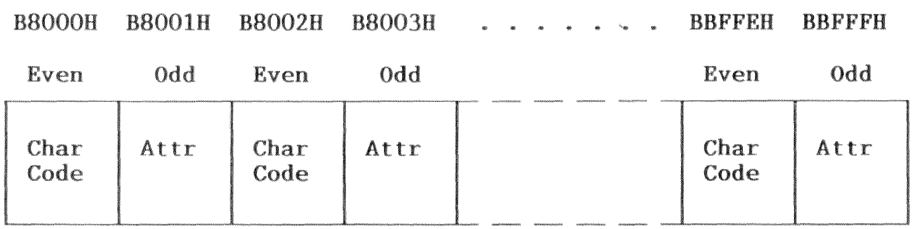


Figure 7-2 Character Buffer Format

Table 7-2 Attribute Byte Bit Definitions

Bit	Symbol	Definition
7	Ib	Background intensity / Blink *
6	Rb	Red contribution to background color
5	Gb	Green contribution to background color
4	Bb	Blue contribution to background color
3	If	Foreground intensity
2	Rf	Red contribution to foreground color
1	Gf	Green contribution to foreground color
0	Bf	Blue contribution to foreground color

* The selection of background intensity or blink is determined by bit 7 of control register A. Control register A is described later in this chapter. When blink is enabled, the blink frequency is 1.9 Hz.

Character Position to Memory Location Mapping

Character positions on the screen are identified as row (vertical) and column (horizontal) locations. The first character is displayed in the upper-left corner of the screen, which is location 0,0. To translate between screen positions and the address within the text buffer, use the following formula:

Character code address = start_address + (row * 2 * Y) + (column * 2)

Attribute address = Character code address + 1

Where:

start_address =	Display page start address (see Table 7-3)
row =	0 to 24
column =	0 to 79 (80 X 25 modes) 0 to 39 (40 X 25 modes)
Y =	80 (80 X 25 modes) 40 (40 X 25 modes)

In text modes, the video processor supports multiple display pages. For direct programming, registers R12 and R13 (described later in this chapter) control the display-page start address. Each displayed character requires 2 bytes (character code and attribute byte). Therefore, the 80 x 25 modes require 4000 bytes (80 x 25 x 2) and the 40 x 25 modes require 2000 bytes (40 x 25 x 2). The ROM BIOS also supports multiple display pages and rounds the memory requirements to 4096 bytes and 2048 bytes respectively. Table 7-3 shows the display page addresses as defined by the ROM BIOS.

Table 7-3 Text Mode Display Pages (ROM BIOS)

Address	80 x 25 Display Page	40 x 25 Display Page
B8000H	0	0
B8800H		1
B9000H	1	2
B9800H		3
BA000H	2	4
BA800H		5
BB000H	3	6
BB800H		7

Programmable Cursor

For text modes only, the video controller provides a programmable cursor blink rate and cursor block size. The cursor blink is determined by bits 6-5 of register R10. The cursor blinks with alternate foreground and background color of the character at the cursor position. The cursor block size is controlled by bits 4-0 of registers R10 and R11. Registers R10 and R11 are discussed later in this chapter.

Programmable Character Generator (Font RAM)

The video controller has a 4 Kbyte programmable font RAM. The font RAM can store patterns for 256 characters. Normally, the font RAM is accessible only to the video controller. That is, the font RAM is not mapped into the normal CPU address space. Accessing the font RAM requires that the video mode be one of the text modes 00H, 01H, 02H, or 03H. Bit 4 of control register B (described later in this chapter) controls access to the font RAM. When bit 4 of control register B equals 1, access to the video text buffer is disabled and access to the font RAM is enabled. Only even text buffer addresses are connected to the font RAM. The text buffer to font RAM mapping appears as follows:

Text Buffer Offset	Font Ram Offset
B8000H	0000H (first byte of font RAM)
B8001H	
B8002H	0001H (second byte of font RAM)
B8003H	
B8004H	0002H (third byte of font RAM)
B8005H	
.	.
.	.
.	.
B9FFDH	
B9FFEH	0FFFH (last byte of font RAM)

> NOTE
> The ROM BIOS does not support the use of the font RAM in any graphics video mode.

A character pattern consists of 16 bytes of pixel information. Each byte represents 8 consecutive pixels of a horizontal scan line for the character. The most significant bit (bit 7) corresponds to the left-most pixel (pixel 0). The least significant bit (bit 0) corresponds to pixel 7. Each byte of the character pattern is read or written to an even address. Thus, each character pattern requires 32 bytes of address space and an entire 256-character font requires 8K bytes. To calculate the address of the first byte of a character pattern, use the following formula:

Character pattern start address = B8000H + (character code * 32)

Graphics Mode

Each pixel on the screen is mapped into a bit-field of the corresponding byte in the display buffer. The width of the bit-field can be of 1, 2 or 4 bits depending on whether a 2-color, 4-color, or 16-color format is chosen.

Mapping the Display to Address

The logical display consists of a rectangular array of 200, 252, or 400 scan lines of pixels. For 200 scan line mode, the hardware generates two physical scan lines for each logical scan line. Each scan line is represented by *(M / 8) * n* consecutive bytes, where:

M = Number of pixels per scan line
n = 1, for 1-bit per pixel (2-color display)
n = 2, for 2-bits per pixel (4-color display)
n = 4, for 4-bits per pixel (16-color display)

The memory maps for various graphic formats are shown on the following pages. Each memory map shows two or more blocks of memory that refer to:

$(L \text{ MOD } P) = R$

Where:

L is the desired scan line
P is the number of memory blocks for the current video mode
R is the remainder of the division *L / P*

The remainder, *R*, specifies the memory block for a particular scan line.

320 x 200 4-Color Mode

ROM BIOS Video Modes: 04H and 05H - Industry-Standard

In 4-color mode, a single byte corresponds to 4 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-3 for the memory organization. See Figure 7-4 for the pixel to bit-field map.

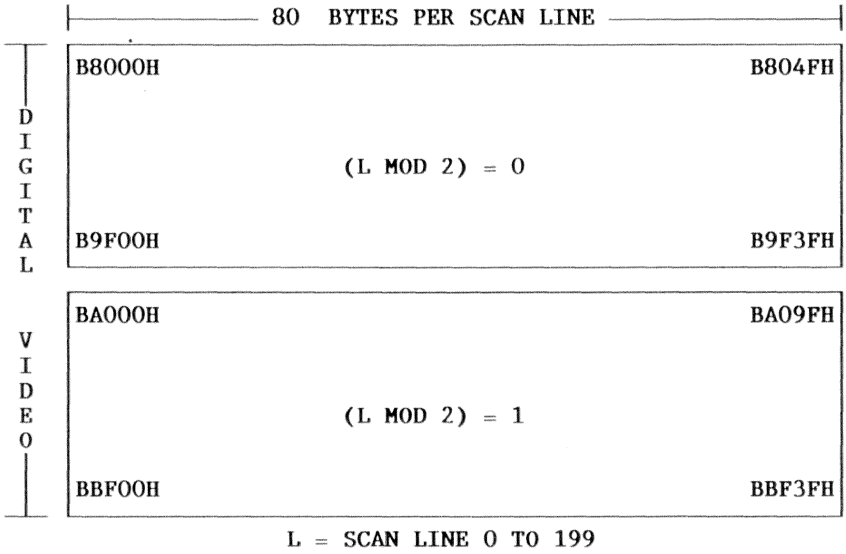


Figure 7-3 Memory Organization for 320 x 200 4-Color Mode

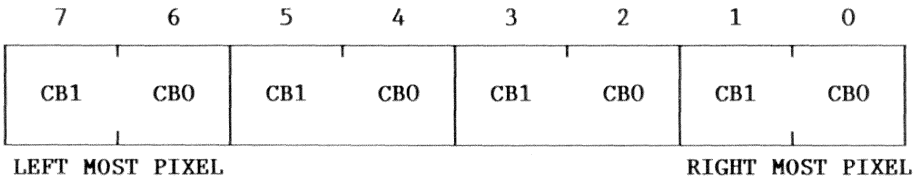


Figure 7-4 Pixel to Bit-Field Map for 4-Color Mode

320 x 200 16-Color Mode

No ROM BIOS Support - DIGITAL Extended

In 16-color mode, a single byte corresponds to 2 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-5 for the memory organization. See Figure 7-6 for the pixel to bit-field map.

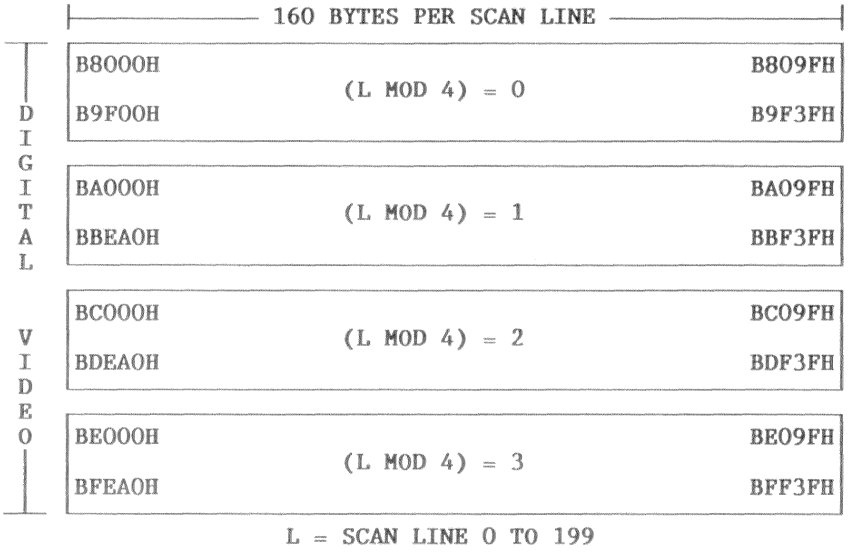


Figure 7-5 Memory Organization for 320 x 200 16-Color Mode

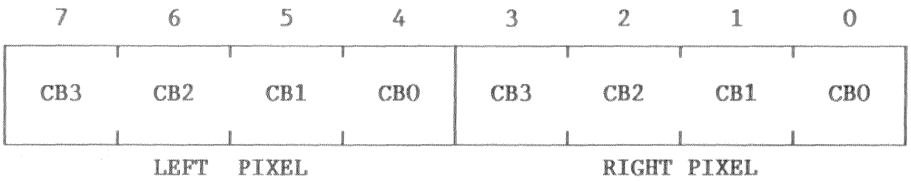


Figure 7-6 Pixel to Bit-Field Map for 16-Color Mode

640 x 200 2-Color Mode

ROM BIOS Video Mode: 06H - Industry-Standard

In 2-color mode, a single byte corresponds to 8 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-7 for the memory organization. See Figure 7-8 for the pixel to bit-field map.

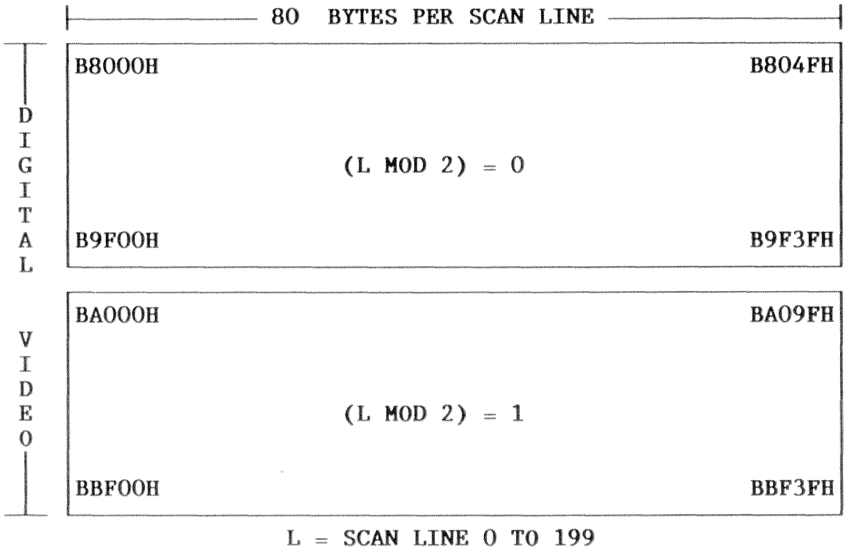


Figure 7-7 Memory Organization for 640 x 200 2-Color Mode

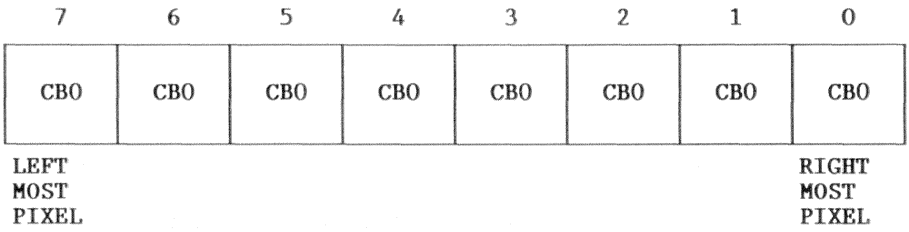


Figure 7-8 Pixel to Bit-Field Map for 2-Color (Monochrome) Mode

640 x 200 4-Color Mode

No ROM BIOS Support - DIGITAL Extended

In 4-color mode, a single byte corresponds to 4 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-9 for the memory organization. See Figure 7-10 for the pixel to bit-field map.

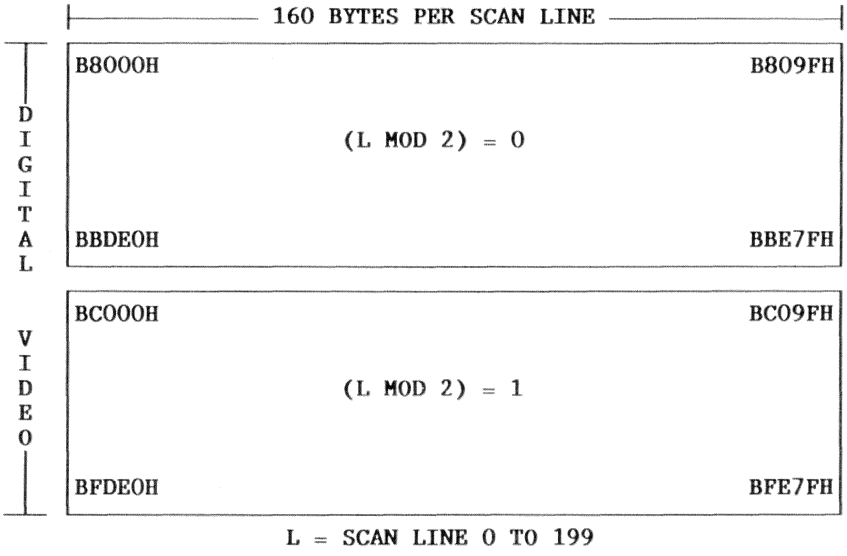


Figure 7-9 Memory Organization for 640 x 200 4-Color Mode

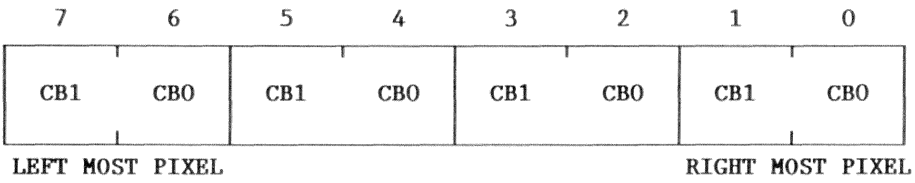


Figure 7-10 Pixel to Bit-Field Map for 4-Color Mode

640 x 400 2-Color Mode

ROM BIOS Video Mode: D0H - DIGITAL Extended

In 2-color mode, a single byte corresponds to 8 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-11 for the memory organization. See Figure 7-12 for the pixel to bit-field map.

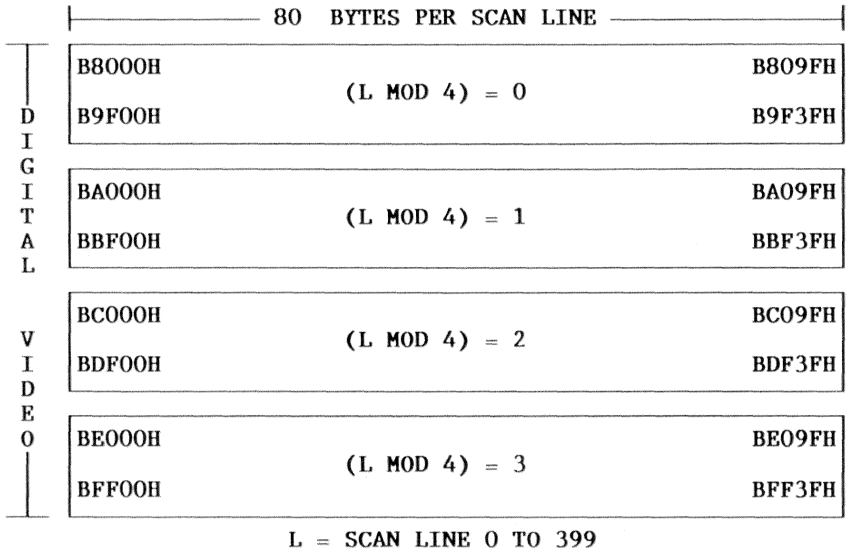


Figure 7-11 Memory Organization for 640 x 400 2-Color Mode

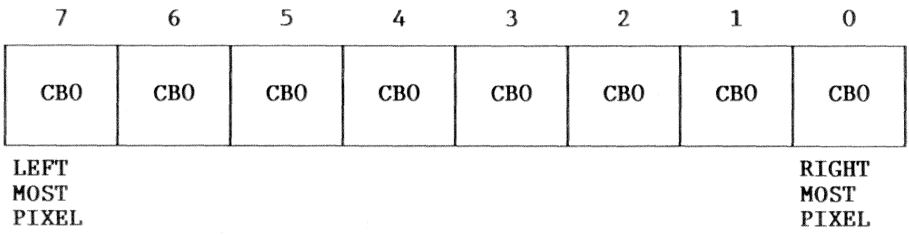


Figure 7-12 Pixel to Bit-Field Map for 2-Color Mode

640 x 400 4-Color Mode

ROM BIOS Video Mode: D1H - DIGITAL Extended

In 4-color mode, a single byte corresponds to 4 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-13 for the memory organization. See Figure 7-14 for the pixel to bit-field map.

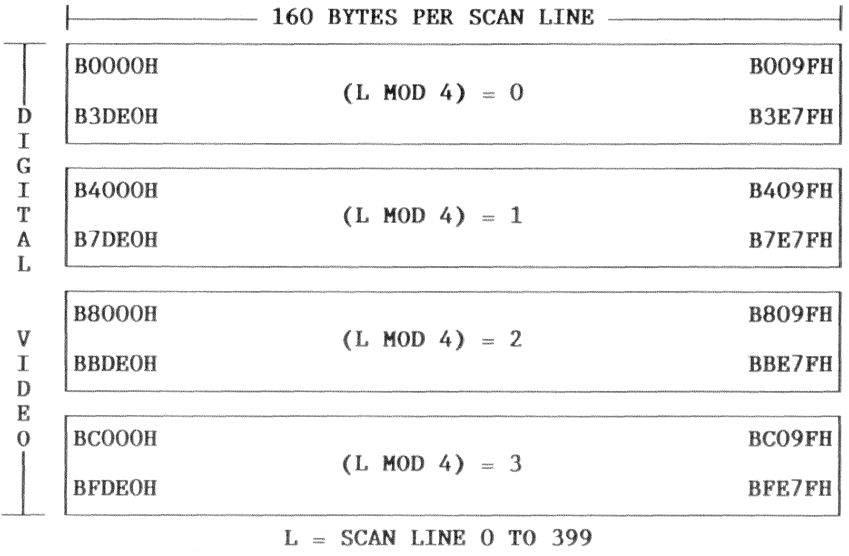


Figure 7-13 Memory Organization for 640 x 400 4-Color Mode

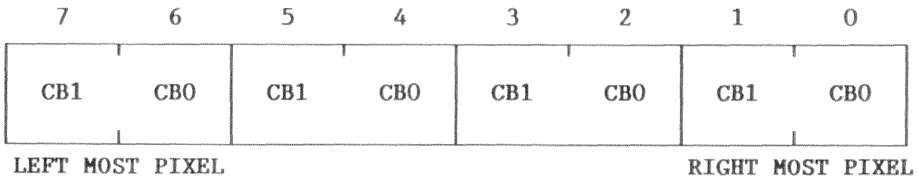


Figure 7-14 Pixel to Bit-Field Map for 4-Color Mode

800 x 252 4-Color Mode

ROM BIOS Video Mode: D2H (Limited ROM BIOS Support) - DIGITAL Extended

In 4-color mode, a single byte corresponds to 4 consecutive pixels on the screen with the most significant bit of the byte corresponding to the left of the screen. See Figure 7-15 for the memory organization. See Figure 7-16 for the pixel to bit-field map.

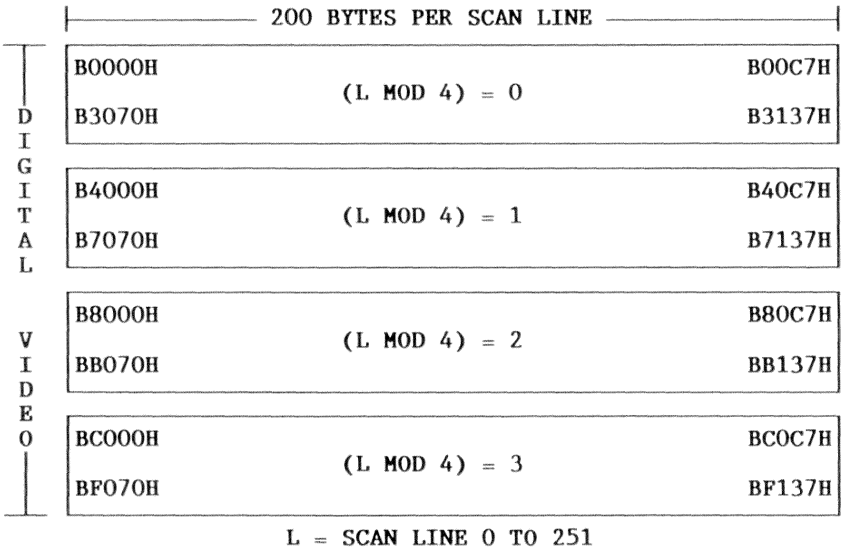


Figure 7-15 Memory Organization for 800 x 252 4-Color Mode

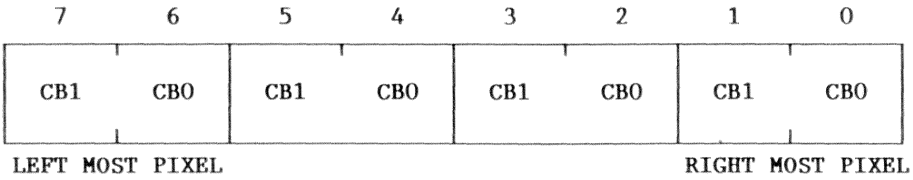


Figure 7-16 Pixel to Bit-Field Map for 4-Color Mode

Video Look-Up Table

The video processor has a video look-up table (VLT) that translates attribute or graphic color data. The VLT is arranged as 16 words of IRGB output data. Each location corresponds to one of the 16 possible colors. When the video controller accesses video memory, the attributes or graphic data are used as an offset into the VLT. The contents of that location in the VLT are sent to the video output circuit. Because the VLT has only 16 entries, the VLT can alter the color interpretation of the bit map without rewriting every pixel.

For 2-color mode graphics (640 x 400 or 640 x 200), the foreground color (pixel equals 1) is determined by the color-select register bits 3-0. The background color (pixel equals 0) is determined by the contents of VLT entry 0. The color select register is described later in this chapter.

Normally, the VLT is accessible only to the video controller. That is, the VLT is not mapped into the normal CPU address space. Accessing the VLT requires that the video mode be one of the text modes 00H, 01H, 02H, or 03H. Bit 2 of control register B (described later in this chapter) controls access to the VLT. When bit 2 of control register B equals 1, access to the video text buffer is disabled and access to the VLT is enabled.

NOTE
Only write access to the VLT is enabled. To read the VLT indirectly, program the video processor for 320 x 200 16-color mode. For each of the 16 possible colors (00H-0FH):

1. Write the same color value to each pixel.
2. Wait until the display is inactive (register B bit 7 equals 1)
3. Disable CPU interrupts (CLI instruction)
4. Wait until the display is active (register B bit 7 equals 0)
5. Status register A bits 7-4 (IRGB) are equal to the contents of the VLT location specified by the color value.
6. Enable CPU interrupts (STI instruction)

Only even-text buffer addresses are connected to the VLT. The text buffer to VLT mapping appears as follows:

Text Buffer Offset	VLT Offset	Text Buffer Offset	VLT Offset
B8000H	0000H	B8010H	0008H
B8001H		B8011H	
B8002H	0001H	B8012H	0009H
B8003H		B8013H	
B8004H	0002H	B8014H	000AH
B8005H		B8015H	
B8006H	0003H	B8016H	000BH
B8007H		B8017H	
B8008H	0004H	B8018H	000CH
B8009H		B8019H	
B800AH	0005H	B801AH	000DH
B800BH		B801BH	
B800CH	0006H	B801CH	000EH
B800DH		B801DH	
B800EH	0007H	B801EH	000FH
B800FH		B801FH	

Text mode attributes are referenced in the order IRGB, but the VLT addressing and contents are referenced in the order RGBI. To calculate the offset accessed by any IRGB value, use the following bit values:

Bit Value	Attribute
0	I (Intensity)
1	B (Blue)
2	G (Green)
3	R (Red)

Thus, a text attribute of intensified red (IRGB = C0H) accesses location 09H of the 16 locations in the VLT.

On power-up or system reset, the VLT is initialized to the values in Table 7-4. The VLT values defined in Table 7-4 support video modes 00H, 01H, 02H, 03H, 04H, 05H, 06H and D0H. When changing from any of these modes to video mode D1H or D2H, initialize the VLT to the values defined in Table 7-5.

Table 7-4 Default VLT Contents

Offset				Contents				Color	Intensity
A3 R	A2 G	A1 B	A0 I	D3 R	D2 G	D1 B	D0 I		
0	0	0	0	0	0	0	0	Black	0
0	0	0	1	0	0	0	1	Gray	1
0	0	1	0	0	0	1	0	Blue	2
0	0	1	1	0	0	1	1	Light blue	3
0	1	0	0	0	1	0	0	Green	4
0	1	0	1	0	1	0	1	Light green	5
0	1	1	0	0	1	1	0	Cyan	6
0	1	1	1	1	1	1	0	White	14
1	0	0	0	1	0	0	0	Red	8
1	0	0	1	1	0	0	1	Light red	9
1	0	1	0	1	0	1	0	Magenta	10
1	0	1	1	1	0	1	1	Light magenta	11
1	1	0	0	1	1	0	0	Brown	12
1	1	0	1	1	1	0	1	Yellow	13
1	1	1	0	0	1	1	1	Light cyan	7
1	1	1	1	1	1	1	1	Intense white	15

Table 7-5 VLT Contents for Video Modes D1H and D2H

Offset				Contents				Color	Intensity
A3 R	A2 G	A1 B	A0 I	D3 R	D2 G	D1 B	D0 I		
0	0	0	0	0	0	0	0	Black	0
0	0	0	1	0	1	0	0	Green	4
0	0	1	0	1	0	0	0	Red	8
0	0	1	1	0	1	1	1	Light cyan	7
0	1	0	0	Not Used					
0	1	0	1	Not Used					
0	1	1	0	Not Used					
0	1	1	1	Not Used					
1	0	0	0	Not Used					
1	0	0	1	Not Used					
1	0	1	0	Not Used					
1	0	1	1	Not Used					
1	1	0	0	Not Used					
1	1	0	1	Not Used					
1	1	1	0	Not Used					
1	1	1	1	Not Used					

Video System Registers

Table 7-6 lists the video processor input/output (I/O) registers.

Table 7-6 Video Processor I/O Registers

Address	Width	R/W	Register Name	Compatibility
03D0H	4-0	W	CRTC Index Register	DIGITAL Extended
03D1H	7-0	R/W	CRTC Data Register	DIGITAL Extended
03D4H	4-0	W	CRTC Index Register	Industry-Standard
03D5H	7-0	R/W	CRTC Data Register	Industry-Standard
03D8H	7-0	W	Control Register A	Industry-Standard
03D9H	7-0	W	Color Select Register	Industry-Standard
03DAH	7-0	R	Status Register A	Industry-Standard
03DDH	7-0	R	Status Register B	DIGITAL Extended
03DEH	7-0	R	Write Data Register	DIGITAL Extended
03DFH	7-0	W	Control Register B	DIGITAL Extended
0C80H	7-0	R/W	Special Purpose Register	DIGITAL Extended

Special Purpose Register (0C80H)

7	6	5	4	3	2	1	0
WRITE PROTECT	TRACK 0	INDEX	SPEED	DISABLE VIDEO	SPLIT BAUD	DISABLE COMM	SPEED SELECT

Bit	R/W	Description
7	R	Write protect 0 = Selected diskette drive is not write protected 1 = Selected diskette drive is write protected
6	R	Track 0 0 = Head of selected diskette drive is not at track 0 1 = Head of selected diskette drive is at track 0
5	R	Index 0 = Index hole not in position for selected diskette drive 1 = Index hole in position for selected diskette drive
4	R	Speed Indicator 0 = Modem control speed select asserted 1 = Modem control speed select not asserted
3	R/W	Disable Video 0 = Video controller disabled 1 = Video controller enabled
2	R/W	Split Baud Rates 0 = (Receive = Transmit = programmed) 1 = (Receive = 1200) (Transmit = programmed)
1	R/W	Disable Communications 0 = Integral communications ports connected to I/O address space 1 = Integral communications ports disconnected from I/O address space
0	R/W	Speed Select 0 = Speed select asserted 1 = Speed select not asserted

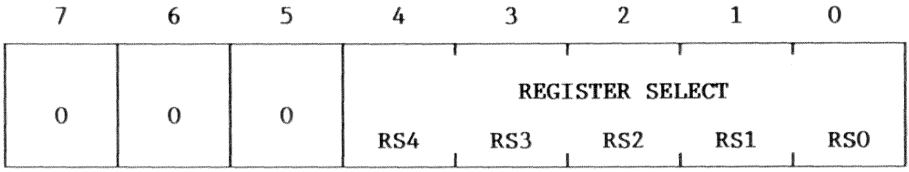
The special purpose register is located at I/O address 0C80H. When bit 3 equals 0, the entire DIGITAL video system is disconnected from the memory and I/O address space. This allows the installation and use of industry-standard video adapters in the VAXmate workstation.

If the ROM BIOS finds an industry-standard video adapter during the power-up sequence, the ROM BIOS clears bit 3 of the special purpose register. This allows the industry-standard video adapter to function without conflict.

CRTC Registers

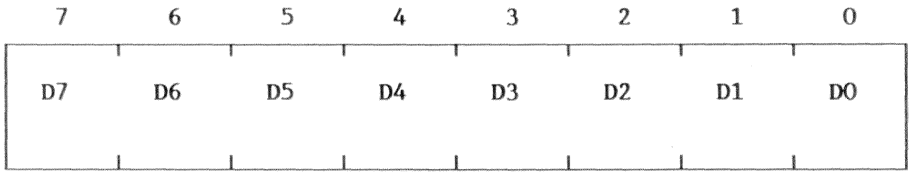
The CRT controller (CRTC) has two registers, the index and data registers, that are accessible in the CPU I/O space. Writing a value to the index register selects one of the 18 internal registers R0-R17. The selected register is read or written through the data register.

Index Register (03D0H/03D4H)



Bit	R/W	Description
7-5	W	Always 0
4-0	W	REGISTER SELECT (RS4-RS0) A value between 0 and 17 written to this register selects one of the corresponding internal registers (R0-R17).

Data Register (03D1H/03D5H)



Bit	R/W	Description
7-0	R/W	Data and width are dependent upon the register selected by the index register. To determine if the data register can be read or written, see the description of the register selected by the index register.

The index and data registers can be accessed through two sets of I/O ports. The industry-standard set is 03D4H (index) and 03D5H (data). The DIGITAL extended set is 03D0H (index) and 03D1H (data). Data written to the industry-standard set pass through a translation ROM and then go to the CRTC. Data written to the DIGITAL extended set go directly to the CRTC.

The translation ROM converts CRTC parameters, for an industry-standard color graphics adapter, to values that are correct for the extended capabilities of the DIGITAL video system. Thus, applications that directly program the CRTC of an industry-standard color graphics adapter function correctly.

Table 7-7 lists the CRTC internal registers and their functions. Table 7-8 lists the corresponding parameters for the video modes defined in Table 7-1. The parameters listed in Table 7-8 are written to the CRTC through the DIGITAL extended I/O ports 03D0H (index) and 03D1H (data).

Table 7-7 CRTC Internal Registers

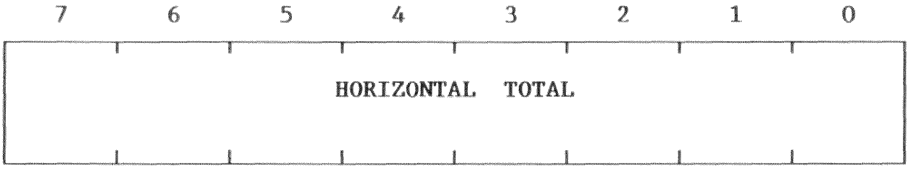
Register	Index	R/W	Description
R0	00H	W	Horizontal total
R1	01H	W	Horizontal displayed
R2	02H	W	Horiz sync position
R3	03H	W	Sync width
R4	04H	W	Vertical total
R5	05H	W	Vertical total adjust
R6	06H	W	Vertical displayed
R7	07H	W	Vertical sync position
R8	08H	W	Interlace/Skew
R9	09H	W	Max scan line address
R10	0AH	W	Cursor start
R11	0BH	W	Cursor end
R12	0CH	R/W	Start address (High byte)
R13	0DH	R/W	Start address (Low byte)
R14	0EH	R/W	Cursor address (High byte)
R15	0FH	R/W	Cursor address (Low byte)
R16	10H	R	Light pen (High byte) *
R17	11H	R	Light pen (Low byte) *

* The DIGITAL video system does not support light pens.

Table 7-8 CRTC Register Values

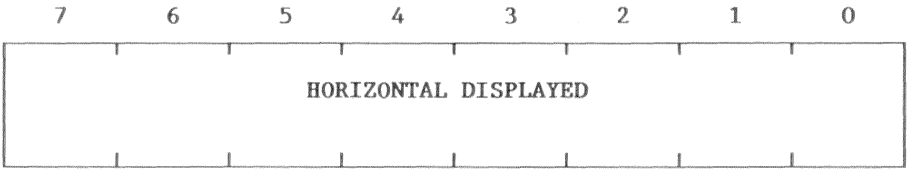
Register	320 x 200 16-color Graphics	800 x 252 4-color Graphics	640 x 400 4-color Graphics	320 x 200 4-color 640 x 200 2-color Graphics	80 x 25 Text	40 x 25 Text
R0	69H	83H	69H	34H	69H	34H
R1	50H	64H	50H	28H	50H	28H
R2	58H	6DH	58H	2CH	58H	2DH
R3	58H	5AH	58H	54H	58H	54H
R4	36H	6DH	6DH	6DH	1AH	1AH
R5	00H	01H	00H	00H	08H	08H
R6	32H	3FH	64H	64H	19H	19H
R7	33H	53H	66H	66H	19H	19H
R8	40H	40H	42H	40H	40H	40H
R9	07H	03H	03H	03H	0FH	0FH
R10	00H	00H	00H	00H	00H	00H
R11	0FH	0FH	0FH	0FH	0FH	0FH
R12	00H	00H	00H	00H	00H	00H
R13	00H	00H	00H	00H	00H	00H
R14	00H	00H	00H	00H	00H	00H
R15	00H	00H	00H	00H	00H	00H

Register R0



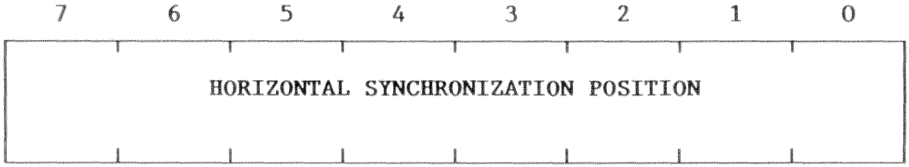
Bit	R/W	Description
7-0	W	HORIZONTAL TOTAL This register determines the horizontal synchronization frequency. It is the number of displayed characters (R1) plus the retrace (in character times) minus one.

Register R1



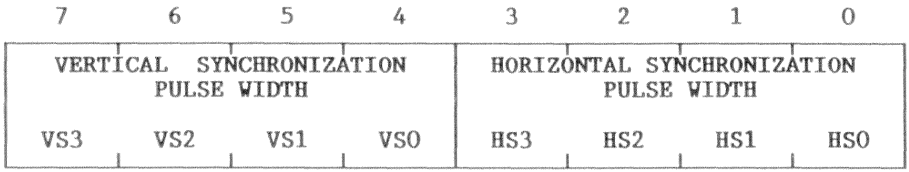
Bit	R/W	Description
7-0	W	HORIZONTAL DISPLAYED This register determines the number of displayed characters on a line. The value in R1 must be less than the value in R0.

Register R2



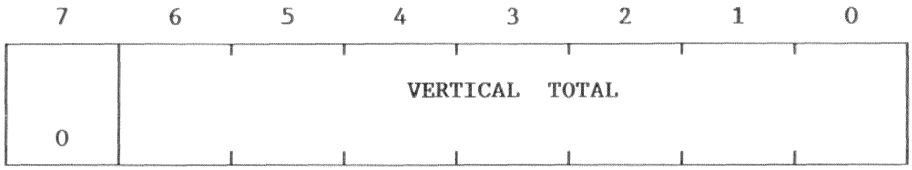
Bit	R/W	Description
7-0	W	HORIZONTAL SYNCHRONIZATION POSITION This register determines the position of the horizontal synchronization delay and the horizontal scan delay. When this value is increased, the display shifts left. When this value is decreased, the display shifts right.

Register R3



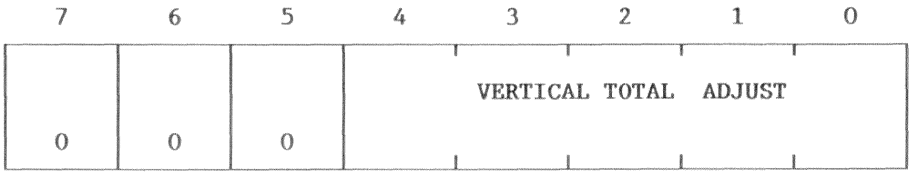
Bit	R/W	Description
7-4	W	VERTICAL SYNCHRONIZATION PULSE WIDTH A value of 1-15 produces a pulse width of the indicated number of scan-line periods. A value of zero produces a pulse width of 16 scan-line periods.
3-0	W	HORIZONTAL SYNCHRONIZATION PULSE WIDTH A value of 1-15 produces a pulse width of the indicated number of character periods. If the value equals 0, then a horizontal synchronization pulse is not provided.

Register R4



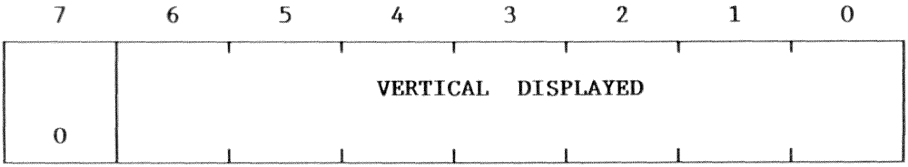
Bit	R/W	Description
7	W	Always 0
6-0	W	VERTICAL TOTAL This value determines the vertical synchronization frequency. It is the number of displayed character lines plus the retrace (in character line times) minus one.

Register R5



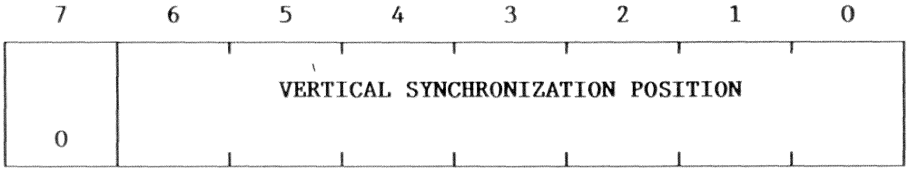
Bit	R/W	Description
7-5	W	Always 0
4-0	W	VERTICAL TOTAL ADJUST This value is the number of scan-line periods required, in addition to R4, to produce a vertical synchronization frequency of exactly 50Hz or 60Hz.

Register R6



Bit	R/W	Description
7	W	Always 0
6-0	W	VERTICAL DISPLAYED This value specifies the number of displayed character lines. It must be less than the value in R4.

Register R7



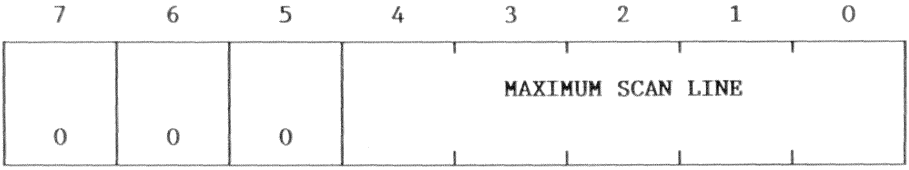
Bit	R/W	Description
7	W	Always 0
6-0	W	VERTICAL SYNCHRONIZATION POSITION This value determines the position of the vertical synchronization delay and the vertical scan delay. When this value is increased, the display shifts up. When this value is decreased, the display shifts down.

Register R8

7	6	5	4	3	2	1	0
CURSOR SKEW		DISPLAY ENABLE SKEW				INTERLACE MODE	
CS1	CS0	DS1	DS0	0	0	IM1	IM0

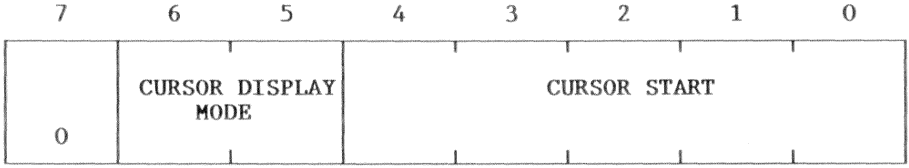
Bit	R/W	Description
7-6	W	CURSOR SKEW 00 = No skew 01 = One character skew 10 = Two character skew 11 = Invalid value
5-4	W	DISPLAY ENABLE SKEW 00 = No skew 01 = One character skew 10 = Two character skew 11 = Invalid value
3-2	W	Always 0
1-0	W	INTERLACE MODE 00 = Normal mode 01 = Interlace synchronization mode 10 = Normal mode 11 = Interlace synchronization and video mode

Register R9



Bit	R/W	Description
7-5	W	Always 0
4-0	W	MAXIMUM SCAN LINE This value specifies one less than the number of scan lines per character line including spacing.

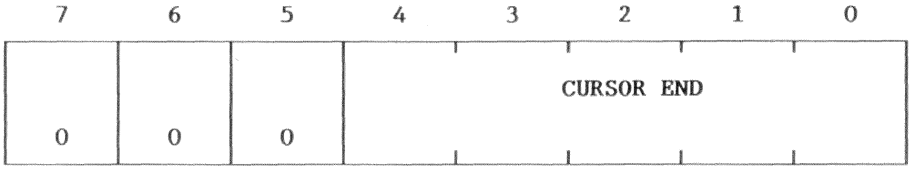
Register R10



Bit	R/W	Description
7	W	Always 0
6-5	W	CURSOR DISPLAY MODE 00 = Nonblinking cursor 01 = Cursor not displayed 10 = Blinking cursor (3.75 Hz) 11 = Blinking cursor (1.875 Hz)
4-0	W	CURSOR START This value specifies the scan line, within the character cell, on which the cursor starts. A value of 0 starts the cursor at the top of the character cell.

This register is meaningful only in text video modes.

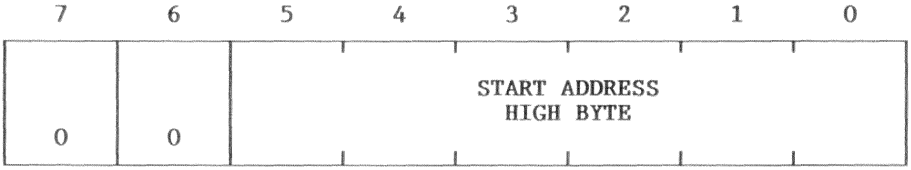
Register R11



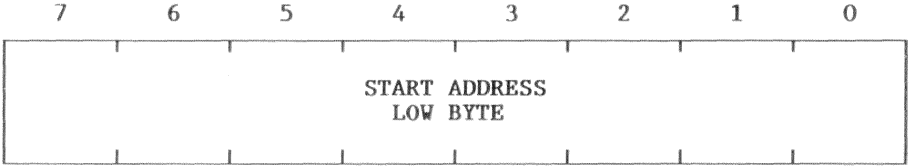
Bit	R/W	Description
7-5	W	Always 0
4-0	W	CURSOR END This value specifies the scan line, within the character cell, on which the cursor ends. A value of 15 ends the cursor at the bottom of the character cell.

This register is meaningful only in text video modes.

Register R12

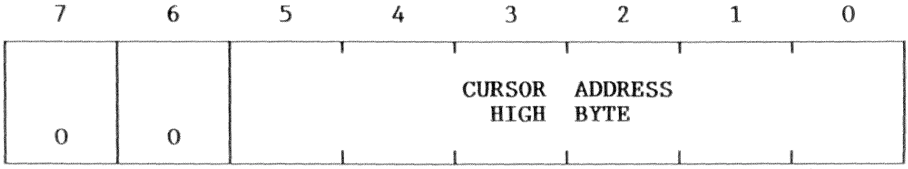


Register R13

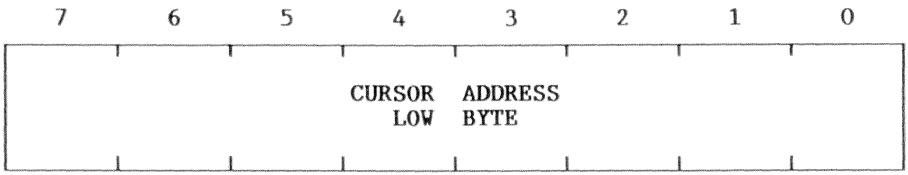


R12 and R13 are a write-only register pair that determine which part of the video RAM is used to generate the display. The address in R12 and R13 must be an even value. This address points to the first character position.

Register R14

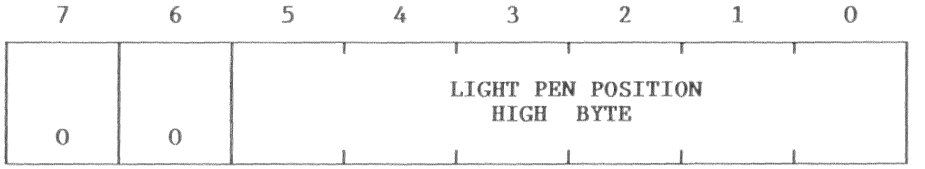


Register R15

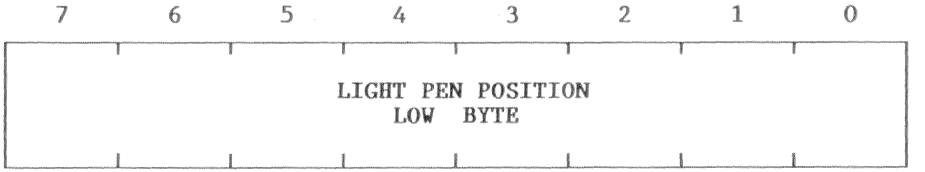


R14 and R15 are a read/write register pair that determine the location of the cursor as an offset from the beginning of video RAM. The address in R14 and R15 must be an even value. The address points to the character byte of a character byte/attribute byte pair.

Register R16



Register R17



R16 and R17 are a read only register pair that capture the CRTC refresh address when the light pen strobe pin is pulsed.

NOTE
The VAXmate workstation does not support the use of light pens.

Status Register A (03DAH)

7	6	5	4	3	2 1	0
VIDEO I	VIDEO R	VIDEO G	VIDEO B	VSYNC	LIGHT PEN	RETRACE

Bit	R/W	Description
7	R	VIDEO I - Video Intensity Signal 0 = Video intensity signal inactive 1 = Video intensity signal active
6	R	VIDEO R - Video Red Signal 0 = Video red signal inactive 1 = Video red signal active
5	R	VIDEO G - Video Green Signal 0 = Video green signal inactive 1 = Video green signal active
4	R	VIDEO B - Video Blue Signal 0 = Video blue signal inactive 1 = Video blue signal active
3	R	VSYNC - Vertical Synchronization 0 = Vertical synchronization inactive 1 = Vertical synchronization active
2-1	R	LIGHT PEN (Contents undefined)
0	R	RETRACE (Horizontal or vertical) 0 = Display active 1 = Retrace period Because the dual-port RAM eliminates display interference caused by accessing video memory, checking this bit is not required. For those programs that do check, this bit flips with each read. Because a retrace period appears to be in effect every other time it is checked, this has the effect of speeding up video memory accesses.

Status Register B (03DDH)

7	6	5	4	3	2	1	0
VIDEO BLANK	CR-B3	CR-B5	CR-A4	CR-A1	CR-A0	WRITE CHECK	PORT CHECK

Bit	R/W	Description
7	R	VIDEO BLANK 0 = Video is in an active display state 1 = Video is in a blanking state
6	R	CR-B3 Control Register B bit 3 (Display enabled)
5	R	CR-B5 Control Register B bit 5 (Control register A bit 3 enable)
4	R	CR-A4 Control Register A bit 4 (Mode bit 2)
3	R	CR-A1 Control Register A bit 1 (Mode bit 1)
2	R	CR-A0 Control Register A bit 0 (Mode bit 0)
1	R	WRITE CHECK 0 = Since the write data register (03DEH) was last read, an I/O write to port 03D4H or 03D5H has not occurred. 1 = Since the write data register (03DEH) was last read, an I/O write to port 03D4H or 03D5H has occurred. This bit is cleared by reading the write data register (03DEH).
0	R	PORT CHECK 0 = Of the pair, 03D4H and 03D5H, 03D4H was the last port written. 1 = Of the pair, 03D4H and 03D5H, 03D5H was the last port written. This bit is used in conjunction with bit 1.

Write Data Register (03DEH)

7	6	5	4	3	2	1	0

Bit	R/W	Description
7-0	R	Contains the last data written into the CRTC through register 03D4H or 03D5H. Status register B bits 1-0 indicate which port the data was written to. Reading this register clears status register B bit 1.

Color Select Register (03D9)

7	6	5	4	3	2	1	0
0	0	CPS	SIC	I	R	G	B

Bit	R/W	Description
7-6	W	Always 0, always ignored
5	W	CPS - Color Palette Select (See Table 7-9 and Table 7-10)
4	W	SIC - Select Intensified Colors (See Table 7-9 and Table 7-10)
3	W	I - Intensity (See Table 7-9)
2	W	R - Red (See Table 7-9)
1	W	G - Green (See Table 7-9)
0	W	B - Blue (See Table 7-9)

The use of the color select register bits depends on the current video mode. Table 7-9 describes the bit meanings for the affected modes. Table 7-10 describes the color palettes selected by bits 5-4 (CPS and SIC).

Table 7-9 Color Select Register Bit Assignments

Bit	Text Modes	320 x 200 4-Color Graphics	640 x 200 x 2-Color 640 x 400 x 2-Color Graphics
7	Ignored	Ignored	Ignored
6	Ignored	Ignored	Ignored
5	Ignored	CPS	Ignored
4	Ignored	SIC	Ignored
3-0	Border color	Border and background color	Foreground color

NOTE
For the VAXmate workstation, the border color is always black.

Table 7-10 Color Palettes Selected by CPS and SIC

Color cb1	Bit cb0	CPS = 0 SIC = 0	CPS = 1 SIC = 0	CPS = 0 SIC = 1	CPS = 1 SIC = 1
0	0	Background	Background	Background	Background
0	1	Green	Cyan	Light green	White
1	0	Red	Magenta	Light red	Light magenta
1	1	Brown	Light cyan	Light yellow	Intense white

The color bits (cb1/cb0) in Table 7-10 are any 2 bits that describe a pixel color in the 320 x 200 4-color video mode.

Control Register A (03D8H)

7	6	5	4	3	2	1	0
0	0	BLINK ENABLE	MODE BIT 2	DISPLAY ENABLE	0	MODE BIT 1	MODE BIT 0

Bit	R/W	Description
7-6	W	Always 0
5	W	BLINK ENABLE 0 = Text mode background intensity bit (I) remains in effect 1 = Text mode background intensity bit (I) becomes blink bit
4	W	MODE BIT 2 (See Table 7-11)
3	W	DISPLAY ENABLE If control register B (03DFH) bit 5 equals 0, this bit is ignored. If control register B bit 5 equals 1, the following is true: 0 = Display disabled 1 = Display enabled
2	W	Always 0 (Reserved)
1	W	MODE BIT 1 (See Table 7-11)
0	W	MODE BIT 0 (See Table 7-11)

Table 7-11 lists the video modes selected by the mode bits in control registers A and B.

Table 7-11 Selecting Video Modes

Control Register A Mode Bits 2	1	0	Control Register B Bit 7	Mode	Compatibility
0	0	0	0	40 x 25 Text	Industry-standard
0	0	1	0	80 x 25 Text	Industry-standard
0	1	0	0	320 x 200 x 4 color graphics	Industry-standard
0	1	1	0	320 x 200 x 16 color graphics	DIGITAL extended
1	0	0	0	640 x 400 x 2 color graphics	DIGITAL extended
1	0	1		640 x 200 x 4 color graphics	DIGITAL extended
1	1	0	0	640 x 200 x 2 color graphics	Industry-standard
1	1	1	0	640 x 400 x 4 color graphics	DIGITAL extended
1	1	1	1	800 x 252 x 4 color graphics	DIGITAL extended

Control Register B (03DFH)

7	6	5	4	3	2	1	0
MONITOR MODE	SCREEN SAVER	CR–A5 ENABLE	FONT RAM ENABLE	DISPLAY ENABLE	VLT ENABLE	0	0

Bit	R/W	Description
7	W	MONTIOR MODE 0 = 400 scan lines 1 = 252 scan lines
6	W	SCREEN SAVER Toggling this bit to 0 and then back to 1 blanks the display. The next memory or I/O access to the video address space reenables the display. Program to 1 for normal operation.
5	W	CR-A5 ENABLE 0 = Control register A bit 5 ignored 1 = Control register A bit 5 enabled
4	W	FONT RAM ENABLE 0 = Access to font RAM disabled 1 = Access to font RAM enabled
3	W	DISPLAY ENABLE 0 = Display blanked 1 = Display enabled
2	W	VLT ENABLE 0 = Access to video look-up table disabled 1 = Access to video look-up table enabled
1-0	W	Always 0

Monitor Interface

Table 7-12 lists the monitor interface signals. These signals are applicable to both a monochrome or a color monitor.

Table 7-12 Monitor Interface Signals

Pin No.	Signal Description
1	Horizontal synchronization (active low)
2	Vertical synchronization (active low)
3	Intensity Video (active high)
4	Red Video (active high)
5	Green Video (active high)
6	Blue Video (active high)
7	400/252 select (low for 400 scans; high for 252 scans)
8	(reserved)
9	Signal ground
10	+5 return
11	+5V dc (200 mA max.)
12	(spare)

Monitor Specification Summary

The following are specifications for the monochrome monitor on the VAXmate workstation:

CRT	340 mm (14 in) diagonal, amber or green phosphor
Active Display	240 mm horizontal by 150 mm vertical (9.5 x 6 in)
Resolution	640 pixels horizontal by 400 pixels vertical 800 pixels horizontal by 252 pixels vertical
Horizontal scan rate	26.40 kHz (640 x 400) 26.49 kHz (800 x 252)
Vertical scan rate	60 Hz noninterlaced
Video Bandwidth	22.384 MHz (640 x 400) 27.984 MHz (800 x 252)

Programming Example

The following programming example demonstrates:

- Programming the video controller for a specific mode
- Writing the video look-up table
- Reading and writing the font RAM
- Displaying characters in text and graphics modes

NOTE

Whenever possible, ROM BIOS Interrupt 10H video calls are preferred over direct programming of the video hardware.

Do not mix ROM BIOS calls and direct programming of the hardware.

Before directly programming the hardware, use ROM BIOS calls to determine the state of the video system. On exit, use the ROM BIOS to restore the previous state.

CAUTION

Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

The example provides routines as described in the following list:

get_mode_p	Returns a pointer to a table of data about the indicated mode.
get_mess_p	Returns a pointer to character string that describes the indicated mode.
w_vlt	Writes the video look-up table.
r_w_font	Reads or writes the font ram.
mode_init	Initializes the video controller and mode registers from a table of data.
mv_cursor	Positions the cursor to the indicated row and column position.
cursor_on	Positions the cursor and makes it visible.
cursor_off	Makes the cursor invisible.
set_mode	Sets the current mode, clears video memory and enables the display.
screen_on	Enables or disables the display.
clear_vid_mem	Clears the screen by writing the appropriate values to video memory.
do_border	Forms a border around the screen (like a picture frame), by displaying the letter E at the extreme positions of the screen. It also displays a message, in the center of the screen, that describes the current mode.
disp_g	Displays, in graphics mode, the pixel representation of a character.
disp_t	Displays characters for text mode.
video	Sets up the conditions and executes the examples.

This page is intentionally blank.

The constants defined in this example are in the include file VIDEO.H. The other include files, EXAMPLE.H and KYB.H, support the example, but are not pertinent to the video section.

The constant values TRUE and FALSE are used as calling parameters for several routines.

The constant values CRTC_INDEX through CTRL_REGB define the addresses, in input/output space, of the registers used to control the video mode and attributes. These registers are described in Table 7-7.

The constant value VB8 defines the industry-standard start address for color graphics video memory. The constant value VB0 defines the VAXmate extended start address for color graphics video memory. These values are far pointers expressed as long integers.

The structure type VLT defines the organization of the video look-up table. When access is enabled, the first byte of the video look-up table is written at B800H:0000H (segment:offset). The next byte is written at B800H:0002H (segment:offset). Thus, the video look-up table can be defined as an array of 16 structures of type VLT. Notice that this organization should be used only for accessing the video look-up table. It should not be used when reserving space, because 50 percent of the space would be wasted.

The structure type FONT defines the organization of the font RAM. When access is enabled, the first byte of the font RAM is read or written at B800H:0000H (segment:offset). The next byte is read or written at B800H:0002H (segment:offset). Each character font requires 16 bytes. The font for each of the possible 256 characters can be defined. Thus, the font RAM can be defined as a two-dimensional array of structures of type FONT, where the first subscript is 256 and the second subscript is 16. Notice that this organization should be used only for accessing the font RAM. It should not be used when reserving space, because half the space would be wasted.

The structure type M_TABLE defines data or pointers to data that is required to program the various video modes. Later in the example, an array of structures of type M_TABLE is defined. The values used are gathered from information provided earlier in this chapter.

```
#include "video.h"
#include "example.h"
#include "kyb.h"

/*************************************************************************/
/*        Declare constants and structures used in examples              */
/*************************************************************************/

#define TRUE        0xffff                          /* True is nonzero */
#define FALSE       0x0000                          /* False is zero   */
#define CRTC_INDEX  0x03d0          /* crtc index register in i/o space */
#define CRTC_DATA   0x03d1          /* crtc data register in i/o space  */
#define CTRL_REGA   0x03d8          /* control register A in i/o space  */
#define COLR_SELC   0x03d9        /* color select register in i/o space */
#define STAT_REGA   0x03da          /* status register A in i/o space   */
#define STAT_REGB   0x03dd          /* status register B in i/o space   */
#define CTRL_REGB   0x03df          /* control register B in i/o space  */
#define VB8     0xb8000000L     /* normal base address of video memory */
#define VB0     0xb0000000L                   /* extended base address */

typedef struct
{
    unsigned char   vlt_byte;           /* vlt entries at even address */
    unsigned char   skip_byte;             /* skip byte at odd address */
} VLT;

typedef struct
{
    unsigned char   font_byte;        /* font entries at even address */
    unsigned char   skip_byte;           /* skip byte at odd address */
} FONT;

typedef struct
{
    unsigned char   *ct;                    /* pointer into crtc_table */
    unsigned char   *vt;                    /* pointer into vlt_table  */
    unsigned char   cra;     /* control register A value (Table 7-11) */
    unsigned char   crb;                   /* control register B value */
    unsigned char   csr;  /* color select register value (Table 7-10) */
    long           base;                /* segment:offset base address */
    unsigned int    nsp;                       /* number of scan pages */
    unsigned int    sps;                            /* scan page size */
    unsigned int    cb;                       /* color bits per pixel */
    unsigned int    width;   /* bytes per character line or scan line */
    unsigned int    length;   /* in chars or pixels depending on mode */
} M_TABLE;
```

The array *crtc* defines six sets of CRT controller initialization values. The values used are those listed in Table 7-8, which supports all of the defined VAXmate video modes. Notice that each state supports more than one video mode. In that case, the distinguishing factor is the contents of control register A, control register B, the color select register and the video look-up table. These relationships are demonstrated later in the mode_table definition.

The array *vlts* defines two sets of video look-up table initialization values. The values used are those listed in Table 7-4 and Table 7-5. Notice that each state supports more than one video mode. These relationships are demonstrated later in the mode_table definition.

The array *mode_list* is not required to program the video modes, however, the example uses this array to index through the various modes as it performs the demonstration.

NOTE
The two video modes, 0xfe and 0xff, are not defined or supported by the ROM BIOS. The mode numbers, 0xfe and 0xff, are defined only within the limits of this example.

```
/*************************************************************************/
/*       Define table values and declare globals used in examples        */
/*************************************************************************/

unsigned char crtc[6][16] =       /* Refer to Table 7-7 & 7-8 */
{
{ 0x34, 0x28, 0x2d, 0x54, 0x1a, 0x08, 0x19, 0x19,                /* TEXT */
  0x40, 0x0f, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },           /* 40 x 25 */

{ 0x69, 0x50, 0x58, 0x58, 0x1a, 0x08, 0x19, 0x19,                /* TEXT */
  0x40, 0x0f, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },           /* 80 x 25 */

{ 0x34, 0x28, 0x2c, 0x54, 0x6d, 0x00, 0x64, 0x66,             /* GRAPHICS */
  0x40, 0x03, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },      /* 320 x 200 x 4 */
                                                         /* 640 x 200 x 2 */
                                                         /* 640 x 400 x 2 */

{ 0x69, 0x50, 0x58, 0x58, 0x6d, 0x00, 0x64, 0x66,             /* GRAPHICS */
  0x42, 0x03, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },      /* 640 x 400 x 4 */
                                                         /* 640 x 200 x 4 */

{ 0x83, 0x64, 0x6d, 0x5a, 0x6d, 0x01, 0x3f, 0x53,             /* GRAPHICS */
  0x40, 0x03, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },      /* 800 x 250 x 4 */

{ 0x69, 0x50, 0x58, 0x58, 0x36, 0x00, 0x32, 0x33,             /* GRAPHICS */
  0x40, 0x07, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00 },    /* 320 x 200 x 16 */
};

unsigned char vlts[2][16] =
{
  { 0x00, 0x01, 0x02, 0x03,                              /* See Table 7-4 */
    0x04, 0x05, 0x06, 0x0e,
    0x08, 0x09, 0x0a, 0x0b,
    0x0c, 0x0d, 0x07, 0x0f },

  { 0x00, 0x04, 0x08, 0x07,                              /8 See Table 7-5 */
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00 },
};

int mode_list[12] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
                      0x06, 0xd0, 0xd1, 0xd2, 0xfe, 0xff };
```

The array *mode_table* is an array of structures of type M_TABLE. Each structure contains data or pointers to data that are required to program a particular video mode. Refer back to the declaration of the structure type M_TABLE to determine the relative placement or meaning of each value. The base address, number of scan pages, color bits per pixel, and width are determined from Figure 7-3 through Figure 7-16.

The array message is not required to program the video modes, however, the example program uses a string from the array message to identify and confirm the current video mode. The appropriate string is determined by the function *get_mess_p*.

The array *rltr_e* defines the character font for a reverse (mirror image) letter 'E'. It is used to demonstrate writing the font RAM and the effect it has. The character cell size is 8 x 16.

The array *c_font* reserves enough space to store the font for an entire character set (256 characters having a cell size of 8 x 16). The example program copies the current contents of the font RAM to this space.

The variable font_h allows the program to dynamically change, between demonstrations, the height of the character font. The variable font_w is provided for consistency.

The variable vid_mode allows the currently selected mode to be known globally.

```
M_TABLE  mode_table[13] =
{
{ &crtc[0][0], &vlts[0], 0x08, 0x68, 0x00, VB8, 8, 0x0400, 0x04,  40,  25 },
{ &crtc[0][0], &vlts[0], 0x08, 0x68, 0x00, VB8, 8, 0x0400, 0x04,  40,  25 },
{ &crtc[1][0], &vlts[0], 0x09, 0x68, 0x00, VB8, 4, 0x0800, 0x04,  80,  25 },
{ &crtc[1][0], &vlts[0], 0x09, 0x68, 0x00, VB8, 4, 0x0800, 0x04,  80,  25 },
{ &crtc[2][0], &vlts[0], 0x0a, 0x68, 0x00, VB8, 2, 0x2000, 0x02,  80, 200 },
{ &crtc[2][0], &vlts[0], 0x0a, 0x68, 0x00, VB8, 2, 0x2000, 0x02,  80, 200 },
{ &crtc[2][0], &vlts[0], 0x1a, 0x68, 0x07, VB8, 2, 0x2000, 0x01,  80, 200 },
{ &crtc[2][0], &vlts[0], 0x18, 0x68, 0x07, VB8, 4, 0x2000, 0x01,  80, 400 },
{ &crtc[3][0], &vlts[1], 0x1b, 0x68, 0x00, VB0, 4, 0x4000, 0x02, 160, 400 },
{ &crtc[4][0], &vlts[1], 0x1b, 0xe8, 0x00, VB0, 4, 0x4000, 0x02, 200, 250 },
{ &crtc[5][0], &vlts[0], 0x0b, 0x68, 0x00, VB8, 4, 0x2000, 0x04, 160, 200 },
{ &crtc[3][0], &vlts[1], 0x19, 0x68, 0x00, VB8, 2, 0x4000, 0x02, 160, 200 },
};

char message[12][24] =
{
  "  40 x  25  monochrome",
  "  40 x  25  color",
  "  80 x  25  monochrome",
  "  80 x  25  color",
  " 320 x 200 x  4-color",
  " 320 x 200  monochrome",
  " 640 x 200 x  2-color",
  " 640 x 400 x  2-color",
  " 640 x 400 x  4-color",
  " 800 x 250 x  4-color",
  " 320 x 200 x 16-color",
  " 640 x 200 x  4-color",
};

char press [32] = "Press any function key to exit";

char rltr_e[16] = { 0x00, 0x00, 0xfe, 0x02, 0x02, 0x02, 0x7e, 0x02,
                    0x02, 0x02, 0x02, 0xfe, 0x00, 0x00, 0x00, 0x00 };

char c_font[256][16];          /* space to store current character set */

char c_cursor[8] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff } ;

int font_w = 8;                                  /* font width in pixels */
int font_h = 16;                                /* font height in pixels */
int vid_mode = 2;                                  /* current video mode */
```

The function *get_mode_p* provides a single source for a pointer to video mode data.

> **NOTE**
> The two video modes, 0xfe and 0xff, are not defined or supported by the ROM BIOS. The mode numbers, 0xfe and 0xff, are defined only within the limits of this example.

The function *get_mess_p* provides a single source for a pointer to a string that describes the currently selected video mode. This function is not required to program the video modes, however, it is used to support the example program.

```
/*************************************************************************/
/*      get_mode_p() - returns a pointer to a mode table                 */
/*************************************************************************/
M_TABLE *get_mode_p(d_mode)                    /* get mode table pointer */

int d_mode;                                                /* desired mode */
{
  switch(d_mode)                              /* discover desired mode */
  {
    case    0: return(&mode_table[0]);        /*  40 x  25  monochrome */
    case    1: return(&mode_table[1]);             /*  40 x  25  color */
    case    2: return(&mode_table[2]);        /*  80 x  25  monochrome */
    case    3: return(&mode_table[3]);             /*  80 x  25  color */
    case    4: return(&mode_table[4]);         /* 320 x 200 x  4 color */
    case    5: return(&mode_table[5]);        /* 320 x 200  monochrome */
    case    6: return(&mode_table[6]);         /* 640 x 200 x  2 color */
    case 0xd0: return(&mode_table[7]);         /* 640 x 400 x  2-color */
    case 0xd1: return(&mode_table[8]);         /* 640 x 400 x  4-color */
    case 0xd2: return(&mode_table[9]);         /* 800 x 250 x  4-color */
    case 0xfe: return(&mode_table[10]);        /* 320 x 200 x 16-color */
    case 0xff: return(&mode_table[11]);        /* 640 x 200 x  4-color */
  }
}

/*************************************************************************/
/* get_mess_p() returns a pointer to a string that describes the mode  */
/*************************************************************************/
char *get_mess_p()                              /* get message pointer */
{
  switch(vid_mode)                            /* discover current mode */
  {
    case    0: return(&message[0][0]);        /*  40 x  25  monochrome */
    case    1: return(&message[1][0]);             /*  40 x  25  color */
    case    2: return(&message[2][0]);        /*  80 x  25  monochrome */
    case    3: return(&message[3][0]);             /*  80 x  25  color */
    case    4: return(&message[4][0]);         /* 320 x 200 x  4 color */
    case    5: return(&message[5][0]);        /* 320 x 200  monochrome */
    case    6: return(&message[6][0]);         /* 640 x 200 x  2 color */
    case 0xd0: return(&message[7][0]);         /* 640 x 400 x  2-color */
    case 0xd1: return(&message[8][0]);         /* 640 x 400 x  4-color */
    case 0xd2: return(&message[9][0]);         /* 800 x 250 x  4-color */
    case 0xfe: return(&message[10][0]);        /* 320 x 200 x 16-color */
    case 0xff: return(&message[11][0]);        /* 640 x 200 x  4-color */
  }
}
```

The function *w_vlt* writes the video look-up table.

The parameter pva is a pointer to a packed array of byte values.

Notice that the routine waits until the start of video blanking time to perform the operation and that video output is disabled on return.

```
/*********************************************************************/
/*  w_vlt() - copies a set of vlt-values to the video look-up table    */
/*********************************************************************/

void w_vlt(pva)                                          /* write vlt */

register char   *pva;                    /* ptr to array of characters */

{

register int     i;                                       /* loop counter */
VLT far *pvlt;                                 /* pointer to access vlt */

  pvlt = (VLT far *)VB8;                     /* initialize pointer to vlt */
  while(inp(STAT_REGB) & 0x80)             /* wait until display is active */
    ;
 while(inp(STAT_REGB) & 0x80 == 0)            /* wait until beginning of */
    ;                                                  /* display blanked */
  outp(CTRL_REGB, 0x04);                             /* enable vlt access */
  for(i = 0; i < 16; i++)                          /* do all 16 vlt values */
                                        (pvlt++)->
vlt_byte = *pva++;                             /* write to vlt */
  outp(CTRL_REGB, 0);                                /* return to normal */
}
```

The function *r_w_font* reads or writes the font RAM. If the parameter dir is false, it reads from the font RAM. Otherwise, it writes to the font RAM.

The parameter pfa is a pointer to a packed array of byte values.

Notice that to access the font RAM, the current mode must be one of the text modes. The mode is changed temporarily and then restored to the mode indicated by vid_mode.

Also notice that the routine waits until video blanking time to perform the operation and that video output is disabled on return.

```
/*************************************************************************/
/* r_w_font() copies the indicated number of character fonts to or       */
/*            from the font ram starting at the indicated character      */
/*************************************************************************/

void r_w_font(pfa, dir, c_value, count)      /* read or write font ram */

register char   *pfa;                          /* pointer to font array */
int     dir;                                /* direction to move font data */
unsigned char c_value;                        /* start at this char value */
register int count;                          /* number of character fonts */

{

int     i;                                                 /* loop counter */
FONT far *pfnt;                             /* pointer to access font ram */

  mode_init(2);                                        /* text mode required */
  outp(CTRL_REGB, 0x10);                             /* enable font ram access */
  count <<= 4;                          /* 16 bytes of data per char pattern */
  pfnt = (FONT far *)VB8;                 /* initialize pointer to font ram */
  pfnt += (unsigned int)c_value << 4;      /* offset to start of pattern */
  if(dir)                                 /* nonzero means write font ram */
    while(count--)                                   /* do requested count */
      (pfnt++)->font_byte = *pfa++;                   /* write to font ram */
  else                                      /* zero means read font ram */
    while(count--)                                   /* do requested count */
      *pfa++ = (pfnt++)->font_byte;                       /* read font ram */
  outp(CTRL_REGB, 0x00);                           /* disable font ram access */
  mode_init(vid_mode);                          /* restore current video mode */
}
```

The function *mode_init* places the video processor in a predefined mode state as indicated by the parameter d_mode. The video look-up table is initialized first because w_vlt() waits for blanking time to start its operation and the display is disabled when it returns.

Because the CRT controller is in an unstable state during initialization, the video output must be disabled.

The CRT controller has 18 internal registers, R0 through R17. The last two, R16 and R17, are read only. Thus, they are ignored during initialization. The crt controller has two external registers, the index register and the data register. To access one of the 18 internal registers, write the desired register number to the index register and read or write the data register.

The control register A and color select register are initialized to ensure that the contents are appropriate for the mode.

Initializing control register B would enable the display. It was not done at this time to prevent flashing the display if other operations have to be performed. Other operations might include changing the font RAM, changing the video look-up table from the default or preparing the video memory.

The function *mv_cursor* positions the cursor at the desired row and column location.

```
/*************************************************************************/
/*  mode_init() initializes the crtc and mode registers by moving a     */
/*               table of values to the appropriate registers           */
/*************************************************************************/
mode_init(d_mode)                          /* initialize to desired mode */

int     d_mode;                                    /* desired video mode */
{
register int    i;                                        /* loop control */
register char   *pc;                             /* pointer to crtc_table */
M_TABLE  *pmt;                                   /* pointer to mode_table */
unsigned int intr_flag;                                   /* CPU IF state */

  pmt = get_mode_p(d_mode);          /* get pointer to video mode table */
  intr_flag = int_off();                        /* no interrupts please */
  w_vlt(pmt->vt, TRUE);                               /* write vlt data */
  pc = pmt->ct;                        /* assign pointer to crtc_table */
  for(i = 0; i < 16; i++)               /* do registers RO through R15 */
  {
    outp(CRTC_INDEX, i);                  /* indicate desired register */
    outp(CRTC_DATA, *pc++);                 /* write appropriate value */
  }
  outp(CTRL_REGA, pmt->cra);                /* set control register A */
  outp(COLR_SELC, pmt->csr);              /* set color select register */
  int_on(intr_flag);                              /* allow interrupts */
}

/*************************************************************************/
/*  mv_cursor() moves the cursor to the desired location                 */
/*************************************************************************/
mv_cursor(row, col)

int row;                                                 /* desired row */
int col;                                              /* desired column */
{
int i;
register M_TABLE  *pmt;                  /* pointer to video mode table */
unsigned int intr_flag;                                   /* CPU IF state */

  intr_flag = int_off();                          /* no interrupts please */
  pmt = get_mode_p(vid_mode);          /* get pointer to video mode table */
  i = (pmt->width * row) + col;
  if(vid_mode == 0 || vid_mode == 1 || vid_mode == 2 || vid_mode == 3)
  {
    outp(CRTC_INDEX, 14);                   /* indicate desired register */
    outp(CRTC_DATA, i >> 8);                  /* write appropriate value */
    outp(CRTC_INDEX, 15);                   /* indicate desired register */
```

```
    outp(CRTC_DATA, i & 0xff);                    /* write appropriate value */
  }
  int_on(intr_flag);                                    /* allow interrupts */
}
```

The function *cursor_on* positions the cursor and turns the cursor on, so that it is visible.

The function *cursor_off* turns the cursor off, so that it is invisible.

```
/*************************************************************************/
/*  cursor_on() turns the cursor on                                      */
/*************************************************************************/

cursor_on(row, col)

int row;                                        /* desired row position */
int col;                                     /* desired column position */

{
unsigned int intr_flag;                                  /* CPU IF state */

  if(vid_mode == 0 || vid_mode == 1 || vid_mode == 2 || vid_mode == 3)
  {
    intr_flag = int_off();                        /* no interrupts please */
    mv_cursor(row, col);
    outp(CRTC_INDEX, 10);                     /* indicate desired register */
    outp(CRTC_DATA, 0);                         /* write appropriate value */
    outp(CRTC_INDEX, 11);                     /* indicate desired register */
    outp(CRTC_DATA, font_h - 1);                /* write appropriate value */
    int_on(intr_flag);                                /* allow interrupts */
  }
}

/*************************************************************************/
/*  cursor_off() turns the cursor off                                    */
/*************************************************************************/

cursor_off()
{
unsigned int intr_flag;                                  /* CPU IF state */

  if(vid_mode == 0 || vid_mode == 1 || vid_mode == 2 || vid_mode == 3)
  {
    intr_flag = int_off();                        /* no interrupts please */
    outp(CRTC_INDEX, 10);                     /* indicate desired register */
    outp(CRTC_DATA, 17);                        /* write appropriate value */
    int_on(intr_flag);                                /* allow interrupts */
  }
}
```

The function *set_mode* establishes a new video mode. It does this by initializing the mode, clearing the screen (video memory) to an empty (blank) state, enabling the display and advertising the new mode in vid_mode.

Notice that the video memory is cleared after the mode is initialized. This is because each mode enables only certain sections of video memory.

The function *screen_on* enables or disables the video output as indicated by the parameter flag. This is done through control register B.

```
/*************************************************************************/
/*  set_mode()  sets the mode as indicated, clears the screen and        */
/*              sets the current video mode flag                         */
/*************************************************************************/
set_mode(d_mode)                              /* set desired video mode */

int     d_mode;                                          /* desired mode */
{
M_TABLE  *pmt;                                /* pointer to mode_table */

  vid_mode = d_mode;                        /* tell world what new mode is */
  pmt = get_mode_p(d_mode);               /* get pointer to video mode table */
  mode_init(d_mode);                    /* disable display & initialize mode */
  clear_vid_mem();                                    /* clear screen */
  screen_on(TRUE);                                 /* enable the display */
}

/*************************************************************************/
/* screen_on() enables or disables the display (blanking / unblanking) */
/*************************************************************************/
screen_on(flag)                             /* disable or enable display */

int     flag;                                             /* what to do */
{

register M_TABLE  *pmt;                     /* pointer to video mode table */

  if(flag)                                /* nonzero means enable display */
  {
    pmt = get_mode_p(vid_mode);         /* get pointer to video mode table */
    outp(CTRL_REGB, pmt->crb);           /* control reg B enables display */
  }
  else outp(CTRL_REGB, 0);                    /* all bits off will disable */
}
```

The function *clear_vid_mem* initializes video memory to a value appropriate for the current mode. That is, in text modes the character byte is set to a space character and the attribute byte is set to medium intensity white foreground and a black background. For graphics modes, the color bits are set to zero (black). Only the addressable video memory is initialized.

Notice that the pointer to video memory is declared as a pointer to an integer. The video memory data bus is a full 16-bit bus. Because text mode utilizes a character and attribute byte pair and graphics mode values are all zero, it is appropriate to take advantage of the 16-bit data bus. Also, the normal storage for an integer is low byte first and high byte second.

The scan page size is specified in bytes. To calculate the correct memory size, the number of scan pages is multiplied by half of the scan page size.

```
/*************************************************************************/
/*  clear_vid_mem() based on the current mode, video memory is cleared */
/*                  to spaces or NULLs. The size of memory to clear    */
/*                  is calculated from the mode table.                 */
/*************************************************************************/

clear_vid_mem()
{

register unsigned int size;                          /* loop control */
int far *pvm;                                  /* pointer to video memory */
M_TABLE *pmt;                                /* pointer to video mode table */

  pmt = get_mode_p(vid_mode);                   /* ptr to current mode data */
  pvm = (int far *)pmt->base;                  /* ptr to start of video mem */
  size = pmt->nsp * (pmt->sps >> 1);          /* number of integers to init */
  switch(vid_mode)                              /* text or graphics mode ? */
  {
    case 0:                                   /* text modes initialized to */
    case 1:                                      /* a space character with */
    case 2:                                            /* medium intensity */
    case 3:
      while(size--) *pvm++ = 0x0720;             /* write char & attribute */
      break;

    default:                                   /* for graphics modes, just */
      while(size--) *pvm++ = 0x0000;                   /* set all bits off */
      break;
  }
}
```

The function *do_border* displays the indicated character in a pattern or border (like a picture frame) at the extremes of the screen. It first decides whether the mode is a text mode or a graphics mode. For text mode, it retrieves the row and column counts from the mode table. Graphics mode programming is slightly more difficult. The row count is calculated by dividing the total scan lines by the fonts scan line height. The column count is calculated by dividing scan line width (measured in bytes) by the the number of color bits per pixel (only character fonts 8 pixels wide are supported in the example).

It then executes a for loop to generate the pattern. After the pattern is displayed, a message is displayed describing the current video mode. Notice that in graphics mode, the font height is temporarily changed to display the message. This is because the message is displayed using an 8 x 16 character font and it must be accommodated when displaying the border in an 8 x 8 character font.

```
/*************************************************************************/
/*  do_border()     From the mode table, the maximum number of           */
/*                  displayable lines is calculated and a border         */
/*                  is drawn using the indicated character.              */
/*************************************************************************/

do_border(pc)

char    *pc;

{

unsigned char attr;
char    *pm;
int     row, rows;
int     col, cols;
int     t_font_h;
M_TABLE *pmt;

  pmt = get_mode_p(vid_mode);              /* get pointer to mode table */
  pm = get_mess_p(vid_mode);                 /* get pointer to message */
  switch(vid_mode)                        /* discover the current mode */
  {
    case 0:                                            /* text mode 0 or */
    case 1:                                            /* text mode 1 or */
    case 2:                                            /* text mode 2 or */
    case 3:                                            /* text mode 3 ?  */
      attr = 0x07;     /* black background & medium intensity foregrnd */
      rows = pmt->length;                 /* length specified in table */
      cols = pmt->width;                  /*  width specified in table */
```

```
        for(row = 0; row < rows; row++)
        {
          disp_t(row, 0, *pc, attr);                    /* write left side */
          if(row == 0 || row == rows - 1)              /* first or last row */
            for(col = 1; col < cols - 1; col++)
              disp_t(row, col, *pc, attr);
          disp_t(row, cols - 1, *pc, attr);            /* write right side */
        };
          col = (cols - strlen(pm)) >> 1;                /* center message */
          while(*pm)
              disp_t(rows >> 1, col++, *pm++, attr);        /* do message */
          pm = press;
          col = (cols - strlen(pm)) >> 1;                /* center message */
          while(*pm)
              disp_t((rows >> 1) + 1, col++, *pm++, attr); /* do message */
          break;

      default:
          attr = 0xff >> (8 - pmt->cb);                 /* medium intensity */
          rows = pmt->length / font_h;                 /* rows for this font */
          cols = pmt->width / pmt->cb;                  /* columns this mode */
          for(row = 0; row < rows; row++)                     /* do all rows */
          {
              disp_t(row, 0, *pc, attr);                /* write left side */
              if(row == 0 || row == rows - 1)
                  for(col = 1; col < cols - 1; col++)
                      disp_t(row, col, *pc, attr);
              disp_t(row, cols - 1, *pc, attr);        /* write right side */
          };
          t_font_h = font_h;                            /* save current font */
          font_h = 16;                                /* message font is 8 x 16 */
          col = (cols - strlen(pm)) >> 1;                /* center message */
          while(*pm)
             disp_t(pmt->length >> 5, col++, *pm++, attr); /* do message */
          pm = press;
          col = (cols - strlen(pm)) >> 1;                /* center message */
          while(*pm)
              disp_t((rows >> 1) + 1, col++, *pm++, attr); /* do message */
          font_h = t_font_h;                          /* restore current font */
          break;
    }
}
```

The function *disp_g* displays a character at the desired row and column location. The color bit pixels that form the character are set to value in attr. The routine assumes that it is provided an attribute that is consistent with the number of color bits per pixel.

The routine calculates the position as a constant offset to the first byte of the first scan line. After the scan line has been displayed, the scan line offset is incremented to the next scan page. After writing a scan line to each scan page, the scan line offset is zeroed and the constant offset is incremented by the length of one scan line. The actual position is the constant offset plus the scan line offset.

The middle for loop ensures that all eight pixels of the scan line are displayed. For example, a 16-color display requires four color bits per pixel or four bytes per character scan line.

For each pixel, the interior for loop shifts the color bit image and if the pixel bit is set, the color bit attribute is *OR*ed into the color bit image. The number of pixel representations per byte is determined by dividing the font width by the number of color bits per pixel. Only eight pixel wide fonts are supported by the example.

```
/*************************************************************************/
/*  disp_g()    Draws a character in graphics mode at the calculated     */
/*              row and column position.                                  */
/*************************************************************************/
disp_g(row, col, pc, attr)

int row, col;
register unsigned char *pc;
unsigned char attr;
{
unsigned char far *pvm;                          /* pointer to video memory */
unsigned char bi;                                          /* bit image */
unsigned char bd;                                  /* for bit testing  */
int bc;                                          /* bit count (current) */
int tbc;                                       /* terminating bit count */
int scan;                                          /* current scan line */
long c_off;                                        /* character offset */
unsigned int s_off;                                /* scan line offset */
register M_TABLE *pmt;                                       /* mode data */

  pmt = get_mode_p(vid_mode);                  /* get pointer to mode table */
  c_off = pmt->base + (col * pmt->cb) +         /* address of first byte  */
    (row * pmt->width * (font_h / pmt->nsp));
  tbc = font_w / pmt->cb;                   /* bit image bit count per byte */
  s_off = 0;                                           /* scan line offset */
  for(scan = 0; scan < font_h; scan++, pc++)          /* do all scan lines */
  {
    if(scan && scan % pmt->nsp == 0)             /* done all scan pages ? */
    {
      c_off += pmt->width;                      /* offset by one scan line */
      s_off = 0;                               /* reset to first scan page */
    }
    pvm = (char far *)(c_off + s_off);       /* ptr to first byte of scan */
    s_off += pmt->sps;          /* offset to next scan page, for next pass */
    for(bd = 0x80; bd;)                    /* start at left most bit/pixel */
    {
      bi = 0x00;                                          /* null bit image */
      for(bc = 0; bc < tbc; bc++, bd >>= 1)           /* do a byte of scan */
      {
        bi <<= pmt->cb;              /* shift bit image by # of color bits */
        if(*pc & bd) bi |= attr;                   /* or in next bit image */
      }
      *pvm++ = bi;                             /* write a byte of scan line */
    }
  }
}
```

The function *disp_t* displays a character and attribute at the desired row and column location.

The example uses zero-based row and column values. When calculating the position, this saves subtracting one from the row and column values.

Notice that the pointer to video memory is declared as a pointer to an integer. The video memory data bus is a full 16-bit bus. Because text mode utilizes a character and attribute byte pair, it is appropriate to take advantage of the 16-bit data bus. Also, the normal storage for an integer is low byte first and high byte second.

```
/*************************************************************************/
/*  disp_t()    writes a character and its attribute to the indicated */
/*              row and column position                                */
/*************************************************************************/


disp_t(row, col, c, attr)

int     row;
int     col;
unsigned char c;
unsigned char attr;

{
int far *pvm;                                /* pointer to video memory */
register M_TABLE *pmt;

  if(vid_mode == 0 | vid_mode == 1 | vid_mode == 2 | vid_mode == 3)
  {
    pmt = get_mode_p(vid_mode);             /* get pointer to mode table */
                                             /* get address of character */
    pvm = (int far *)(pmt->base +
          (row * (pmt->width << 1) + (col << 1)));
    *pvm = ((int)attr << 8) | (int)c;          /* character & attribute */
  }
  else disp_g(row, col, &c_font[c][0], attr);
}
```

The function *video* sets up various conditions and executes the example program. The major points are:

1. Read the current font and save it.
2. Select a video mode.
3. Change the pattern of the letter 'E' to a mirror image of the letter 'E.'
4. Restore the pattern of the letter 'E' from the saved font.
5. Display the letter 'E' border pattern in the selected video mode.
6. Restore the video mode to mode 3 and exit.

```
/*************************************************************************/
/* video() - execute video examples                                      */
/*************************************************************************/

video()
{

static MESSAGE mvid[] =                                    /* video menu */
{
  {  3, 33, "Video Example" },
  {  5, 24, "F1.  Select video mode" },
  {  6, 24, "F2.  Invert letter E" },
  {  7, 24, "F3.  Restore letter E" },
  {  8, 24, "F4.  Display selected video mode" },
  {  9, 24, "F10. Return to Main menu" },
  {  0,  0,  0 },
};

static MESSAGE mvmd[] =                                    /* video menu */
{
  {  3, 31, "Select Video Mode" },
  {  5, 24, "F1.  40 x 25 Text" },
  {  6, 24, "F2.  80 x 25 Text" },
  {  7, 24, "F3.  320 x 200 x 4-color" },
  {  8, 24, "F4.  320 x 200 x 16-color" },
  {  9, 24, "F5.  640 x 200 x 2-color" },
  { 10, 24, "F6.  640 x 200 x 4-color" },
  { 11, 24, "F7.  640 x 400 x 2-color" },
  { 12, 24, "F8.  640 x 400 x 4-color" },
  { 13, 24, "F9.  800 x 252 x 4-color" },
  { 14, 24, "F10. Return to video example" },
  {  0,  0,  0 },
};

char line[512];                                  /* to hold input line */
```

```
int i;                                          /* to hold menu selection */
int mode;                                         /* temp value for mode */
unsigned char c;

  r_w_font(&c_font[0][0], FALSE, 0x00, 256);       /* read all font ram */
  set_mode(3);                                      /* reset video mode */
  disp_menu(mvid);                                /* display the video menu */
  line[0] = 0;                                        /* null terminated */
  while(1)                                          /* forever (see F10) */
  {
    switch(line[0])                             /* determine menu selection */
    {
      case F1:                                          /* select video mode */
        disp_menu(mvmd);                          /* display the mode menu */
        line[0] = get_fkey();                /* get a function key selection */
        switch(line[0])
        {
          case F1: mode = 1;    break;
          case F2: mode = 3;    break;
          case F3: mode = 4;    break;
          case F4: mode = 0xfe; break;
          case F5: mode = 6;    break;
          case F6: mode = 0xff; break;
          case F7: mode = 0xd0; break;
          case F8: mode = 0xd1; break;
          case F9: mode = 0xd2; break;
          case F10: break;
        }
        break;

      case F2:                                    /* reverse 'E' to font ram */
        r_w_font(&rltr_e[0], TRUE, 'E', 1);
        set_mode(3);                                      /* reset video mode */
        break;

      case F3:                                    /* restore 'E' to font ram */
        r_w_font(&c_font['E'][0], TRUE, 'E', 1);
        set_mode(3);                                      /* reset video mode */
        break;

      case F4:                                     /* display selected mode */
        set_mode(mode);                             /* set the desired mode */
        do_border("E");              /* write letter 'E' at screen extremes */
        while(1) if(get_key(&c) >= 0 && c == 0) break;
        while(1) if(get_key(&c) >= 0) break;
        set_mode(3);                               /* set the desired mode */
        break;
```

```
        case F10:                          /* return to caller (main menu) */
          return;
      }
      disp_menu(mvid);                              /* display the rtc menu */
      line[0] = get_fkey();       /* get a function key for menu selection */
    }
}
```

Chapter 8
Keyboard-Interface Controller and Keyboard

Introduction

The keyboard-interface controller is an Intel 8042 microcomputer with internal firmware developed by DIGITAL. The keyboard-interface controller provides a physical and logical interface between the LK250 keyboard and the VAXmate CPU. Also, it provides several hardware control functions that are unrelated to the LK250 keyboard.

The LK250 keyboard is an intelligent keyboard with 105 keys. The keys are divided into functional groups as follows:

- 57 key main section
- 20 key function-key section
- 18 key keypad section
- 10 key auxiliary and direction key section

Keyboard-Interface Controller

Physical Interface to the CPU

The VAXmate CPU communicates with the keyboard-interface controller through two 8-bit parallel ports in input/output (I/O) address space 0060H and 0064H. The registers accessed through these ports are:

- Reading port 0064H accesses the status register.
- Writing port 0064H accesses the command register.
- Reading port 0060H accesses the output data register.
- Writing port 0060H accesses the input data register.

The contents and usage of these registers are described later in this chapter.

Physical Interface to the Keyboard

The keyboard-interface controller communicates with the LK250 keyboard over a bi-directional serial data line. A second line provides serial clock rate. A combination of signals and timing on the serial clock line provides the communications protocol. These lines are in the coiled cable that connects the LK250 keyboard and the VAXmate workstation.

Logical Interface

The keyboard-interface controller has two modes of operation, pass-through or translate. In pass-through mode, all data received from the LK250 keyboard are stored in the output buffer without modification. In translate mode, all data in the range 00H-99H and F0H are translated to an industry-standard or a DIGITAL extended scan code and stored in the output buffer. Bit 6 of the command byte controls this mode of operation. The keyboard-interface controllers default mode is pass through.

> **NOTE**
> During the system powerup initialization, the ROM BIOS sets the keyboard-interface controller to translate mode. Therefore, the operating system sees the keyboard-interface controller in translate mode.

Control Functions

The keyboard-interface controller performs some control and status functions that are unrelated to LK250 keyboard operation. Therefore, the keyboard-interface controller has two 8-bit ports, port 1 and port 2, that are in the keyboard-interface controller I/O address space. Commands are available that instruct the keyboard-interface controller to read port 1 and read or write port 2. Those commands are discussed in the command register section. Table 8-1 defines the port 1 bits and Table 8-2 defines the port 2 bits.

Table 8-1 Port 1 Bit Definitions

Bit	Description
7	Undefined
6	Always 0
5-3	Undefined
2	EXPANSION BOX INSTALLED 0 = Expansion box installed 1 = Expansion box not installed
1	RAM OPTION INSTALLED 0 = RAM option installed 1 = RAM option not installed
0	RAM OPTION ERROR 0 = RAM option parity error 1 = No error

Table 8-2 Port 2 Bit Definitions

Bit	Description
7	Keyboard data (output)
6	Keyboard clock inverted (output)
5	Undefined
4	CPU IRQ1 0 = Keyboard-interface controller not interrupting 1 = Keyboard-interface controller interrupt to CPU
3	Undefined
2	Undefined
1	Gate system address line 20 0 = System address line 20 enabled. Use this state for 80286 virtual protected mode. 1 = System address line 20 disabled. Use this state for 80286 real mode.
0	Reset 80286 CPU (affects only the CPU) 0 = 80286 CPU in reset state 1 = 80286 CPU not in reset state

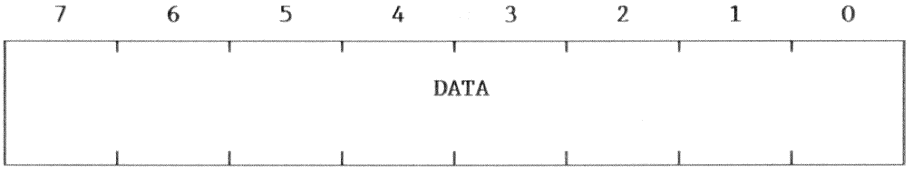
Keyboard-Interface Controller Diagnostics

On powerup, the keyboard-interface controller executes a diagnostic test. The test verifies the keyboard-interface controllers ROM and RAM. The keyboard-interface controller completes the test within 200 ms after powerup. During the powerup test, the LK250 keyboard is ignored.

Failing this test is considered a fatal system error. If an error occurs during powerup testing, the keyboard-interface controller will not respond to any input, and must be reset. To reset the keyboard-interface controller, turn the VAXmate workstation off and then on.

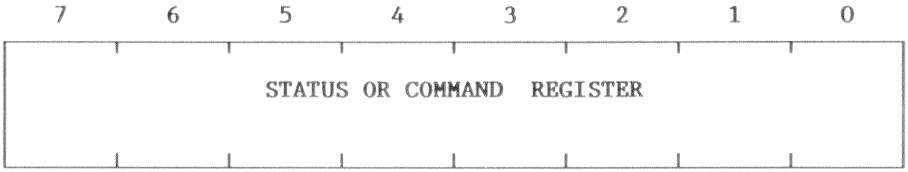
Keyboard-Interface Controller Registers

Data Register (0060H)



Bit	R/W	Description
7-0	R/W	The CPU reads or writes this port to exchange data with the keyboard-interface controller

Command Register (0064H)



Bit	R/W	Description
7-0	R	The CPU reads the keyboard-interface controller status register
	W	The CPU writes the keyboard-interface controller command register

Status Register (0064H)

7	6	5	4	3	2	1	0
PARITY ERROR	RECEIVE TIME-OUT	XMIT TIME-OUT	KEYBRD INHIBIT SWITCH	COMMAND /DATA	SYSTEM FLAG	INPUT BUFFER FULL	OUTPUT BUFFER FULL

Bit	R/W	Description
7	R	PARITY ERROR 0 = No parity error 1 = Parity error The keyboard to keyboard-interface controller serial communications use odd parity checking. When data from the keyboard contains a parity error, this bit is set to 1. Reading the status register clears this bit.
6	R	RECEIVE TIMEOUT 0 = No receive timeout error 1 = An in-progress transmission from the keyboard was not completed within 2 ms. Reading the status register clears this bit.
5	R	XMIT TIMEOUT - Transmit Timeout 0 = No transmit timeout error 1 = An in-progress transmission from the keyboard-interface controller to the keyboard was not completed within 2 ms. Reading the status register clears this bit. This bit is also used in combination with the parity error bit or the receive timeout bit to establish additional error conditions as follows: • If the transmission to the keyboard completes within 2 ms, but the response from the keyboard is not received within 20 ms, then the transmit and receive timeout bits are set to 1. • If the transmission to the keyboard and the response from the keyboard complete within the designated time frame, but the response contains a parity error, then the transmit timeout and parity error bits are set to 1.

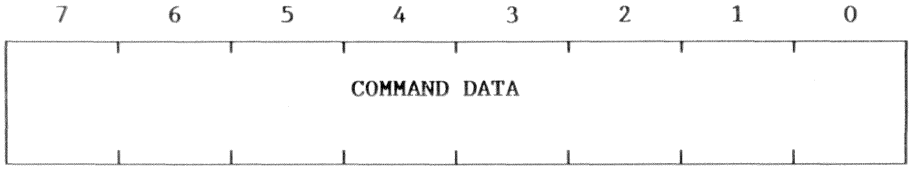
Bit	R/W	Description (Status Register - cont.)
4	R	KEYBRD INHIBIT SWITCH - Keyboard Inhibit Switch 0 = Keyboard is inhibited (locked) 1 = Keyboard is active (unlocked) The VAXmate workstation does not have a keyboard lock, so this bit is always set to 1.
3	R	COMMAND/DATA 0 = Of the pair (0060H/0064H), the last I/O write was to the data register, at I/O address 0060H. 1 = Of the pair (0060H/0064H), the last I/O write was to the command register, at I/O address 0064H. This bit is used internally by the keyboard-interface controller. It determines whether the byte in the keyboard-interface controllers input buffer is a command or data.
2	R	SYSTEM FLAG 0 = ROM BIOS should execute a normal powerup initialization process 1 = A virtual protected mode task request to reset the CPU to real mode The ROM BIOS interprets the state of the system flag to determine the reason the VAXmate CPU reset pin was toggled. If the system flag is set, the VAXmate CPU is returning to real mode from virtual protected mode. Otherwise, the ROM BIOS executes a normal powerup initialization process. During the keyboard-interface controllers powerup initialization sequence, the keyboard-interface controller clears the system flag to 0. Setting or clearing the system flag is a two step process: Issue a write-command-byte instruction by writing the value 0060H to the command register at I/O address 0064H. The write-command-byte instruction is discussed in detail later in this chapter. Write the command byte to the input data register at I/O address 0060H. The system flag reflects the value of bit 2 of the command byte. If bit 2 of the command byte is set to 1, so is the system flag. When bit 2 is 0, the system flag is cleared to 0.

Bit	R/W	Description (Status Register - cont.)
1	R	INPUT BUFFER FULL 0 = Keyboard-interface controller input data buffer is empty 1 = Keyboard-interface controller input data buffer contains data that has not been processed by the keyboard-interface controller. Data written to the input data register is transferred to the keyboard-interface controllers input data buffer. This bit reflects the status of the input data buffer.
0	R	OUTPUT BUFFER FULL 0 = Keyboard-interface controller output data buffer is empty 1 = Keyboard-interface controller output data buffer contains data that the CPU has not read

The status register is an 8-bit read-only register at I/O address 0064H. It contains information about the keyboard-interface controller and keyboard. Status changes do not cause an interrupt. The status register can be read at any time.

The keyboard-interface controller command E1H allows the CPU to initialize bits 7-4 of the status register. See the keyboard-interface controller command register description.

Command Register (0064H)



Bit	Description
7-0	Keyboard-interface controller commands from the CPU

The data written to this register are instructions to the keyboard-interface controller. Table 8-3 lists the keyboard-interface controller commands.

Table 8-3 Keyboard-Interface Controller Commands

Command Value	Description	Command Value	Description
00H-1FH	Reserved	C0H	Read Port 1
20H	Read command byte	C1H	Read Port 1
21H-5FH	Reserved	C2H-CFH	Reserved
60H	Write command byte	D0H	Read Port 2
61H-A9H	Reserved	D1H	Write Port 2
AAH	Self-test	D2H-DFH	Reserved
ABH	Interface test	E0H	Read test inputs
ACH	Reserved for expansion	E1H	Write status register
ADH	Disable keyboard	E2H-EFH	Reserved
AEH	Enable keyboard	F0H-FFH	Pulse output port
AFH-BFH	Reserved	——	——

Read Command Byte (20H)
Write Command Byte (60H)

The keyboard-interface controller has an internally stored command byte that determines how the interface controller responds to various conditions. The two commands, read command byte and write command byte, provide access to that command byte. When the keyboard-interface controller receives a read-command-byte command, it places the internally stored command byte in the output buffer. When the keyboard-interface controller receives a write-command-byte command, it stores the next data byte, written to the input data register at I/O address 0060H, to the internally stored command byte. Table 8-4 defines the command byte bits.

Table 8-4 Command Byte Bit Definitions

Bit	R/W	Definition
7	R/W	Always 0
6	R/W	Scan Code Type 0 = The LK250 scan codes are passed through without conversion (pass-through mode). 1 = The keyboard-interface controller converts the LK250 keyboard scan codes to industry-standard one byte values (translate mode).
5	R/W	Interface Type - Always 0
4	R/W	Disable Keyboard 0 = Keyboard enabled 1 = Keyboard disabled The keyboard-interface controller disables the keyboard by setting the clock line to a low state. This bit is cleared by writing this command byte or by writing LK250 keyboard data to the input data register at I/O address 0060H.
3	R/W	Inhibit Override 0 = Keyboard inhibit switch enabled (see status register bit 4). 1 = Keyboard inhibit switch disabled and the key lock function is overridden (see status register bit 4).

Table 8-4 **Command Byte Bit Definitions** (cont.)

Bit	R/W	Definition
2	R/W	System Flag 0 = ROM BIOS should execute a normal powerup initialization process 1 = A virtual protected mode task request to reset the CPU to real mode The ROM BIOS interprets the state of the system flag to determine the reason the VAXmate CPU reset pin was toggled. If the system flag is set, the VAXmate CPU is returning to real mode from virtual protected mode. Otherwise, the ROM BIOS executes a normal powerup initialization process. During the keyboard-interface controllers powerup initialization sequence, the keyboard-interface controller clears the system flag to zero. See the status register bit 2 description.
1	R/W	Always 0
0	R/W	Enable Output-Buffer-Full Interrupt 0 = Keyboard-interface controller does not generate interrupts to the CPU 1 = When the keyboard-interface controller places data in the output buffer, the keyboard-interface controller generates an interrupt to the VAXmate CPU.

Self-Test (AAH)

This command causes the keyboard-interface controller to perform internal diagnostic tests. The command byte is reset to 10H (keyboard disabled) and interrupts to the VAXmate CPU are disabled. The system flag (status register bit 2) is not changed. One of three possible diagnostic result codes is placed in the output buffer.

Diagnostic Code	Meaning
55H	No errors detected
FEH	Invalid ROM checksum
FDH	RAM test failed

Interface Test (ABH)

This command causes the keyboard-interface controller to test the state of the LK250 keyboard-interface lines. One of five possible diagnostic result codes is placed in the output buffer.

Diagnostic Code	Meaning
00H	No errors detected
01H	LK250 keyboard clock line always low
02H	LK250 keyboard clock line always high
03H	LK250 keyboard data line always low
04H	LK250 keyboard data line always high

Disable Keyboard (ADH)

This command sets bit 4 of the keyboard-interface controller command byte. The keyboard-interface controller disables the keyboard-interface by setting the clock line low.

Enable Keyboard (AEH)

This command clears bit 4 of the keyboard-interface controller command byte. The keyboard-interface controller enables the keyboard interface by freeing the clock line.

Read Port 1 (C0H)

The keyboard-interface controller reads port 1 (bits 7-0), sets bits 3-0 to 1 (for compatibility), and places the result in the output buffer. For port 1 bit definitions, see Table 8-1. Because it overwrites any data in the buffer, this command should not be used unless the output buffer is empty.

Read Port 1 (C1H)

The keyboard-interface controller reads port 1 (bits 7-0) and places bits 7-0 (without modification) in the output buffer. For port 1 bit definitions, see

Table 8-1. Because it overwrites any data in the buffer, this command should not be used unless the output buffer is empty.

Read Port 2 (D0H)

The keyboard-interface controller reads port 2 (bits 7-0) and places bits 7-0 in the output buffer. For port 2 bit definitions, see Table 8-2. Because it overwrites any data in the buffer, this command should not be used unless the output buffer is empty.

Write Port 2 (D1H)

The keyboard-interface controller takes the next byte of data written to the input data register at I/O address 0060H and writes it to the keyboard-interface controller output port 2. For port 2 bit definitions, see Table 8-2.

> CAUTION
> Bit 0 of keyboard-interface controller output port 2 is connected to the VAXmate CPU reset circuitry. Clearing bit 0 to a zero places the VAXmate CPU in a permanent reset state.

Read Test Inputs (E0H)

The keyboard-interface controller reads its T0 and T1 inputs and places the data in the output buffer. This command writes over any data in the output buffer that has not been read by the CPU. Bit 0 corresponds to the state of T0 and bit 1 corresponds to the state of T1.

Write Status Register (E1H)

This is a diagnostic command. The keyboard-interface controller takes bits 7-4 of the next byte written to the input data register (60H) and writes them to bits 7-4 of the status register. Only bits 7-4 are affected.

After the next keyboard-interface controller command that makes data available to the VAXmate CPU, the keyboard-interface controller updates the status register to reflect the current status.

Pulse Output Port (F0H-FFH)

The keyboard-interface controller pulses bits 3-0 of port 2 according to the value of bits 3-0 of the command value. Any of the bits 3-0 that are clear in the command byte are pulsed low for approximately six microseconds.

All four bits (3-0) are affected by this command, but bits 3-2 are not connected to anything. Bit 1 gates the address line A20. Bit 0 is connected to the VAXmate CPU reset circuitry. The ROM BIOS uses this command to return to real mode from virtual protected mode.

Keyboard-Interface Controller Error Handling

If the LK250 keyboard initiates a transmission sequence and the data is received with a parity error, the keyboard-interface controller issues a resend command. If the response to the resend command is bad, the keyboard-interface controller places an FFH code in the output buffer and sets the parity error bit in the status register.

If the keyboard-interface controller cannot start a transmission within 15 milliseconds or it cannot complete a transmission within two milliseconds after it was started, the transmit timeout bit is set in the status register and an FEH (keyboard resend request) is placed in the output buffer.

If the LK250 keyboard does not respond within 20 milliseconds, then the transmit timeout bit and receive timeout bit are set in the status register. If the LK250 keyboard response was received with a parity error, then the transmit timeout bit and the parity error bit are set in the status register and an FEH (keyboard resend request) is placed in the output buffer.

If the keyboard-interface controller has not inhibited the keyboard and a LK250 keyboard transmission does not complete within two milliseconds after it is started, the receive timeout bit is set in the status register. A resend command is not issued by the keyboard-interface controller.

LK250 Keyboard

Scan Codes

When a key is pressed, the LK250 keyboard transmits a value, in the range 01H-99H, that identifies the pressed key. This value is known as a *scan code.* If the key is held down for a time that exceeds the autorepeat delay time, the keyboard repeatedly transmits the scan code at the autorepeat rate. When a key is released, the LK250 keyboard transmits a release code of F0H followed by the scan code of the released key.

The LK250 keyboard transmits scan codes to the keyboard-interface controller. If the keyboard-interface controller is in pass-through mode, the keyboard-interface controller transmits the scan code or release code (F0H) and scan code without conversion. If the keyboard-interface controller is in translate mode, the keyboard-interface controller converts the LK250 scan code or release code (F0H) and scan code to an industry-standard 1-byte value. In the industry-standard representation, a released key is indicated by adding 80H to the translated scan code value.

Table 8-5 lists the LK250 scan codes, their equivalent keyboard-interface-controller translated industry-standard value, and keyboard location label. Figure 8-1, a representation of the LK250 keyboard, shows location labels for each key position.

Table 8-6 lists values in the scan code range (01H-99H) that the LK250 keyboard does not use to represent keys. However, in translate mode, the keyboard-interface controller translates them to the indicated industry-standard value.

Figure 8-1 Keyboard Position Labels

Table 8-5 LK250 Scan Codes and Industry-Standard Equivalent Value

LK250 Scan Code	Translated Industry-Standard Scan Code	Keyboard Position	Key Name
01H	43H	G08	F9
03H	3FH	G03	F5
04H	3DH	G01	F3
05H	3BH	G99	F1
06H	3CH	G00	F2
09H	44H	G09	F10
0AH	42H	G07	F8
0BH	40H	G05	F6
0CH	3EH	G02	F4
0DH	0FH	D00	Tab
0EH	29H	B00	` ~
11H	38H	A99	Alt
12H	2AH	B99	Left Shift
14H	1DH	C99	Ctrl
15H	10H	D01	q Q
16H	02H	E01	1 !
1AH	2CH	B01	z Z
1BH	1FH	C01	s S
1CH	1EH	C01	a A
1DH	11H	D02	w W
1EH	03H	E02	2 @
21H	2EH	B03	c C
22H	2DH	B02	x X
23H	20H	C03	d D
24H	12H	D03	e E
25H	05H	E04	4 $
26H	04H	E03	3 #
29H	39H	A01	Space Bar
2AH	2FH	B04	v V
2BH	21H	C04	f F
2CH	14H	D05	t T

Table 8-5 LK250 Scan Codes and Industry-Standard Equivalent Value (cont.)

LK250 Scan Code	Translated Industry-Standard Scan Code	Keyboard Position	Key Name
2DH	13H	D04	r R
2EH	06H	E05	5 %
31H	31H	B06	n N
32H	30H	B05	b B
33H	23H	C06	h H
34H	22H	C05	g G
35H	15H	D06	y Y
36H	07H	E06	6 ^
3AH	32H	B07	m M
3BH	24H	C07	j J
3CH	16H	D07	u U
3DH	08H	E07	7 &
3EH	09H	E08	8 *
41H	33H	B08	, <
42H	25H	C08	k K
43H	17H	D08	i I
44H	18H	D09	o O (Letter)
45H	0BH	E10	0)
46H	0AH	E09	9 (
49H	34H	B09	. >
4AH	35H	B10	/ ?
4BH	26H	C09	l L
4CH	27H	C10	; :
4DH	19H	D10	p P
4EH	0CH	E11	- _
52H	28H	C11	' "
54H	1AH	D11	[{
55H	0DH	E12	= +
58H	3AH	C00	Lock
59H	36H	B11	Right Shift
5AH	1CH	C13	Return
5BH	1BH	D12	] }

Table 8-5 LK250 Scan Codes and Industry-Standard Equivalent Value (cont.)

LK250 Scan Code	Translated Industry-Standard Scan Code	Keyboard Position	Key Name
5DH	2BH	E12	\ |
66H	0EH	E13	Delete (Word/Char)
69H	4FH	B20	1 End
6BH	4BH	C20	4 Left-Arrow
6CH	47H	D20	7 Home
70H	52H	A20	0 Ins
71H	53H	A22	. Del
72H	50H	B21	2 Down-Arrow
73H	4CH	C21	5
74H	4DH	C22	6 Right-Arrow
75H	48H	D21	8 Up-Arrow
76H	01H	E20	Esc
77H	45H	E21	NumLock
79H	4EH	C23	+ (Keypad)
7AH	51H	B22	3 PgDn
7BH	4AH	D23	- (Keypad)
7CH	37H	E23	PrtSc *
7DH	49H	D22	9 PgUp
7EH	46H	E22	ScrlLock Break
7FH	54H	———	SysReq (Alt/F20)
83H	41H	G06	F7
84H	54H	G23	F20
85H	55H	E16	Find
86H	56H	E17	Insert Here
87H	57H	E18	Remove
88H	58H	D16	Select
89H	59H	D17	Prev
8AH	5AH	D18	Next
8BH	5BH	C17	Up-Arrow
8CH	5CH	B16	Left-Arrow
8DH	5DH	B17	Down-Arrow
8EH	5EH	B18	Right-Arrow

Table 8-5 LK250 Scan Codes and Industry-Standard Equivalent Value (cont.)

LK250 Scan Code	Translated Industry-Standard Scan Code	Keyboard Position	Key Name
8FH	5FH	G11	F11
90H	60H	G12	F12
91H	61H	G13	F13
92H	62H	G14	F14
93H	63H	G15	Help
94H	64H	G16	Do
95H	65H	G20	F17
96H	66H	G21	F18
97H	67H	G22	F19
98H	68H	E00	Compose
99H	69H	A23	Enter (Keypad)

Table 8-6 Scan Codes Translated But Not Used

Unused Scan Code	Translated Industry-Standard Scan Code	Unused Scan Code	Translated Industry-Standard Scan Code
02H	41H	51H	73H
07H	58H	53H	74H
08H	64H	56H	62H
0FH	59H	57H	6EH
10H	65H	5CH	75H
13H	70H	5EH	63H
17H	5AH	5FH	76H
18H	66H	60H	55H
19H	71H	61H	56H
1FH	5BH	62H	77H
20H	67H	63H	78H
27H	5CH	64H	79H
28H	68H	65H	7AH
2FH	5DH	67H	7BH
30H	69H	68H	7CH
37H	5EH	6AH	7DH
38H	6AH	6DH	7EH
39H	72H	6EH	7FH
3FH	5FH	6FH	6FH
40H	6BH	78H	57H
47H	60H	80H	80H
48H	6CH	81H	81H
4FH	61H	82H	82H
50H	6DH	———	———

LK250 Keyboard Command Codes

Table 8-7 provides a summary of the LK250 command codes. Following Table 8-7 are descriptions of each command. The command descriptions give the purpose of the command, the actions taken by the keyboard, and the response code transmitted by the LK250 keyboard. The LK250 keyboard response codes are described later in this chapter.

Table 8-7 LK250 Keyboard Command Codes

Value	Description
00H-AAH	Invalid Commands
ABH	Request Keyboard ID (Digital Extended)
ACH	Enter Digital Extended Scan Code Mode
ADH	Exit Digital Extended Scan Code Mode
AEH	Set Keyboard LED (Digital Extended)
AFH	Reset Keyboard LED (Digital Extended)
B0H	Set Keyclick Volume (Digital Extended)
B1H	Enable Autorepeat
B2H	Disable Autorepeat
B3H	Keyboard Mode Lock
B4H	Keyboard Mode Unlock
B5H-ECH	Reserved
EDH	LEDs On/Off
EEH	Echo
EFH-F2H	Reserved
F3H	Set Autorepeat Delay and Rate
F4H	Enable Key Scanning
F5H	Disable Key Scanning and Restore to Defaults
F6H	Restore To Defaults
F7H-FDH	Reserved
FEH	Resend
FFH	Reset

Invalid Commands (00H-AAH)

Industry-Standard

When the LK250 keyboard receives a code in the range 00H-AAH, it transmits a resend request. The LK250 keyboard does not transmit an acknowledge (ACK) for codes in this range.

Request Keyboard ID (ABH)

DIGITAL Extended

When the LK250 keyboard receives the code ABH, the keyboard clears its output buffer and transmits a 2-byte response that identifies the version and mode of the LK250 keyboard. The first byte returned is the version number. The second byte is the current operating mode, 01H for industry-standard mode or 02H for DIGITAL extended mode.

To successfully execute this command, the keyboard-interface controller must be in pass-through mode. Otherwise, the keyboard controller attempts to translate the data to an industry-standard scan code.

The LK250 keyboard does not transmit an acknowledge (ACK) for this command.

Enter DIGITAL Extended Scan Code Mode (ACH)

DIGITAL Extended

When the LK250 keyboard receives the code ACH, the keyboard enables DIGITAL extended mode, turns off LED #4, and transmits an acknowledge (ACK).

Exit DIGITAL Extended Scan Code Mode (ADH)

DIGITAL Extended

When the LK250 keyboard receives the code ADH, the keyboard enables industry-standard mode, turns on LED #4, and transmits an acknowledge (ACK).

In this mode, only industry-standard scan codes are generated.

Set Keyboard LED (AEH)

DIGITAL Extended

When the LK250 keyboard receives the code AEH, the keyboard turns on LED #4, and transmits an acknowledge (ACK). For LEDs 1-3, see LEDs On/Off (EDH).

Reset Keyboard LED (AFH)
DIGITAL Extended

When the LK250 keyboard receives the code AFH, the keyboard turns off LED #4, and transmits an acknowledge (ACK). For LEDs 1-3, see LEDs On/Off (EDH).

Set Keyclick Volume (B0H)
DIGITAL Extended

This is a 2-byte command consisting of a command byte and a value byte. When the LK250 keyboard receives the code B0H, the keyboard stops scanning for keys, transmits an acknowledge (ACK), and waits for the second byte. When the LK250 keyboard receives the second byte, the keyboard sets the volume to the indicated value, transmits an acknowledge (ACK) and returns to the previous scanning state.

If bit 7 of the value byte is set to 1, the value is interpreted as a command. The current command is aborted, the new command is executed and the LK250 returns to the previous scanning state.

The volume value byte can have one of the following values:

Value	Volume
00H	No keyclick
02H	Soft
04H	Medium
06H	Loud

Enable Autorepeat (B1H)
DIGITAL Extended

When the LK250 keyboard receives the code B1H, the keyboard enables autorepeat for all keys and transmits an acknowledge (ACK). Autorepeat enabled is the default condition.

Disable Autorepeat (B2H)
DIGITAL Extended

When the LK250 keyboard receives the code B2H, the keyboard disables autorepeat for all keys and transmits an acknowledge (ACK).

Keyboard Mode Lock (B3H)

DIGITAL Extended

When the LK250 keyboard receives the code B3H, the keyboard disables the key combination ALT/F17 and transmits an acknowledge (ACK). The key combination ALT/F17 toggles the keyboard between DIGITAL extended mode and industry-standard mode. This key combination is detected and handled by the keyboard, not the ROM BIOS.

Keyboard Mode Unlock (B4H)

DIGITAL Extended

When the LK250 keyboard receives the code B4H, the keyboard enables the key combination ALT/F17 and transmits an acknowledge (ACK). The key combination ALT/F17 toggles the keyboard between DIGITAL extended mode and industry-standard mode. This key combination is detected and handled by the keyboard, not the ROM BIOS. ALT/F17 enabled is the default condition.

Reserved (B5H-ECH)

DIGITAL Extended

When the LK250 keyboard receives any of the reserved codes B5H-ECH, the keyboard transmits an acknowledge (ACK) and resumes its previous scanning state.

LEDs On/Off (EDH)

DIGITAL Extended

This is a 2-byte command consisting of a command byte and a value byte. When the LK250 keyboard receives the code EDH, the keyboard stops scanning for keys, transmits an acknowledge (ACK), and waits for the second byte. When the LK250 keyboard receives the second byte, the keyboard sets the LEDs to the indicated value, transmits an acknowledge (ACK) and returns to its previous scanning state.

If bit 7 of the value byte is set to 1, the value is interpreted as a command. The current command is aborted, the new command is executed and the LK250 returns to the previous scanning state.

The bits in the LED value byte have the following meanings:

Bit	Description
7-3	Reserved, always 0
2	CapsLock 0 = CapsLock LED off 1 = CapsLock LED on
1	NumLock 0 = NumLock LED off 1 = NumLock LED on
0	Scroll Lock 0 = Scroll Lock LED off 1 = Scroll Lock LED on

Echo (EEH)

Industry-Standard

When the LK250 keyboard receives the code EEH, the keyboard transmits the same echo code (EEH) and continues its previous scanning state. This command is a diagnostic tool.

Reserved (EFH-F2H)

Industry-Standard

When the LK250 keyboard receives any of the reserved codes EFH-F2H, the keyboard transmits an acknowledge (ACK) and resumes its previous scanning state.

Set Autorepeat Delay and Rate (F3H)

Industry-Standard

This is a 2-byte command consisting of a command byte and a value byte. When the LK250 keyboard receives the code F3H, the keyboard stops scanning for keys, transmits an acknowledge (ACK), and waits for the second byte. When the LK250 keyboard receives the second byte, the keyboard sets the autorepeat delay and rate to the indicated values, transmits an acknowledge (ACK) and returns to its previous scanning state.

If bit 7 of the value byte is set to 1, the value is interpreted as a command. The current command is aborted, the new command is executed and the LK250 returns to the previous scanning state.

The bits in the delay and rate value byte have the following meanings:

Bit	Description
7	Always 0
6-5	Autorepeat delay (± 20%) 00 = .25 seconds 01 = .5 seconds 10 = .75 seconds 11 = 1.0 second
4-0	Autorepeat rate (± 20%)

Bits 4-0	Rate per second	Bits 4-0	Rate per second
00000 =	29.98	10000 =	7.49
00001 =	26.65	10001 =	6.66
00010 =	23.98	10010 =	6.00
00011 =	21.80	10011 =	5.45
00100 =	19.98	10100 =	5.00
00101 =	18.45	10101 =	4.61
00110 =	17.13	10110 =	4.28
00111 =	15.99	10111 =	4.00
01000 =	14.99	11000 =	3.75
01001 =	13.32	11001 =	3.33
01010 =	11.99	11010 =	3.00
01011 =	10.90	11011 =	2.73
01100 =	9.99	11100 =	2.50
01101 =	9.22	11101 =	2.31
01110 =	8.56	11110 =	2.14
01111 =	7.99	11111 =	2.00

Enable Key Scanning (F4H)

Industry-Standard

When the LK250 keyboard receives the code F4H, the keyboard clears its output buffer, transmits an acknowledge (ACK), and begins scanning for keys. Use of this command assumes that key scanning was previously disabled. Any key strokes that were recognized but not transmitted are lost.

Disable Key Scanning and Restore to Defaults (F5H)

Industry-Standard

When the LK250 keyboard receives the code F5H, the keyboard stops scanning, clears its output buffer, restores conditions to the default powerup state, transmits an acknowledge (ACK), and waits for additional commands. Keyboard scanning remains disabled. Any key strokes that were recognized but not transmitted are lost.

Feature	Default
Autorepeat delay	.5 seconds
Autorepeat rate	29.98 per second
ALT/F17 mode toggle	Enabled
LEDs	Off
Digital extended mode	Disabled

Restore To Defaults (F6H)

Industry-Standard

When the LK250 keyboard receives the code F5H, the keyboard stops scanning, clears its output buffer, restores conditions to the default powerup state, transmits an acknowledge (ACK), and resumes its previous scanning state.

Feature	Default
Autorepeat delay	.5 seconds
Autorepeat rate	29.98 per second
ALT/F17 mode toggle	Enabled
LEDs	Off
Digital extended mode	Disabled

Reserved (F7H-FDH)

Industry-Standard

When the LK250 keyboard receives any of the reserved codes F7H-FDH, the keyboard transmits an acknowledge (ACK) and resumes its previous scanning state.

Resend (FEH)

Industry-Standard

When the LK250 keyboard receives the code FEH, the keyboard repeats its last transmission. This command is used when the keyboard-interface controller detects a reception error. It can be sent only in response to a transmission from the LK250 keyboard. Additionally, the resend command (FEH) must be transmitted before the keyboard-interface controller allows the keyboard to start another transmission. If the retransmitted data is still bad, the keyboard-interface controller places FFH in its output buffer and sets the parity error and timeout error status bits.

Reset (FFH)

Industry-Standard

When the LK250 keyboard receives the code FFH, the keyboard transmits an acknowledge (ACK) and waits for the the acknowledge to be accepted. Receipt of the acknowledge (ACK) is indicated by reading the acknowledge (ACK) from the keyboard-interface controller, setting port 2 bits 7-6 to 1, and leaving them set for at least 500 μs. When the keyboard recognizes acceptance of the acknowledge (ACK), the keyboard invokes its self-test diagnostic. On completing the self-test, the keyboard transmits AAH (self-test success) or FCH (self-test failure). As a result of the self-test, the keyboard is reset to default conditions and the keyboard output buffer is empty.

Feature	Default
Autorepeat delay	.5 seconds
Autorepeat rate	29.98 per second
ALT/F17 mode toggle	Enabled
LEDs	Off
Digital extended mode	Disabled

LK250 Keyboard Responses

The LK250 keyboard must reply to every transmission sent to it. Most of the time the reply is an acknowledge (ACK), but some commands have special responses. Refer to LK250 keyboard command descriptions for details. Table 8-8 lists the LK250 keyboard responses.

Table 8-8 LK250 Keyboard Responses

Value	Description
00H	Buffer Overrun
AAH	Self-Test Success
EEH	ECHO
FAH	Acknowledge (ACK)
FCH	Power-Up Self-Test Failure
FDH	Not Used
FEH	Resend

Buffer overrun (00H)

Industry-Standard

This code indicates that the LK250 keyboard output buffer (20 bytes) has been filled to capacity (19 key codes) and an attempt was made to store another code. The 20th code is replaced with 00H.

If the keyboard-interface controller is in translate mode, the LK250 keyboard buffer-overrun code 00H is translated to an industry-standard buffer overrun code of FFH.

Self-test success (AAH)

Industry-Standard

This code indicates successful completion of the LK250 keyboard self-test diagnostics. The self-test diagnostics are invoked as part of the powerup sequence and by the reset command (FFH).

ECHO (EEH)

Industry-Standard

This code indicates receipt of an echo command (EEH). In response to an echo command (EEH), the LK250 transmits echo (EEH) instead of acknowledge (ACK) (FAH).

Release Prefix (F0H)

Industry-Standard

This code indicates that a key was released. It is followed by the scan code of the released key.

If the keyboard-interface controller is in translate mode, this 2-byte sequence is translated to an industry-standard 1-byte release code by adding 80H to the translated scan code.

Acknowledge (ACK) (FAH)

Industry-Standard

This code is transmitted by the LK250 keyboard in response to most of the valid commands. Some commands have special responses. For example, request keyboard identification (ABH), echo (EEH), and resend (FEH). For additional information, see the descriptions of the LK250 keyboard commands.

Self-Test Failure (FCH)

Industry-Standard

This code transmitted by the LK250 keyboard to indicate that a keyboard component failed the self-test diagnostics. Self-test is invoked during the powerup sequence and the reset command (FFH).

Resend (FEH)

Industry-Standard

This code is transmitted by the LK250 keyboard in response to a keyboard-interface controller transmission containing a parity error. Providing that the last transmission to the keyboard was not a resend command (FEH), the keyboard interrupt handler should retransmit the last keyboard command.

LK250 Keyboard Error Handling

If the LK250 keyboard receives an invalid command or a parity error, it replies with a resend command.

U.S. and Foreign Keyboards

The LK250 keyboard is shipped with key legends appropriate for the destination country. Figure 8-2 through Figure 8-15 show the key legends for the various country keyboards.

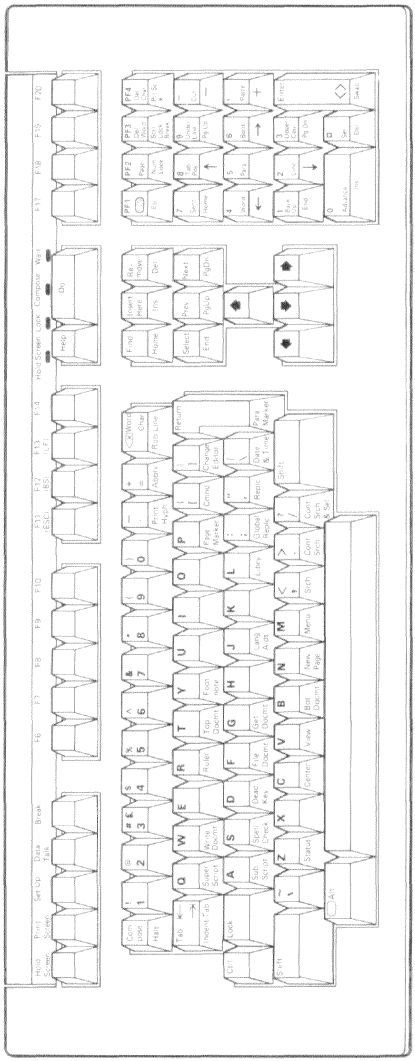


Figure 8-2 U.S./U.K. Keyboard

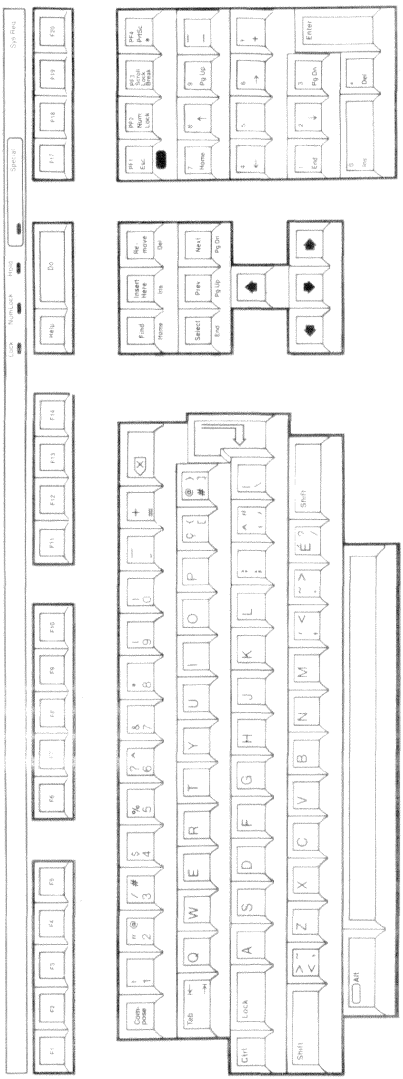


Figure 8-3 Canadian/English Keyboard

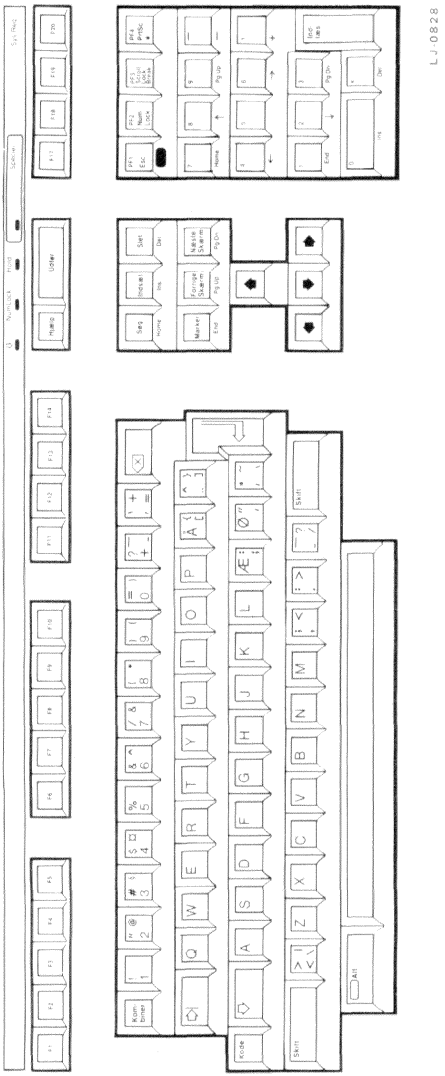


Figure 8-4 Danish Keyboard

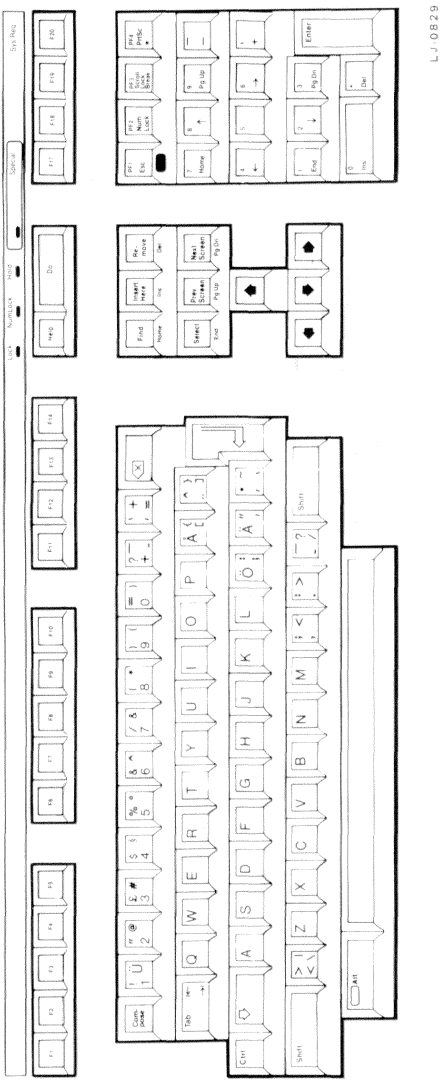


Figure 8-5 Finnish Keyboard

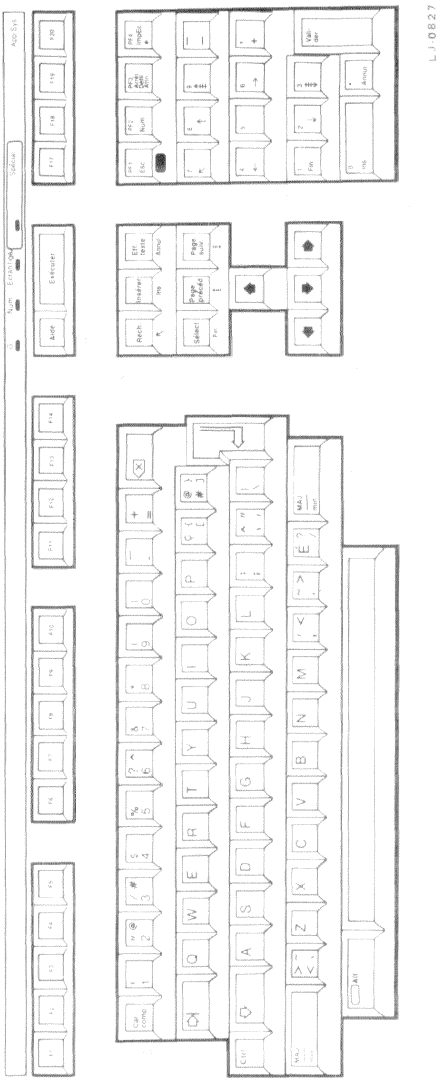


Figure 8-6 French/Canadian Keyboard

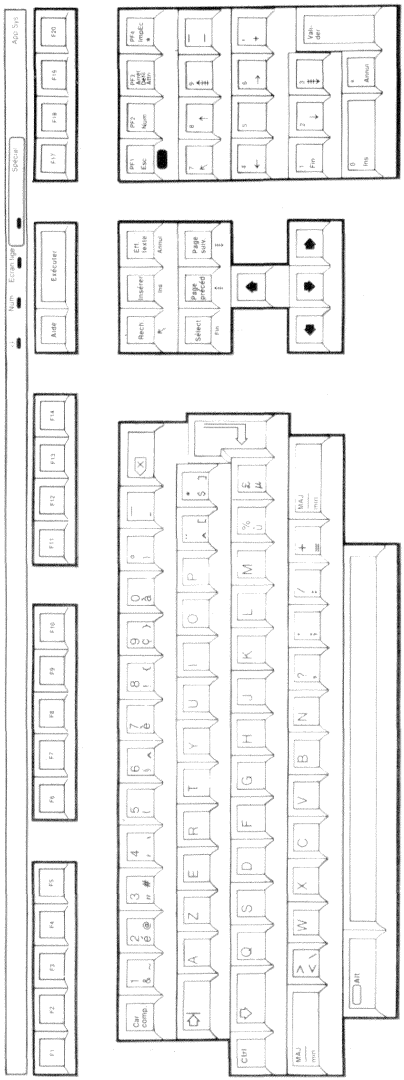


Figure 8-7 French Keyboard

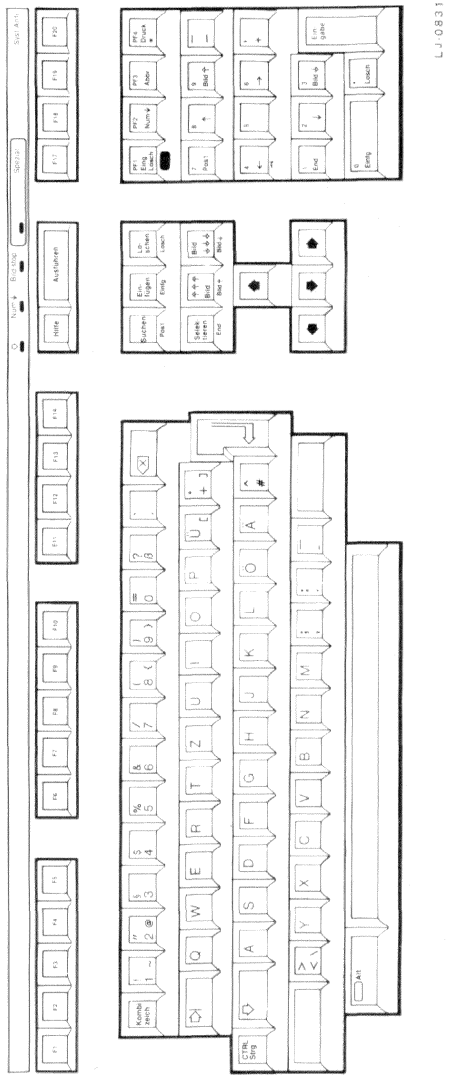


Figure 8-8 German/Austrian Keyboard

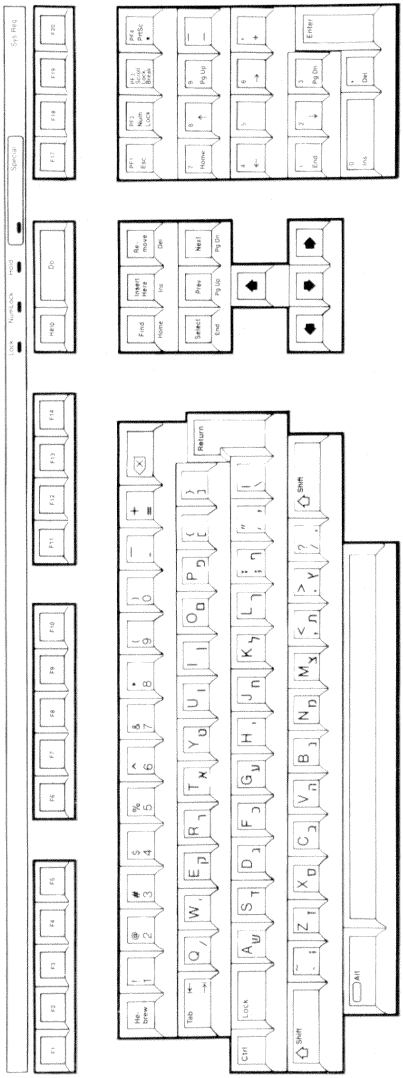


Figure 8-9 Hebrew Keyboard

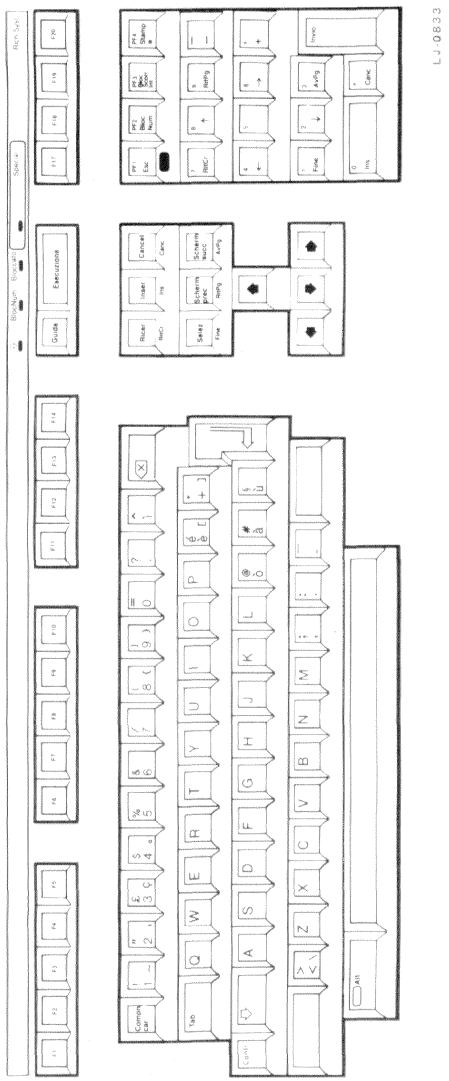


Figure 8-10 Italian Keyboard

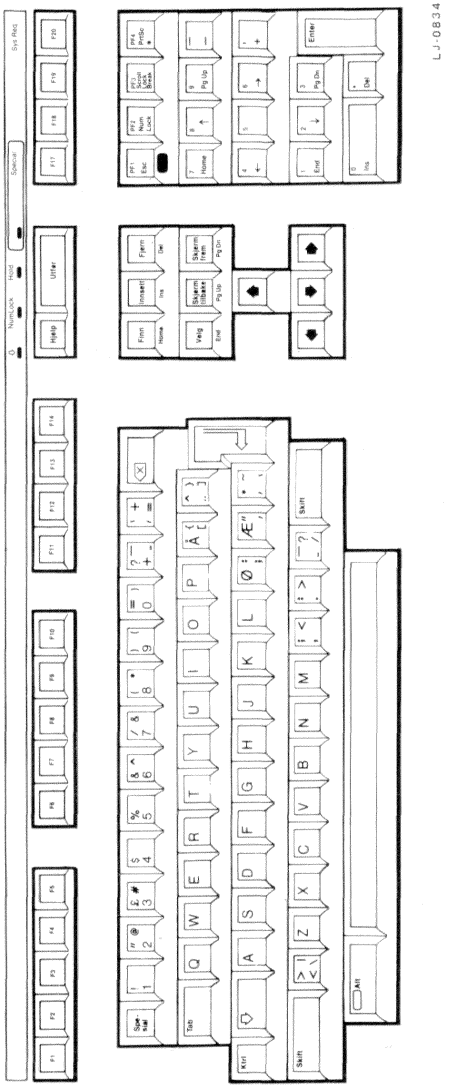


Figure 8-11 Norwegian Keyboard

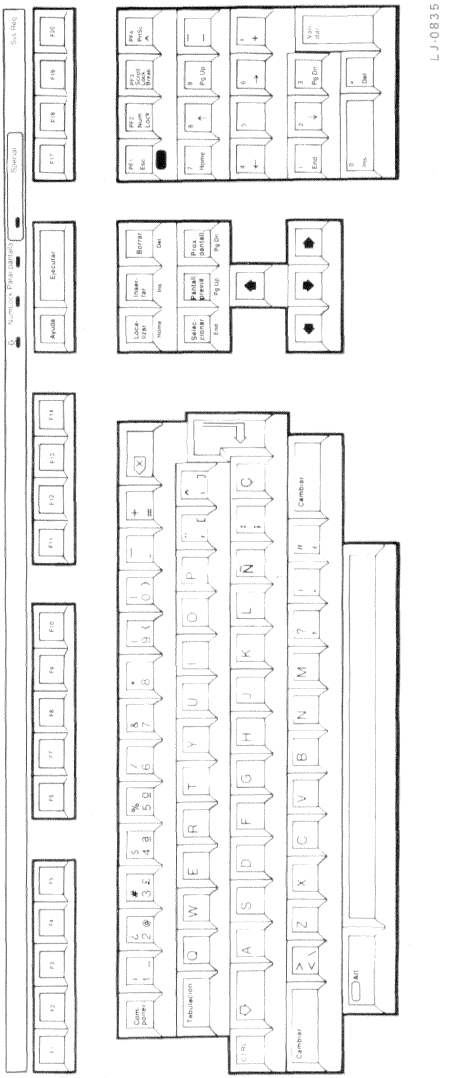


Figure 8-12 Spanish Keyboard

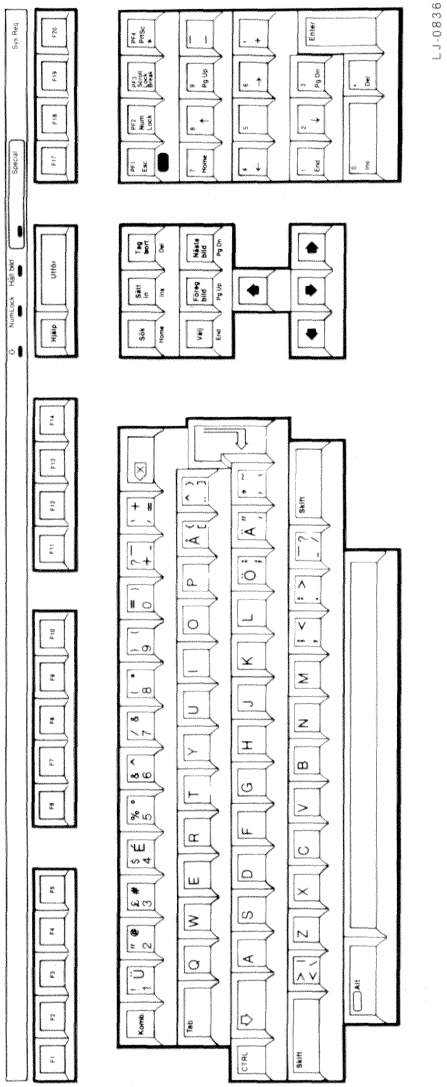


Figure 8-13 Swedish Keyboard

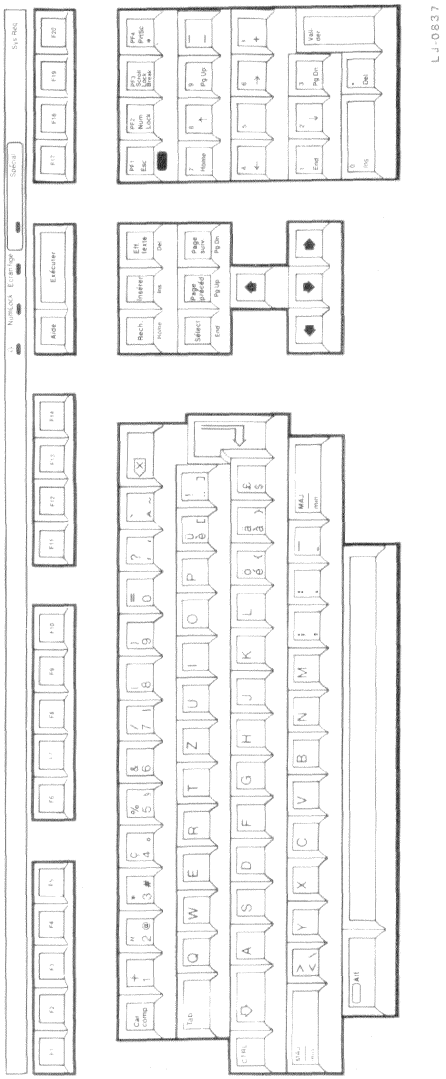


Figure 8-14 Swiss/French Keyboard

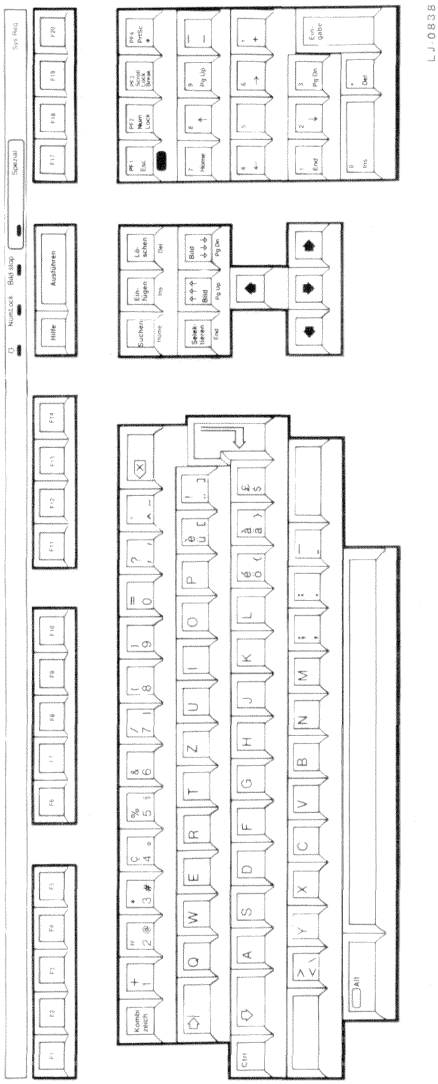


Figure 8-15 Swiss/German Keyboard

Programming Example

The subroutines in the keyboard example provide keyboard input support for all of the examples in this manual. The keyboard example demonstrates:

- Communicating with the keyboard-interface controller
- The use of keyboard translation tables
- Extended features of the LK250 keyboard

> **CAUTION**
> Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

The example provides routines as described in the following list.

wr_kcc	Writes a command byte to the keyboard-interface controller
wr_kcd	Writes a data byte to the keyboard-interface controller
pass_thru	Sets the keyboard-interface controller to pass-through mode
kyb_init	Prepares the interrupt structure for keyboard interrupts
wr_kyb	Places data in a ring buffer for output to the keyboard
kyb_led	Turns LEDs on or off according to the keyboard state
kyb_send	Retrieves data from a ring buffer and writes it to the keyboard
kyb_rest	Restores the previous keyboard interrupt structure
get_key	Gets a 1-byte value from the keyboard input ring buffer
kyb_int_hand	Handles keyboard-interface-controller interrupts and performs scan code translations
kyb_exm	Provides an example environment for examining various aspects of the keyboard

```
#include "rb.h"
#include "example.h"

/*************************************************************************/
/* define constants and structures used in keyboard examples             */
/*************************************************************************/

#define COMMAND 0x64                    /* command register in I/O space */
#define DATAREG 0x60                       /* data register in I/O space */

/* define mask bits for keyboard state flag */

#define S_LSHF  0x01                      /* left  shift key is pressed */
#define S_RSHF  0x02                      /* right shift key is pressed */
#define S_CTRL  0x04                          /* control key is pressed */
#define S_ALT   0x08                        /* Alternate key is pressed */
#define S_SCRL  0x10                        /* Scroll lock is in effect */
#define S_NUM   0x20                      /* Numerics lock is in effect */
#define S_CAPS  0x40                          /* Caps lock is in effect */
#define S_INS   0x80                           /* Insert mode is active */

/* define modifier-key values returned by keyboard */

#define CTRL    0x1d                                    /* control key */
#define LSHF    0x2a                                  /* left shift key */
#define RSHF    0x36                                 /* right shift key */
#define ALT     0x38                                   /* alternate key */
#define CAPS    0x3a                               /* Lock/Caps Lock key */
#define NUML    0x45                                     /* NumLock key */
#define SCRL    0x46                                    /* ScrlLock key */
#define INS     0x52                                 /* Ins (Insert) key */
#define DEL     0x53                                        /* Del key */

/* define some keyboard interface controller commands */

#define RDCB    0x20                                 /* read command byte */
#define WRCB    0x60                                /* write command byte */
```

```
/* define some keyboard commands */

#define LED123  0xed                          /* control leds 1,2, and 3 */
#define LED4_ON 0xae                                  /* turn led #4 on */
#define LED4_OF 0xaf                                 /* turn led #4 off */
#define ENT_EXM 0xac          /* enter DIGITAL extended keyboard mode */
#define EXT_EXM 0xad           /* exit DIGITAL extended keyboard mode */
#define AREPON  0xb1                                  /* auto-repeat on */
#define AREPOFF 0xb2                                 /* auto-repeat off */
#define SETAR   0xf3                          /* set auto-repeat rate */
#define SETVOL  0xb0                            /* set speaker volume */
#define KYBID   0xab                  /* return keyboard ID and state */
#define RESDEF  0xf6               /* reset keyboard to default values */

/* define some keyboard responses */

#define B_FULL  0x00                              /* keyboard buffer full */
#define RESEND  0xfe             /* request to resend command or data */
#define ACK     0xfa                              /* keyboard acknowledge */

/* define some state dependent keyboard table index constants */
/* Note: The ROM BIOS has separate alpha and numeric tables. */
/*       This example combines the alpha and numeric tables. */

#define T_NORM  0x00                         /* look in normal key table */
#define T_ALT   0x02           /* look in alternate combination table */
#define T_CTRL  0x04                      /* look in control key table */
#define T_SHFT  0x06                           /* look in shift table */
#define T_A_N   0x08                     /* look in alphanumeric table */

#define KB_SIZ  128       /* example keyboard buffer size is 128 bytes */
```

```
/*************************************************************************/
/* define key translation tables and variables                           */
/*************************************************************************/

unsigned char kyb_state;                    /* state of modifier keys */
unsigned char last_send;              /* last character sent to keyboard */
unsigned char key_buff[2][KB_SIZ];     /* place to store incoming keys */
unsigned char released;                  /* last released key for demo */
unsigned char depressed;                /* last depressed key for demo */

RING_BUFF ki_rb;                       /* ring buffer control structure */
RING_BUFF ko_rb;                       /* ring buffer control structure */

/* define  the keyboard tables */
/* Note: The ROM BIOS has separate alpha and numeric tables. */
/*       This example combines the alpha and numeric tables. */

static unsigned char keyboard[0x70][9] =
{
/* NORMAL  |    ALT    | CONTROL  |   SHIFT    | A/N  */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,       /* overrun */
0x1b, 0x01, 0xff, 0xff, 0x1b, 0x01, 0x1b, 0x01, 0x00,          /* ESC */
0x31, 0x02, 0x00, 0x78, 0xff, 0xff, 0x21, 0x02, 0x00,            /* 1 */
0x32, 0x03, 0x00, 0x79, 0x00, 0x03, 0x40, 0x03, 0x00,            /* 2 */
0x33, 0x04, 0x00, 0x7a, 0xff, 0xff, 0x23, 0x04, 0x00,            /* 3 */
0x34, 0x05, 0x00, 0x7b, 0xff, 0xff, 0x24, 0x05, 0x00,            /* 4 */
0x35, 0x06, 0x00, 0x7c, 0xff, 0xff, 0x25, 0x06, 0x00,            /* 5 */
0x36, 0x07, 0x00, 0x7d, 0x1e, 0x07, 0x5e, 0x07, 0x00,            /* 6 */
0x37, 0x08, 0x00, 0x7e, 0xff, 0xff, 0x26, 0x08, 0x00,            /* 7 */
0x38, 0x09, 0x00, 0x7f, 0xff, 0xff, 0x2a, 0x09, 0x00,            /* 8 */
0x39, 0x0a, 0x00, 0x80, 0xff, 0xff, 0x28, 0x0a, 0x00,            /* 9 */
0x30, 0x0b, 0x00, 0x81, 0xff, 0xff, 0x29, 0x0b, 0x00,            /* 0 */
0x2d, 0x0c, 0x00, 0x82, 0x1f, 0x0c, 0x5f, 0x0c, 0x00,            /* - */
0x3d, 0x0d, 0x00, 0x83, 0xff, 0xff, 0x2b, 0x0d, 0x00,            /* = */
0x08, 0x0e, 0xff, 0xff, 0x7f, 0x0e, 0x08, 0x0e, 0x00,           /* BS */
0x09, 0x0f, 0xff, 0xff, 0xff, 0xff, 0x00, 0x0f, 0x00,          /* TAB */
0x71, 0x10, 0x00, 0x10, 0x11, 0x10, 0x51, 0x10, S_CAPS,          /* Q */
0x77, 0x11, 0x00, 0x11, 0x17, 0x11, 0x57, 0x11, S_CAPS,          /* W */
0x65, 0x12, 0x00, 0x12, 0x05, 0x12, 0x45, 0x12, S_CAPS,          /* E */
0x72, 0x13, 0x00, 0x13, 0x12, 0x13, 0x52, 0x13, S_CAPS,          /* R */
0x74, 0x14, 0x00, 0x14, 0x14, 0x14, 0x54, 0x14, S_CAPS,          /* T */
0x79, 0x15, 0x00, 0x15, 0x19, 0x15, 0x59, 0x15, S_CAPS,          /* Y */
0x75, 0x16, 0x00, 0x16, 0x15, 0x16, 0x55, 0x16, S_CAPS,          /* U */
0x69, 0x17, 0x00, 0x17, 0x09, 0x17, 0x49, 0x17, S_CAPS,          /* I */
0x6f, 0x18, 0x00, 0x18, 0x0f, 0x18, 0x4f, 0x18, S_CAPS,          /* O */
0x70, 0x19, 0x00, 0x19, 0x10, 0x19, 0x50, 0x19, S_CAPS,          /* P */
0x5b, 0x1a, 0xff, 0xff, 0x1b, 0x1a, 0x7b, 0x1a, 0x00,            /* [ */
```

```
0x5d, 0x1b, 0xff, 0xff, 0x1d, 0x1b, 0x7d, 0x1b, 0x00,          /* ] */
0x0d, 0x1c, 0xff, 0xff, 0x0a, 0x1c, 0x0d, 0x1c, 0x00,        /* RET */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,       /* CTRL */
0x61, 0x1e, 0x00, 0x1e, 0x01, 0x1e, 0x41, 0x1e, S_CAPS,        /* A */
0x73, 0x1f, 0x00, 0x1f, 0x13, 0x1f, 0x53, 0x1f, S_CAPS,        /* S */
0x64, 0x20, 0x00, 0x20, 0x04, 0x20, 0x44, 0x20, S_CAPS,        /* D */
0x66, 0x21, 0x00, 0x21, 0x06, 0x21, 0x46, 0x21, S_CAPS,        /* F */
0x67, 0x22, 0x00, 0x22, 0x07, 0x22, 0x47, 0x22, S_CAPS,        /* G */
0x68, 0x23, 0x00, 0x23, 0x08, 0x23, 0x48, 0x23, S_CAPS,        /* G */
0x6a, 0x24, 0x00, 0x24, 0x0a, 0x24, 0x4a, 0x24, S_CAPS,        /* J */
0x6b, 0x25, 0x00, 0x25, 0x0b, 0x25, 0x4b, 0x25, S_CAPS,        /* K */
0x6c, 0x26, 0x00, 0x26, 0x0c, 0x26, 0x4c, 0x26, S_CAPS,        /* L */
0x3b, 0x27, 0xff, 0xff, 0xff, 0xff, 0x3a, 0x27, 0x00,          /* ; */
0x27, 0x28, 0xff, 0xff, 0xff, 0xff, 0x22, 0x28, 0x00,          /* ' */
0x60, 0x29, 0xff, 0xff, 0xff, 0xff, 0x7e, 0x29, 0x00,          /* ` */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,         /* LS */
0x5c, 0x2b, 0xff, 0xff, 0x1c, 0x2b, 0x7c, 0x2b, 0x00,          /* \ */
0x7a, 0x2c, 0x00, 0x2c, 0x1a, 0x2c, 0x5a, 0x2c, S_CAPS,        /* Z */
0x78, 0x2d, 0x00, 0x2d, 0x18, 0x2d, 0x58, 0x2d, S_CAPS,        /* X */
0x63, 0x2e, 0x00, 0x2e, 0x03, 0x2e, 0x43, 0x2e, S_CAPS,        /* C */
0x76, 0x2f, 0x00, 0x2f, 0x16, 0x2f, 0x56, 0x2f, S_CAPS,        /* V */
0x62, 0x30, 0x00, 0x30, 0x02, 0x30, 0x42, 0x30, S_CAPS,        /* B */
0x6e, 0x31, 0x00, 0x31, 0x0e, 0x31, 0x4e, 0x31, S_CAPS,        /* N */
0x6d, 0x32, 0x00, 0x32, 0x0d, 0x32, 0x4d, 0x32, S_CAPS,        /* M */
0x2c, 0x33, 0xff, 0xff, 0xff, 0xff, 0x3c, 0x33, 0x00,          /* , */
0x2e, 0x34, 0xff, 0xff, 0xff, 0xff, 0x3e, 0x34, 0x00,          /* . */
0x2f, 0x35, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x35, 0x00,          /* / */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,  /* Right Shift */
0x2a, 0x37, 0xff, 0xff, 0x00, 0x72, 0xff, 0xff, 0x00,       /* PRTSC */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,         /* ALT */
0x20, 0x39, 0x20, 0x39, 0x20, 0x39, 0x20, 0x39, 0x00,    /* Space Bar */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,    /* CAPS Lock */
0x00, 0x3b, 0x00, 0x68, 0x00, 0x5e, 0x00, 0x54, 0x00,          /* F1 */
0x00, 0x3c, 0x00, 0x69, 0x00, 0x5f, 0x00, 0x55, 0x00,          /* F2 */
0x00, 0x3d, 0x00, 0x6a, 0x00, 0x60, 0x00, 0x56, 0x00,          /* F3 */
0x00, 0x3e, 0x00, 0x6b, 0x00, 0x61, 0x00, 0x57, 0x00,          /* F4 */
0x00, 0x3f, 0x00, 0x6c, 0x00, 0x62, 0x00, 0x58, 0x00,          /* F5 */
0x00, 0x40, 0x00, 0x6d, 0x00, 0x63, 0x00, 0x59, 0x00,          /* F6 */
0x00, 0x41, 0x00, 0x6e, 0x00, 0x64, 0x00, 0x5a, 0x00,          /* F7 */
0x00, 0x42, 0x00, 0x6f, 0x00, 0x65, 0x00, 0x5b, 0x00,          /* F8 */
0x00, 0x43, 0x00, 0x70, 0x00, 0x66, 0x00, 0x5c, 0x00,          /* F9 */
0x00, 0x44, 0x00, 0x71, 0x00, 0x67, 0x00, 0x5d, 0x00,         /* F10 */
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,     /* NumLock */
0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x00,     /* ScrlLock */
0x00, 0x47, 0x07, 0x00, 0x00, 0x77, 0x37, 0x47, S_NUM,    /* keypad 7 */
0x00, 0x48, 0x08, 0x00, 0xff, 0xff, 0x38, 0x48, S_NUM,    /* keypad 8 */
0x00, 0x49, 0x09, 0x00, 0x00, 0x84, 0x39, 0x49, S_NUM,    /* keypad 9 */
```

```
0x2d, 0x4a, 0xff, 0xff, 0xff, 0xff, 0x2d, 0x4a, 0x00,      /* keypad - */
0x00, 0x4b, 0x04, 0x00, 0x00, 0x73, 0x34, 0x4b, S_NUM,     /* keypad 4 */
0xff, 0xff, 0x05, 0x00, 0xff, 0xff, 0x35, 0x4c, S_NUM,     /* keypad 5 */
0x00, 0x4d, 0x06, 0x00, 0x00, 0x74, 0x36, 0x4d, S_NUM,     /* keypad 6 */
0x2b, 0x4e, 0xff, 0xff, 0xff, 0xff, 0x2b, 0x4e, 0x00,      /* keypad + */
0x00, 0x4f, 0x01, 0x00, 0x00, 0x75, 0x31, 0x4f, S_NUM,     /* keypad 1 */
0x00, 0x50, 0x02, 0x00, 0xff, 0xff, 0x32, 0x50, S_NUM,     /* keypad 2 */
0x00, 0x51, 0x03, 0x00, 0x00, 0x76, 0x33, 0x51, S_NUM,     /* keypad 3 */
0x00, 0x52, 0x00, 0x00, 0xff, 0xff, 0x30, 0x52, S_NUM,     /* keypad 0 */
0x00, 0x53, 0xff, 0xff, 0xff, 0xff, 0x2e, 0x53, S_NUM,     /* keypad . */
0x00, 0x98, 0xff, 0xff, 0x00, 0xb0, 0x00, 0xa4, 0x00,          /* F20 */
0x00, 0x85, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,         /* FIND */
0x00, 0x86, 0xff, 0xff, 0x00, 0xc3, 0xff, 0xff, 0x00,       /* INSERT */
0x00, 0x87, 0xff, 0xff, 0x00, 0xc1, 0xff, 0xff, 0x00,       /* REMOVE */
0x00, 0x88, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,       /* SELECT */
0x00, 0x89, 0xff, 0xff, 0x00, 0xc4, 0xff, 0xff, 0x00,         /* PREV */
0x00, 0x8a, 0xff, 0xff, 0x00, 0xc2, 0xff, 0xff, 0x00,         /* NEXT */
0x00, 0x8b, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,           /* UP */
0x00, 0x8c, 0xff, 0xff, 0x00, 0xbf, 0xff, 0xff, 0x00,           /* LT */
0x00, 0x8d, 0xff, 0xff, 0x00, 0xc0, 0xff, 0xff, 0x00,           /* RT */
0x00, 0x8e, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,           /* DN */
0x00, 0x8f, 0x00, 0xb3, 0x00, 0xa7, 0x00, 0x9b, 0x00,          /* F11 */
0x00, 0x90, 0x00, 0xb4, 0x00, 0xa8, 0x00, 0x9c, 0x00,          /* F12 */
0x00, 0x91, 0x00, 0xb5, 0x00, 0xa9, 0x00, 0x9d, 0x00,          /* F13 */
0x00, 0x92, 0x00, 0xb6, 0x00, 0xaa, 0x00, 0x9e, 0x00,          /* F14 */
0x00, 0x93, 0x00, 0xb7, 0x00, 0xab, 0x00, 0x9f, 0x00,          /* F15 */
0x00, 0x94, 0x00, 0xb8, 0x00, 0xac, 0x00, 0xa0, 0x00,          /* F16 */
0x00, 0x95, 0x00, 0xb9, 0x00, 0xad, 0x00, 0xa1, 0x00,          /* F17 */
0x00, 0x96, 0x00, 0xba, 0x00, 0xae, 0x00, 0xa2, 0x00,          /* F18 */
0x00, 0x97, 0x00, 0xbb, 0x00, 0xaf, 0x00, 0xa3, 0x00,          /* F19 */
0x00, 0xbd, 0x00, 0xbd, 0x00, 0xbd, 0x00, 0xbd, 0x00,       /* COMPOSE */
0x0d, 0x9a, 0x00, 0xbe, 0x00, 0xb2, 0x00, 0xa6, 0x00,         /* ENTER */
};
```

```
/*************************************************************************/
/* wr_kcc() - Write keyboard controller command                          */
/*************************************************************************/

wr_kcc(cmd)

unsigned char cmd;                              /* command byte to write */

{
  outp(COMMAND, cmd);                           /* write command byte */
  while((inp(COMMAND) & 0x02))          /* wait until KC has read command */
    ;
}

/*************************************************************************/
/* wr_kcd() - Write keyboard controller data                             */
/*************************************************************************/

wr_kcd(data)

unsigned char data;                             /* data byte to write */

{
  outp(DATAREG, data);                          /* write command data */
  while((inp(COMMAND) & 0x02))            /* wait until KC has read data */
    ;
}
```

```
/*************************************************************************/
/* pass_thru() - set keyboard controller pass through mode               */
/*************************************************************************/

pass_thru(flag)

int flag;                                   /* if TRUE, set pass through */
                                              /* else clear pass through */
{
unsigned char c;                    /* tmp to hold internal command byte */
unsigned int intr_flag;                      /* tmp to hold CPU IF state */

  intr_flag = int_off();                              /* turn interrupts off */
  wr_kcc(RDCB);                              /* give me current command byte */
  while(!(inp(COMMAND) & 0x01))             /* wait until output buffer full */
    ;
  c = inp(DATAREG);                    /* read command byte from data buffer */
  if(flag) c &= ~0x40;/* if TRUE, do not decode keyboard transmissions */
  else c |= 0x40;                     /* else, decode keyboard transmissions */
  wr_kcc(WRCB);      /* tell interface controller to write command byte */
  wr_kcd(c);                                  /* send command byte to write */
  int_on(intr_flag);                                    /* allow interrupts */
}

/*************************************************************************/
/* kyb_init() - keyboard interrupt initialization                        */
/*************************************************************************/

kyb_init()
{
  kyb_state = 0;                                    /* no states established */
                                                   /* establish ring buffers */
  init_rb(&ki_rb, &key_buff[0][0], KB_SIZ, 30, 15);
  init_rb(&ko_rb, &key_buff[1][0], KB_SIZ, 30, 15);
  imask(0,1,0);              /* disable PIC input for keyboard interface */
  iv_init(0x09);                        /* keyboard interrupt is int 0x09 */
  imask(0,1,1);               /* enable PIC input for keyboard interface */
}
```

```
/*************************************************************************/
/* wr_kyb() - put value in output buffer to keyboard and start send     */
/*************************************************************************/

wr_kyb(value)

unsigned char value;                        /* value to send to keyboard */

{
unsigned intr_flag;                      /* to hold current CPU IF state */

  intr_flag = int_off();                          /* CPU interrupts off */
  rb_in(&ko_rb, value);                     /* put value in ring buffer */
  if(!last_send) kyb_send(0);        /* if keyboard not waiting for ACK */
  int_on(intr_flag);                           /* allow interrupts now */
}

/*************************************************************************/
/* kyb_led() - handle changes to LED state                               */
/*************************************************************************/

kyb_led()
{
unsigned char state;                        /* temporary state variable */

  state = kyb_state >> 4;                        /* shift into position */
  state &= 0x07;                              /* only bits 2:0 are valid */
  wr_kyb(LED123);                    /* keyboard LED command to buffer */
  wr_kyb(state);                               /* LED state to buffer */
}

/*************************************************************************/
/* kyb_send() - send command to keyboard                                 */
/*************************************************************************/

kyb_send(resend)

int resend;                                /* re-send character if true */

{
  if(resend) outp(DATAREG, last_send);       /* re-send command or data */
  else if(rb_out(&ko_rb, &last_send) >= 0)/* get char from output buff */
    outp(DATAREG, last_send);                  /* send command or data */
  else last_send = 0;                          /* no character to send */
}
```

```
/*************************************************************************/
/* kyb_rest() - restore keyboard interrupts to system                    */
/*************************************************************************/

kyb_rest()
{
    iv_rest(0x09);                    /* keyboard interrupt is int 0x09 */
}

/*************************************************************************/
/* get_key() - get character from keyboard input buffer                  */
/*************************************************************************/

int get_key(pc)                                /* get char from input buf */

unsigned char *pc;                              /* where to put character */

{
    return(rb_out(&ki_rb, pc));               /* get char from input buff */
}

/*************************************************************************/
/* put_key() - put key sequence into keyboard input buffer               */
/*************************************************************************/

int put_key(pc)                              /* put key seq into input buf */

unsigned char *pc;                                /* where to get sequence */

{
    if(pc[0] == 0xff && pc[1] == 0xff) beep();  /* invalid key combo ? */
    else
    {
      rb_in(&ki_rb, *pc++);                       /* write ASCII character */
      return(rb_in(&ki_rb, *pc));                       /* write scan code */
    }
}
```

```
/*************************************************************************/
/* kyb_int_hand() - keyboard interrupt handler                           */
/*************************************************************************/

kyb_int_hand()
{

unsigned char key;                  /* tmp to hold keyboard transmission */

  key = inp(DATAREG);                       /* get keyboard transmission */
  if(key == B_FULL) beep();                  /* tell typist, buffer full */
  else if(key & 0x80)                /* key release or keyboard reply ? */
  {
    switch(key)
    {
      case ACK:                                /* keyboard acknowledge ? */
        kyb_send(0);
        break;

      case RESEND:                           /* keyboard re-send request ? */
        kyb_send(1);
        break;

      default:                                   /* must be a key release */
        switch(released = key & 0x7f)     /* save released key for demo */
        {                                   /* and check modifier keys */
          case CTRL:                         /* control key released ? */
            kyb_state &= ~S_CTRL;             /* clear from state byte */
            break;

          case LSHF:                        /* left shift key released ? */
            kyb_state &= ~S_LSHF;             /* clear from state byte */
            break;

          case RSHF:                       /* right shift key released ? */
            kyb_state &= ~S_RSHF;             /* clear from state byte */
            break;

          case ALT:                         /* alternate key released ? */
            kyb_state &= ~S_ALT;              /* clear from state byte */
            break;

          case CAPS:       /* caps lock, numlock or scrllock released ? */
          case NUML:        /* these are toggle keys, only means it was */
          case SCRL:      /* released. toggle is performed when pressed */
              break;
```

```
        case INS:                               /* insert key released ? */
          kyb_state &= ~S_INS;                  /* clear from state byte */
          break;

        default:                                          /* no default */
          break;
      }
  }
}
else switch(depressed = key & 0x7f)      /* save depress key for demo */
{
  case CTRL:                                 /* control key pressed ? */
    kyb_state |= S_CTRL;                        /* set in state byte */
    break;

  case LSHF:                               /* left shift key pressed ? */
    kyb_state |= S_LSHF;                        /* set in state byte */
    break;

  case RSHF:                              /* right shift key pressed ? */
    kyb_state |= S_RSHF;                        /* set in state byte */
    break;

  case ALT:                                /* alternate key pressed ? */
    kyb_state |= S_ALT;                         /* set in state byte */
    break;

  case CAPS:                               /* caps lock key pressed ? */
    if(kyb_state & S_CAPS)                                /* caps on ? */
      kyb_state &= ~S_CAPS;                          /* turn caps off */
    else kyb_state |= S_CAPS;              /* otherwise, turn caps on */
    kyb_led();                                      /* adjust Lock LED */
    break;

  case NUML:                                /* numlock key pressed ? */
    if(kyb_state & S_NUM)                                 /* numlock on ? */
      kyb_state &= ~S_NUM;                        /* turn numlock off */
    else kyb_state |= S_NUM;           /* otherwise, turn numlock on ? */
    kyb_led();                                   /* adjust NumLock LED */
    break;

  case SCRL:                               /* scrllock key pressed ? */
    if(kyb_state & S_SCRL)                               /* scrllock on ? */
      kyb_state &= ~S_SCRL;                      /* turn scrllock off */
    else kyb_state |= S_SCRL;           /* otherwise, turn scrllock on */
    kyb_led();                                  /* adjust ScrlLock LED */
```

```
    break;

  case INS:                                    /* insert key pressed ? */
    kyb_state |= S_INS;                          /* set in state byte */
    break;

  default:                                   /* test for combination keys */
    if(kyb_state & S_ALT)                        /* alt key pressed ? */
    {
      if(kyb_state & S_CTRL)                   /* ctrl key pressed ? */
      {
          if(key == DEL) sys_reset();              /* CTRL/ALT/DEL ? */
          else beep();           /* NOTE: No ALT/CTRL table in demo. */
      }
      else if(kyb_state & (S_LSHF | S_RSHF)) /* shift key pressed ? */
      {
        beep();                  /* NOTE: No ALT/SHIFT table in demo */
      }
      else put_key(&keyboard[key][T_ALT]);         /* use alt table */
    }
    else if(kyb_state & S_CTRL)                /* ctrl key pressed ? */
    {
      if(kyb_state & (S_LSHF | S_RSHF))      /* shift key pressed ? */
      {
        beep();                /* NOTE: No CTRL/SHIFT table in demo */
      }
      else put_key(&keyboard[key][,T_CTRL]);       /* use ctrl table */
    }
    else if(kyb_state & (S_LSHF | S_RSHF))   /* shift key pressed ? */
    {
      if(keyboard[key][T_A_N] & S_CAPS)        /* alpha character ? */

        if(kyb_state & S_CAPS)             /* caps lock in effect ? */
          put_key(&keyboard[key][T_NORM]);    /* use normal table ? */
        else put_key(&keyboard[key][T_SHFT]);    /* use shift table */
```

```
        }
        else if(keyboard[key][T_A_N] & S_NUM)  /* numeric character ? */
        {
          if(kyb_state & S_NUM)                /* numlock in effect ? */
            put_key(&keyboard[key][T_NORM]);     /* use normal table */
          else put_key(&keyboard[key][T_SHFT]);   /* use shift table */
        }
        else put_key(&keyboard[key][T_SHFT]);     /* use shift table */
      }
      else put_key(&keyboard[key][T_NORM]);       /* use normal table */
      break;
  }
  eoi(0);                                    /* end of interrupt to PIC */
}
```

```
/*************************************************************************/
/* kyb_exm() - keyboard example program                                  */
/*************************************************************************/

kyb_exm()
{
static MESSAGE mmain[] =                                  /* opening menu */
{
    {  3, 24, "Keyboard Example" },
    {  5, 24, "F1.  Toggle DIGITAL extended mode on or off" },
    {  6, 24, "F2.  Set key click volume" },
    {  7, 24, "F3.  Toggle autorepeat on or off" },
    {  8, 24, "F4.  Set autorepeat delay and rate" },
    {  9, 24, "F5.  Show keyboard version and mode" },
    { 10, 24, "F6.  Restore keyboard to defaults" },
    { 11, 24, "F7.  Show last depressed and last released keys" },
    { 12, 24, "F10. Return to Main menu" },
    {  0,  0,  0 },
};

char line[512];                                  /* to hold input line */
int ext_mode = 0;                  /* to hold extended mode toggle state */
int auto_rep = 0;                    /* to hold auto-repeat toggle state */
int d;                                                  /* to hold input */
int x;                                                  /* to hold input */
int y;                                                  /* to hold input */
unsigned int intr_flag;                          /* to hold CPU IF state */

#define ROW 14                                 /* where to put input lines */
#define COL 17

  wr_kyb(AREPON);                               /* ensure autorepeat is on */
  wr_kyb(EXT_EXM);                                   /* exit extended mode */
  wr_kyb(LED4_ON);                                           /* led #4 on */
  line[0] = 0;                                               /* null line */
  while(1)                                          /* forever until F10 */
  {
    disp_menu(mmain);                /* display menu for keyboard example */
    switch(line[0])                                /* which function key ? */
    {
      case F1:                                     /* toggle extended mode */
        if(ext_mode)                                 /* in extended mode ? */
        {
          wr_kyb(EXT_EXM);                         /* exit extended mode */
          wr_kyb(LED4_ON);                                   /* led #4 on */
          ext_mode = 0;                                /* clear toggle */
        }
```

```
  else                                        /* not in extended mode */
  {
    wr_kyb(ENT_EXM);                          /* enter extended mode */
    wr_kyb(LED4_OF);                                 /* led #4 off */
    ext_mode = 1;                                    /* set toggle */
  }
  break;

case F2:                                    /* change keyclick volume */
  disp_str(ROW, COL,
   "The key click volume has a range of 0 to 6 in 4 increments");
  disp_str(ROW + 1, COL,
     "0 = Off  2 = Low  4 = Medium  6 = High");
  disp_str(ROW + 7, COL,
     "Enter key click volume (0 - 6):");
  get_keys(ROW + 7, COL + 36, line);                 /* get chioce */
  sscanf(line, "%d", &y);                          /* ascii to int */
  if(y < 0 || y > 6) y = 4;                    /* keep it in bounds */
  wr_kyb(SETVOL);                        /* write set volume command */
  wr_kyb(y);                                     /* write volume data */
  break;

case F3:                                    /* toggle auto-repeat ? */
  if(auto_rep)                                  /* auto-repeat off ? */
  {
    wr_kyb(AREPON);                                /* autorepeat on */
    auto_rep = 0;                                   /* clear toggle */
  }
  else                                          /* auto-repeat is on */
  {
    wr_kyb(AREPOFF);                             /* autorepeat off */
    auto_rep = 1;                                    /* set toggle */
  }
  break;

case F4:                                    /* set auto-repeat rate ? */
  disp_str(ROW, COL,
   "The autorepeat rate has a range of 2 to 30 in 32 increments");
  disp_str(ROW + 1, COL,
     "The autorepeat rate is calculated as follows:");
  disp_str(ROW + 2, COL,
     "Rate = 1 / (.00417 * (2^Y) * (X + 8)");
  disp_str(ROW + 3, COL,
  "The delay, before autorepeat begins, has a range of");
  disp_str(ROW + 4, COL,
      ".25 seconds to 1 second in .25 second increments");
  disp_str(ROW + 5, COL,
```

```
    "The delay, before autorepeat begins, is calculated as follows:");
    disp_str(ROW + 6, COL, "delay = (D + 1) * .25");
    disp_str(ROW + 7, COL,
       "Enter repeat rate Y value (0 - 3):");
    disp_str(ROW + 8, COL,
       "Enter repeat rate X value (0 - 7):");
    disp_str(ROW + 9, COL,
       "Enter delay value D (0 - 3):");
    get_keys(ROW + 7, COL + 36, line);                  /* get input */
    sscanf(line, "%d", &y);                           /* ascii to int */
    if(y < 0 || y > 3) y = 2;                        /* keep in bounds */
    get_keys(ROW + 8, COL + 36, line);                  /* get input */
    sscanf(line, "%d", &x);                           /* ascii to int */
    if(x < 0 || x > 3) x = 2;                        /* keep in bounds */
    get_keys(ROW + 9, COL + 30, line);                  /* get input */
    sscanf(line, "%d", &d);                           /* ascii to int */
    if(d < 0 || d > 3) d = 2;                        /* keep in bounds */
    y <<= 3;                          /* shift into correct position */
    d <<= 5;                          /* shift into correct position */
    wr_kyb(SETAR);             /* write set auto-repeat rate command */
    wr_kyb(d | y | x);                     /* write auto-repeat data */
    break;

  case F5:                              /* read keyboard ID and state */
    intr_flag = int_off();                        /* CPU interrupts off */
    pass_thru(TRUE);  /* interface controller in pass-through mode */
    wr_kcd(KYBID);                      /* write keyboard ID command */
    while(!(inp(COMMAND) & 0x01))       /* wait until data available */
      ;
    y = inp(DATAREG);                             /* read version byte */
    while(!(inp(COMMAND) & 0x01))       /* wait until data available */
      ;
    x = inp(DATAREG);                               /* read mode byte */
    pass_thru(FALSE);             /* interface controller interprets */
    int_on(intr_flag);                            /* CPU interrupts on */
    if(x == 2)                             /* DIGITAL extended mode ? */
      sprintf(line,
        "Keyboard version #%d is in DIGITAL extended mode %d", y, x);
    else                                      /* industry-standard mode */
      sprintf(line,
       "Keyboard version #%d is in industry-standard mode %d", y, x);
    disp_str(ROW, COL, line);                          /* display data */
    break;

  case F6:                     /* reset keyboard to default values ? */
    wr_kyb(RESDEF);               /* write reset to default command */
    break;
```

```
        case F7:            /* show last pressed and last released keys ? */
          line[0] = 0;                                          /* null line */
          disp_str(ROW, 0, "Press F7 again to cancel");
          while(line[0] != 0 || line[1] != F7)          /* F7 terminates */
          {
            sprintf(line, "Last depressed: %02x", depressed);
            disp_str(ROW + 3, 0, line);
            sprintf(line, "Last  released: %02x", released);
            disp_str(ROW + 3, 40, line);
            chk_dt();                          /* display date and time ? */
            if(get_key(&line[0])) /* scan code */
              while(!get_key(&line[1])) /* character code */
                ;
          }
          break;

        case F10:
          return;
      }
      line[0] = get_fkey();                        /* get menu selection */
    }
  }
```

Chapter 9
Serial Communications

Overview

The 8250A universal asynchronous receiver/transmitters (UART) in the VAXmate workstation provide asynchronous communications for the communications port, the printer port, and the modem port.

The 8250A UART converts parallel data from the internal buses to serial data for transmission to external devices. The 8250A UART receives serial data and converts it to parallel data for the internal buses. The serial data has the format of a start bit; five to eight data bits; an optional parity bit; and 1, 1-1/2, or 2 stop bits.

The 8250A UART also has a programmable baud rate generator.

Additional Sources of Information

The following documents provide additional information on programming the 8250A UART serial communications devices.

- *Series 8000 Microprocessor Family Handbook* (National Semiconductor Corporation)
- *1984 Data Communications Products Handbook* (Western Digital Corporation)

8250A UART Registers

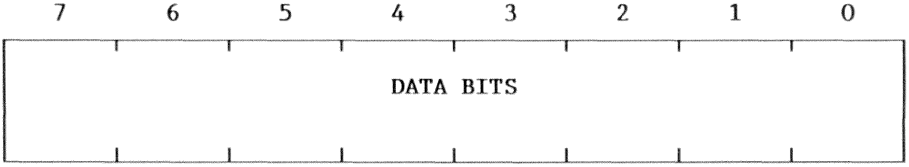
Table 9-1 lists the 8250A UART registers and their registers at each port.

Table 9-1 8250A UART Register Addresses

Register	Com1	Modem (Com2)	Printer
Receive buffer (Read)/Transmit holding register (Write) or Divisor latch (LSB) *	03F8H	02F8H	0CA0H
Interrupt enable or Divisor latch (MSB) *	03F9H	02F9H	0CA1H
Interrupt identification	03FAH	02FAH	0CA2H
Line control	03FBH	02FBH	0CA3H
Modem control	03FCH	02FCH	0CA4H
Line status	03FDH	02FDH	0CA5H
Modem status	03FEH	02FEH	0CA6H

* Bit 7 of the line control register controls access to the divisor latches. The line control register is described later in this chapter.

Receive Buffer Register/Transmitter Holding Register (03F8H/02F8H/0CA0H)



Bit	R/W	Description
7-0	R	Reading this register accesses the receive buffer.
	W	Writing this register accesses the transmitter holding register.
		Bit 0 (the least significant bit) is the first bit transmitted or received.
		When the line control register is programmed for word lengths of less than 8 bits, the unused bits are read as zeros.
		When bit 7 of the line control register is set to 1, reading or writing this register accesses the least significant byte of the divisor latch.

Interrupt Enable Register (03F9H/02F9H/0CA1H)

7	6	5	4	3	2	1	0
0	0	0	0	MODEM STATUS INTRPT	RECEIVE LINE STATUS INTRPT	THRE INTRPT	RECEIVE DATA AVAIL INTRPT

Bit	R/W	Description
7-4	R/W	Always 0
3	R/W	MODEM STATUS INTRPT - Modem Status Interrupt 0 = Modem status interrupt disabled 1 = Modem status interrupt enabled
2	R/W	RECEIVE LINE STATUS INTRPT - Receive Line Status Interrupt 0 = Receive line status interrupt disabled 1 = Receive line status interrupt enabled
1	R/W	THRE INTRPT - Transmit Holding Register Empty Interrupt 0 = Transmit holding register empty interrupt disabled 1 = Transmit holding register empty interrupt enabled
0	R/W	RECEIVE DATA AVAIL INTRPT - Receive Data Available Interrupt 0 = Receive data available interrupt disabled 1 = Receive data available interrupt enabled

Writing all 0s to this register disables the 8250A UART interrupt structure. If any of bits 3-0 are set, the 8250A UART interrupt structure is enabled. Only the functions with set bits can cause an interrupt.

Because the 8250A UART has only one interrupt output line, you must read the interrupt identification register to determine which function or functions caused the interrupt. Later in this chapter, the interrupt identification register description defines the interrupt conditions for each function.

> NOTE
> Each 8250A UART has a buffer between the interrupt output line and the peripheral interrupt controller input. This buffer is controlled by bit 3 of the modem control register. Writing a 1 to bit 3 of the modem control register enables the buffer and therefore, the 8250A UART interrupt output line. The modem control

register is described later in this chapter.

To use the 8250A UART in an interrupt-driven environment you must program the peripheral interrupt controller. For more information on the peripheral interrupt controller, see Chapter 3.

Interrupt Identification Register (03FAH/02FAH/0CA2H)

7	6	5	4	3	2	1	0
0	0	0	0	0	INTERRUPT IDENTIFICATION		INTRPT PENDING

Bit	R/W	Description
7-3	R	Always 0
2-1	R	INTERRUPT IDENTIFICATION These two bits identify the highest priority interrupt pending. Table 9-2 defines the meaning of the interrupt identification bits.
0	R	INTRPT PENDING - Interrupt Pending 0 = One or more interrupts pending 1 = No interrupts pending

Table 9-2 Interrupt Identification

Priority Level	Interrupt ID Bits 2 1	Interrupt Enable Register Bit	Interrupt Condition	Interrupt Reset
Highest	1 1	Receiver line status	Overrun, parity error, framing error, or break interrupt	Reading the line status register
Second	1 0	Receive data available	Assembling a complete word in the receive buffer	Reading the receive buffer register
Third	0 1	Transmit holding register empty	Transmit holding register empty	Writing the transmit holding register
Fourth	0 0	Modem status	Change in state of CTS, DSR, RI, or RLSD signals	Reading the modem status register

Line Control Register (03FBH/02FBH/0CA3H)

7	6	5	4	3	2	1	0
DLAB	BREAK CONTROL	STICK PARITY	PARITY SELECT	PARITY ENABLE	STOP BITS	WORD LENGTH	

Bit	R/W	Description
7	R/W	DLAB - Divisor latch access bit 0 = Access to the divisor latch is disabled 1 = Access to the divisor latch is enabled When access to the divisor latch is enabled, the least significant byte of the divisor latch is read or written through the receive buffer/transmit holding register, and the most significant byte of the divisor latch is read or written through the interrupt enable register. The divisor latch is described later in this chapter.
6	R/W	BREAK CONTROL 0 = Break disabled 1 = A space (logic 0) state forced on the serial output
5	R/W	STICK PARITY 0 = Stick parity disabled 1 = If parity is enabled (bit 3 equals 1), stick parity is enabled When stick parity is enabled, the parity bit is transmitted and received in the following manner: • If bit 4 equals 1, the parity bit is always transmitted and received as a 0. • If bit 4 equals 0, the parity bit is always transmitted and received as a 1.
4	R/W	PARITY SELECT 0 = Even parity (except for stick parity as described in bit 5) 1 = Odd parity (except for stick parity as described in bit 5)

Bit	R/W	Description (Line Control Register - cont.)
3	R/W	PARITY ENABLE 0 = Parity disabled 1 = Parity enabled
2	R/W	STOP BITS 0 = 1 stop bit 1 = 1-1/2 stop bits (5-bit word length) 2 stop bits (6, 7, or 8-bit word length) This bit sets the number of stop bits only for transmitted characters. The receiver uses only the first stop bit detected.
1-0	R/W	WORD LENGTH 00 = 5 data bits 01 = 6 data bits 10 = 7 data bits 11 = 8 data bits When reading the receive buffer, unused bits are read as 0.

Modem Control Register (03FCH/02FCH/0CA4H)

7	6	5	4	3	2	1	0
0	0	0	LOOP	OUT2	OUT1	RTS	DTR

Bit	R/W	Description
7-5	R/W	Always 0
4	R/W	LOOP 0 = Diagnostic loopback disabled 1 = Diagnostic loopback enabled (see the discussion that follows)
3	R/W	OUT2 0 = The buffer between the 8250A UART interrupt output and the peripheral interrupt controller input is disabled. 1 = The buffer between the 8250A UART interrupt output and the peripheral interrupt controller input is enabled. Each 8250A UART has a buffer between the interrupt output line and the peripheral interrupt controller input. This buffer is controlled by bit 3. Enabling the buffer enables the 8250A UART interrupt output line. To use the 8250A UART in an interrupt-driven environment you must program the peripheral interrupt controller. For more information on the peripheral interrupt controller, see Chapter 3.
2	R/W	OUT1 (not connected to anything)
1	R/W	RTS - Request to send 0 = RTS is a logic 0 at the external connector 1 = RTS is a logic 1 at the external connector
0	R/W	DTR - Data terminal ready 0 = DTR is a logic 0 at the external connector 1 = DTR is a logic 1 at the external connector

NOTE
See the section "Modem Control Programming Exceptions" in this chapter.

Diagnostic Loopback

When the diagnostic loopback is enabled, the following conditions exist:

- The serial output at the external connector is set to a space (logic 0) state.
- The serial input is internally disconnected.
- The output of the transmit shift register is internally connected to the input of the receive shift register.
- The four modem inputs (DSR, CTS, RI, and RLSD) are internally disconnected. The four modem outputs (DTR, RTS, OUT1, and OUT2) are connected to the four modem inputs as follows:
  - The DTR output is connected to DSR input
  - The RTS output is connected to the CTS input
  - The OUT1 output is connected to the RI input
  - The OUT2 output is connected to the RLSD input

When the diagnostic loopback is enabled, the receive and transmit interrupts continue to function normally. The modem control and the line status interrupts are also functional, but the source of the interrupt is changed as follows:

- The line status interrupt is activated by writing an appropriate value to one of the line status register bits 5-0. A set bit creates the interrupt condition.
- The modem status interrupt is activated by writing an appropriate value to one of the modem status register bits 3-0. A set bit creates the interrupt condition.

Line Status Register (03FDH/02FDH/0CA5H)

7	6	5	4	3	2	1	0
0	XMIT SHIFT REG EMPTY	XMIT HOLDING REG EMPTY	BREAK INTRPT	FRAMING ERROR	PARITY ERROR	OVERRUN ERROR	RECEIVE DATA READY

Bit	R/W	Description
7	R	Always 0
6	R	XMIT SHIFT REG EMPTY - Transmit Shift Register Empty 0 = Transmit shift register contains data being transmitted 1 = Transmit shift register is empty This bit is cleared by writing data to the transmit holding register.
5	R	XMIT HOLDING REG EMPTY - Transmit Holding Register Empty 0 = Transmit holding register is full. 1 = The 8250A UART is ready to accept a character for transmission. If the transmit holding register interrupt enable bit (interrupt enable register) is set, this condition creates an interrupt. This bit is cleared by writing data to the transmit holding register.
4	R/W	BREAK INTRPT - Break Interrupt 0 = Break interrupt is not active 1 = The serial input line at the connector has been held in a mark (logic 1) state for longer than a full character transmission, including start and stop bits. This bit is cleared by reading this register or writing the bit.
3	R/W	FRAMING ERROR 0 = No framing error 1 = The received character did not have a stop bit. This bit is cleared by reading this register or writing the bit.

Bit	R/W	Description (Line Status Register - cont.)
2	R/W	PARITY ERROR 0 = No parity error 1 = Received character did not have the correct parity. This bit is cleared by reading this register or writing the bit.
1	R/W	OVERRUN ERROR 0 = No overrun error 1 = The CPU did not read the data in the Receive Buffer register before the next character was received. Thus, the unread character was destroyed. This bit is cleared by reading this register or writing the bit.
0	R/W	RECEIVE DATA READY 0 = Receive data buffer is empty. 1 = A complete character has been received and assembled into the receive Buffer register. This bit is cleared by reading this register or writing the bit.

When any of bits 4-1 are set, a receiver line status interrupt is generated. When bit 0 is set, a receive data available interrupt is generated.

Modem Status Register (03FEH/02FEH/0CA6H)

7	6	5	4	3	2	1	0
RLSD	RI	DSR	CTS	DELTA RLSD	TRAIL EDGE OF RI	DELTA DSR	DELTA CTS

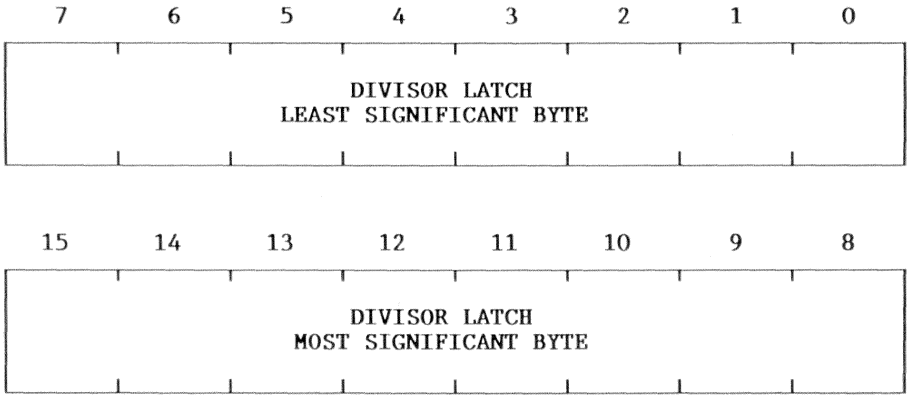
Bit	R/W	Description
7	R	RLSD - Received line signal detect 0 = RLSD at external connector is a logic 0 1 = RLSD at external connector is a logic 1
6	R	RI - Ring indicator 0 = RI at the external connector is a logic 0 1 = RI at the external connector is a logic 1
5	R	DSR - Data set ready 0 = DSR at the external connector is a logic 0 1 = DSR at the external connector is a logic 1
4	R	CTS - Clear to send 0 = CTS at the external connector is a logic 0 1 = CTS at the external connector is a logic 1
3	R	DELTA RLSD - Delta Receive Line Signal Detect 0 = RLSD has not changed state. 1 = Since the last time this register was read, the RLSD bit has changed state. Reading this register clears the bit.
2	R	TRAIL EDGE OF RI - Trailing Edge Of Ring Indicator 0 = The ring indicator has not changed state. 1 = Since the last time this register was read, the ring indicator at the external connector changed from a logic 0 to a logic 1. Reading this register clears the bit.

Bit	R/W	Description (Modem Status Register - cont.)
1	R	Delta DSR - Delta Data Set Ready 0 = DSR has not changed state. Reading the register clears the bit. 1 = Since the last time this register was read, the DSR signal has changed state. Reading this register clears the bit.
0	R	Delta CTS - Delta Clear to Send 0 = CTS has not changed state. 1 = Since the last time this register was read, the CTS signal has changed state. Reading this register clears the bit.

If any of bits 3-0 in the Modem Status register are set, a modem status interrupt is generated.

Bits 7-4 are the complement of the signal levels at the chip input pins. However, the RS-232 receiver buffer inverts these signals, so the bits reflect the true state of the lines at the external connector.

Divisor Latches



When bit 7 of the line control register (Divisor latch access bit) is equal to 1, the least significant byte of the divisor latch is read or written through the receive buffer/transmit holding register, and the most significant byte of the divisor latch is read or written through the interrupt enable register.

These two, 8-bit latches store a 16-bit divisor in the range 1 to 65,535. The output frequency of the baud rate generator is 1.8432 Mhz divided by the 16-bit divisor. These divisors must be loaded during initialization. The desired output frequency is 16 times the desired baud rate. Table 9-3 lists the divisor used for the standard baud rates. Table 9-3 was calculated using the following formula:

$$divisor = (1843200 / 16) / desired\_baud\_rate$$

Table 9-3 Baud Rate Table

Baud Rate	Divisor	Percentage of Error Between Desired and Actual Rate
50	2304	-
75	1536	-
110	1047	— 0.026
134.5	857	+0.058
150	768	-
200	576	-
300	384	-
600	192	-
1200	96	-
1800	64	-
2000	58	+0.69
2400	48	-
3600	32	-
4800	24	-
7200	16	-
9600	12	-
19200	6	-
38400	3	-

Modem Control Programming Exceptions

Speed Select and Speed Indicator control signals are not controlled by the 8250A UART. Instead, these signals are controlled by writing to a special purpose register.

The special purpose register is located at I/O address 0C80H.

> **NOTE**
> When changing the Speed Select or Split Baud Rate, maintain the integrity of the other bits in this register. Split baud rates are achieved by switching the output (RCLK H) between two sources, BD OUT CLK (baud out of the 8250A UART) and a 1200 baud counter.

Special Purpose Register (0C80H)

7	6	5	4	3	2	1	0
WRITE PROTECT	TRACK 0	INDEX	SPEED	DISABLE VIDEO	SPLIT BAUD	DISABLE COMM	SPEED SELECT

Bit	R/W	Description
7	R	Write protect 0 = Selected diskette drive is not write protected 1 = Selected diskette drive is write protected
6	R	Track 0 0 = Head of selected diskette drive is not at track 0 1 = Head of selected diskette drive is at track 0
5	R	Index 0 = Index hole not in position for selected diskette drive 1 = Index hole in position for selected diskette drive
4	R	Speed Indicator 0 = Modem control speed select asserted 1 = Modem control speed select not asserted
3	R/W	Disable Video 0 = Video controller disabled 1 = Video controller enabled
2	R/W	Split Baud Rates 0 = (Receive = Transmit = programmed) 1 = (Receive = 1200) (Transmit = programmed)
1	R/W	Disable Communications 0 = Integral communications ports connected to I/O address space 1 = Integral communications ports disconnected from I/O address space
0	R/W	Speed Select 0 = Speed select asserted 1 = Speed select not asserted

Communications Connector Signals

The communications connector, a 25-pin, male, D-subminiature, is located on the rear bezel of the VAXmate workstation. This connector is functionally compatible with RS-232-C and electrically compatible with RS-423, configured as DTE (Data Terminal Equipment). Table 9-4 lists the signals supported by this connector.

Table 9-4 Communications Connector Signals

Pin	Signal Name
1	Protective ground
2	Transmitted Data
3	Received Data
4	Request to Send
5	Clear to Send
6	Data Set Ready
7	Signal ground
8	Receive Line Signal Detect
9	—
10	—
11	Not used
12	Speed Indicator
13	—
14	—
15	—
16	—
17	—
18	—
19	—
20	Data Terminal Ready
21	—
22	Ring Indicator
23	Speed Select
24	—
25	—

Printer Connector Signals

The printer connector is a 6-pin MMJ, female, modified modular connector located on the rear bezel of the VAXmate workstation. Table 9-5 lists the signals this connector supports.

Table 9-5 Printer Connector Signals

Pin	Signal Name
1	Data Terminal Ready
2	Transmit Data
3	Transmit Common (Signal ground)
4	Receive Common (Signal ground)
5	Receive Data
6	Data Set Ready

Modem Connector Signals

The modem connectors are modular TELCO (telephone line) compatible connectors located on the optional modem board, protruding through the rear panel of the VAXmate workstation. The connectors use an 8-pin, keyed modular housing for an RC11C jack (or CA11 jack in Canada). Table 9-6 lists the signals for a modem connector. Table 9-7 lists the signals for a handset connector.

Table 9-6 Modem Telephone Line Connector Signals

Pin No.	Signal Name	Meaning
1	N.C.	No connection
2	N.C.	No connection
3	MIC	Not used
4	TIP	TELCO signal source
5	RING	TELCO signal return
6	MI	Not used
7	N.C.	No connection
8	N.C.	No connection

Table 9-7 Handset Connector Signals

Pin No.	Signal Name	Meaning
1	N.C.	No connection
2	N.C.	No connection
3	MIC	Not used
4	RING	TELCO signal return
5	TIP	TELCO signal source
6	MI	Not used
7	N.C.	No connection
8	N.C.	No connection

Programming Example

The examples in this chapter demonstrate:

- Initializing an 8250A UARTA serial communications device
- Handling interrupt-driven serial I/O
- Handling hardware and software handshaking protocols

CAUTION

Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

The example provides routines as described in the following list:

device_init	Establishes a known state for a serial port
device_open	Activates the serial port and interrupts
ser_out	Handles character transmissions
restart	Activates interrupt-driven serial transmissions
device_int	Coordinates all of the device interrupts
device_close	Deactivates the serial port and interrupts
put_buf	Puts a character into a ring buffer from the application
puts_buf	Puts a string of characters into a ring buffer from the application
get_buf	Gets a character from a ring buffer for the application
so	The serial example
int_com1	An interrupt vector entry point for the com1 serial port interrupt handler
int_modm	An interrupt vector entry point for the modem serial port interrupt handler
int_prnt	An interrupt vector entry point for the serial printer port interrupt handler

Program Description

Constant Value	Description
P_ENAB through *WORD5*	Define the line control register bit values.
RDRDY through *BREAK*	Define the line status register bit values.
ENA_MOD through *ENA_THE*	Define the interrupt enable register bit values.
DTR through *LOOP*	Define the modem control register bit values.
RLSD through *CTS*	Define the modem status register bit values.
HHS through *DEV_OFF*	Define the bit values used to maintain the driver status. The driver status is part of the structure type DEV_DAT.
CLK_RATE	Is divided by the desired baud rate to give the value for the baud rate divisor registers.
NO_PAR through *S_BIT2*	Clarify the logic of the function, device_init.

```
#include "kyb.h"
#include "rb.h"
#include "example.h"

/*************************************************************************/
/* define constant values used in example serial driver                  */
/*************************************************************************/

/* define line control register bit values */

#define P_ENAB  0x08                        /* parity enabled  */
#define P_EVEN  0x10                      /* EVEN parity select */
#define P_ODD   0x00                      /* ODD  parity select */
#define P_STIK  0x20                     /* enable stick parity */
#define WORD8   0x03                        /* 8-bit word size */
#define WORD7   0x02                        /* 7-bit word size */
#define WORD6   0x01                        /* 6-bit word size */
#define WORD5   0x00                        /* 5-bit word size */

/* define line status register bit values */

#define RDRDY   0x01                          /* received data ready */
#define THRE    0x20                  /* transmit holding reg empty */
#define ERRORS  0x0e                    /* overrun, parity, framing */
#define BREAK   0x10                            /* received break */

/* define interrupt enable register bit values */

#define ENA_MOD 0x08                          /* enable modem status */
#define ENA_REC 0x05                    /* recv line stat & rd rdy */
#define ENA_THE 0x02                    /* enable trans hold empty */
```

```
/* define modem control register bit values */

#define DTR     0x01                          /* data terminal ready */
#define RTS     0x02                             /* request to send */
#define OUT2    0x08                        /* out 2 interrupt ctrl */
#define LOOP    0x10                              /* loopback mode */


/* define modem status register bit values */

#define RLSD    0x80                      /* recv line signal detect */
#define RI      0x40                             /* ring indicator */
#define DSR     0x20                             /* data set ready */
#define CTS     0x10                              /* clear to send */

/* define driver status bit values */

#define HHS     0x80                       /* use hardware handshake */
#define F_XOFF  0x40                          /* x-off pending flag */
#define F_XON   0x20                          /* x-on  pending flag */
#define DEV_CLS 0x10                       /* device close request */
#define SWAIT   0x04                           /* sending stopped */
#define DEV_OFF 0x01                           /* driver off line */

/* define some general constants */

#define CLK_RATE  115200L                  /* 1843200 / 16 = 115200 */
#define X_OFF   0x13                       /* x-off (DC3) character */
#define X_ON    0x11                       /* x-on  (DC1) character */

/* program constants used to initialize the line control register */

#define NO_PAR  0                                       /* no parity */
#define EV_PAR  1                                     /* even parity */
#define OD_PAR  2                                      /* odd parity */
#define SC_PAR  3                                /* stick parity clear */
#define SS_PAR  4                                  /* stick parity set */
#define S_BIT2  0x04                                    /* 2 stop bits */
```

The structure type *SERIAL* declares the relationship of the 8250A UART registers in I/O space. The registers accessed by the members rtbl and iebh depends on the following:

- If bit 7 of the line control register is set, rtbl accesses the low byte of the baud rate divisor and iebh accesses the high byte of the baud rate divisor.
- If bit 7 of the line control register is clear, rtbl accesses the receive register when reading and the transmit register when writing, and iebh accesses interrupt enable register.

The structure type *DEV_DAT* stores the characteristics of the individual devices, the current status, and the pointers to the buffers.

```
/*************************************************************************/
/* define structures used in example serial driver                       */
/*************************************************************************/

typedef struct
{
  unsigned char rtbl;                    /* receive/transmit/baud low */
  unsigned char iebh;                    /* int enable reg/baud high */
  unsigned char int_ident;               /* int identification reg */
  unsigned char line_ctrl;               /* line control register */
  unsigned char modem_ctrl;              /* modem control register */
  unsigned char line_stat;               /* line status register */
  unsigned char modem_stat;              /* modem status register */
} SERIAL;

typedef struct dev_dat
{
  SERIAL        *base;                   /* base i/o address of device */
  unsigned int  pic;                     /* PIC number it belongs to */
  unsigned int  ir_bit;                  /* IR bit in PIC */
  unsigned int  baud;                    /* desired baud rate */
  unsigned int  word_siz;                /* desired word size */
  unsigned int  parity;                  /* even, odd, none, stick */
  unsigned int  stop_bits;               /* 1, 1 - 1/2, 2 */
  unsigned int  req_dsr;                 /* DSR required flag */
  RING_BUFF     *prbi;                   /* ptr to input ring buff */
  RING_BUFF     *prbo;                   /* ptr to output ring buff */
  unsigned char stat_drv;                /* state of driver */
} DEV_DAT;

void int_on();
int  int_off();
```

The function *device_init* establishes a known state for the 8250A UARTA serial device. Two items of interest are stick parity and stop bits. If stick parity is selected, the parity bit is set to the opposite state of the even/odd parity bit. If more than one stop bit is selected, the number of stop bits depends on the word size. If the word size is five data bits, there are 1 1/2 stop bits. All other word sizes generate 2 stop bits. These characteristics are a function of the device, not the code.

```
/*************************************************************************/
/* device_init() - establish a known state for a serial port             */
/*************************************************************************/

void device_init(pdd)                          /*  initialize serial port */

register DEV_DAT *pdd;                          /* pointer to device data */

{

unsigned int  tbaud;                              /* temp to hold results */
unsigned char tpar;                             /* temp to collect results */
register SERIAL *ps;                           /* pointer to SERIAL struct */

  tbaud = CLK_RATE / pdd->baud;          /* calculate baud rate divisor */
  switch(pdd->parity)
  {
    case    NO_PAR:                                          /* no parity */
      tpar = NO_PAR;
      break;

    case    EV_PAR:                                        /* even parity */
      tpar = P_ENAB | P_EVEN;
      break;

    case    OD_PAR:                                         /* odd parity */
      tpar = P_ENAB | P_ODD;
      break;

    case    SC_PAR:                               /* stick, parity bit clear */
      tpar = P_ENAB | P_STIK | P_EVEN;
      break;

    case    SS_PAR:                                 /* stick, parity bit set */
      tpar = P_ENAB | P_STIK | P_ODD;
      break;
  }
  switch(pdd->word_siz)                            /* desired word size ? */
```

```
  {
    case 8:
      tpar |= WORD8;                          /* 8 data bits ? */
      break;

    case 7:
      tpar |= WORD7;                          /* 7 data bits ? */
      break;

    case 6:
      tpar |= WORD6;                            /* 6 data bits */
      break;

    case 5:
      tpar |= WORD5;                          /* 5 data bits ? */
      break;
  }
  if(pdd->stop_bits) tpar |= S_BIT2;                /* set stop bits */
  ps = pdd->base;                                 /* get shorter pointer */
  outp(&ps->iebh, 0);                               /* interrupts off */
  outp(&ps->modem_ctrl, LOOP);                    /* put in loopback mode */
  outp(&ps->line_ctrl, 0x80);                   /* access baud rate divisor */
  outp(&ps->rtbl, tbaud);                      /* baud rate divisor lo byte */
  outp(&ps->iebh, tbaud >> 8);                 /* baud rate divisor hi byte */
  outp(&ps->line_ctrl, tpar);                     /* bits per char, parity */
  pdd->stat_drv |= DEV_OFF;                         /* transmit int off */
}
```

The function *device_open* prepares the driver to handle interrupts and the device to receive characters. Because the modem interrupt has the lowest priority, it is enabled only for hardware handshaking. In that instance, it notifies the driver to restart character transmissions.

Setting the modem control register bit OUT2 connects the 8250A UARTA interrupt output to the 8259A interrupt input. This bit must be set to operate a serial device in an interrupt-driven environment.

```
/*************************************************************************/
/* device_open() - activate serial port and interrupts                   */
/*************************************************************************/

void device_open(pdd)                                    /* open a device */

register DEV_DAT *pdd;                          /* pointer to device data */

{

register SERIAL *ps;                          /* pointer to SERIAL struct */


  ps = pdd->base;                                  /* get shorter pointer */
  inp(&ps->rtbl);                               /* empty receive data buffer */
  inp(&ps->line_stat);                             /* clear status flags */
  if(pdd->stat_drv & HHS)                         /* if hardware handshake */
  {
    if((inp(&ps->modem_stat) & (DSR | CTS)) == (DSR | CTS))
      pdd->stat_drv &= ~SWAIT;                                /* mark as ok */
    else pdd->stat_drv |= SWAIT;                            /* mark as not ok */
    outp(&ps->iebh, ENA_REC | ENA_MOD);          /* receive, line & modem */
  }
  else outp(&ps->iebh, ENA_REC);                  /* just receive and line */
  pdd->stat_drv &= ~DEV_OFF;                      /* driver state = online */
  imask(pdd->pic, pdd->ir_bit, ON);              /* clr the interrupt mask */
  outp(&ps->modem_ctrl, DTR|RTS|OUT2);           /* set modem control bits */
}
```

The function *ser_out* provides a single location for maintaining or restarting interrupt-driven transmissions. It also provides a method for software handshake characters to preempt normal data transmissions. To prevent continuous interrupts, when the output buffer is empty, the transmit interrupt is disabled.

```
/*************************************************************************/
/* ser_out() - transmit interrupt startup and maintenance                */
/*************************************************************************/

ser_out(pdd)

register DEV_DAT *pdd;                          /* pointer to device data */

{

register SERIAL *ps;                          /* pointer to SERIAL struct */
char c;                                         /* character to transmit */
int flag;

void device_close();

  ps = pdd->base;                                 /* get shorter pointer */
  if(inp(&ps->line_stat) & 0x20 &&
    (inp(&ps->modem_stat) & DSR ||                 /* if terminal ready  */
    !pdd->req_dsr))                               /* or dsr not required */
  {
    if(pdd->stat_drv & F_XOFF)                  /* pending x-off request ? */
    {
      pdd->stat_drv &= ~F_XOFF;                    /* clear pending flag */
      c = X_OFF;                                          /* send x-off */
      flag = 1;
    }
    else if(pdd->stat_drv & F_XON)              /* pending x-on request ? */
    {
     pdd->stat_drv &= ~F_XON;                      /* clear pending flag */
       c = X_ON;                                          /* send x-on */
      flag = 1;
    }
    else if(!(pdd->stat_drv & SWAIT))                    /* terminal ready */
    {
      flag = rb_out(pdd->prbo, &c);             /* request character to send */
      if(flag > -1) flag = 1;                       /* character to send ? */
      else
      {
        flag = 0;
        if(pdd->stat_drv & DEV_CLS)
```

```
        {
          device_close(pdd);
          return;
        }
      }
    }
    else flag = 0;
  }
  else flag = 0;
  if(flag)
  {
                                                    /* enable transmit int */
    outp(&ps->iebh, inp(&ps->iebh) | ENA_THE);
    outp(&ps->rtbl, c);                               /* output character */
  }
  else                                             /* disable transmit int */
    outp(&ps->iebh, inp(&ps->iebh) & ~ENA_THE);
}
```

The function *restart* tests the interrupt enable register to determine if interrupts are currently enabled. If they are not, it calls *ser_out* to restart interrupt-driven transmissions.

```
/*********************************************************************/
/* restart - attempt to restart  interrupt-driven serial transmissions */
/*********************************************************************/

void restart(pdd)                                  /* restart serial output */

register DEV_DAT *pdd;                            /* pointer to device data */

{

int flag;                                                    /* temporary */

  if(!(inp(&pdd->base->iebh) & ENA_THE))           /* need to restart ? */
  {
    flag = int_off();                             /* CPU interrupts off */
    ser_out(pdd);                                          /* restart */
    int_on(flag);                                  /* allow interrupts */
  }
}
```

The functions *com1_int*, *modem_int*, and *printer_int* are interrupt handlers for the serial devices. These interrupt handlers call a common interrupt handler, *device_int*.

```
/*********************************************************************/
/* com1_int() - interrupt handler for com1 serial device             */
/*********************************************************************/
void com1_int()
{
extern DEV_DAT devdat[];
void device_int();

  device_int(&devdat[0]);              /* call common interrupt handler */
}

/*********************************************************************/
/* modem_int() - interrupt handler for modem serial device           */
/*********************************************************************/
void modem_int()
{
extern DEV_DAT devdat[];
void device_int();

  device_int(&devdat[1]);              /* call common interrupt handler */
}

/*********************************************************************/
/* printer_int() - interrupt handler for printer serial device       */
/*********************************************************************/
void printer_int()
{
extern DEV_DAT devdat[];
void device_int();

  device_int(&devdat[3]);              /* call common interrupt handler */
}
```

The function *device_int* is the major function within the device driver. It coordinates interrupt processing, handshake protocol, and peripheral interrupt controllers. For any given serial device, this function processes all pending interrupts during a single CPU interrupt. This reduces the CPU interrupt processing overhead.

Device communication errors are noted by placing a question mark in the input stream.

The switch values are tested against the contents of the interrupt identification register.

```
/*************************************************************************/
/* device_int() - interrupt handler for serial device                    */
/*************************************************************************/

void device_int(pdd)                                  /* interrupt handler */

register DEV_DAT *pdd;                             /* pointer to device data */

{

register SERIAL *ps;                             /* pointer to SERIAL struct */
unsigned char r_val;                                /* register value read */

  ps = pdd->base;                                    /* get shorter pointer */
  while (!((r_val = inp(&ps->int_ident)) & 1))          /* while int pend */
  {
    switch(r_val)                                /* discover which interrupt */
    {
      case 6:                                          /* receive error int */
        inp(&ps->line_stat);                                /* clear error */
        if(inp(&ps->modem_stat) & DSR ||                /* if terminal ready */
          !pdd->req_dsr)                              /* or dsr not required */
        {
          rb_in(pdd->prbi, '?');                             /* show error */
        }
        break;


      case 4:                                      /* receive data ready int */
        r_val = inp(&ps->rtbl);                             /* read character */
        if(inp(&ps->modem_stat) & DSR ||                /* if terminal ready */
           !pdd->req_dsr)                             /* or dsr not required */
        {
          if((pdd->stat_drv & HHS) == 0)           /* software handshake ? */
          {
            switch(r_val & 0x7f)
```

```
            {
                case X_OFF:                          /* x-off character ? */
                  pdd->stat_drv |= SWAIT;
                  break;

                case X_ON:
                  pdd->stat_drv &= ~SWAIT;           /* x-on character ? */
                  restart(pdd);
                  break;

                default:
                  if(rb_in(pdd->prbi, r_val) < 1)    /* save character */
                  {                                  /* if buffer near full */
                    pdd->stat_drv |= F_XOFF;              /* stop input */
                    restart(pdd);                    /* restart if needed */
                  }
                  break;
            }
          }
          else if(rb_in(pdd->prbi, r_val) < 1)       /* save character */
            outp(&ps->modem_ctrl, DTR | OUT2);       /* no input please */
        }
        break;

      case 2:                                  /* xmit hold reg empty int */
        ser_out(pdd);                               /* transmit character */
        break;

      case 0:                                        /* modem status int */
        if(pdd->stat_drv & HHS)                  /* hardware handshake ? */
          if((inp(&ps->modem_stat) & (DSR | CTS)) == (DSR | CTS))
          {
            if(pdd->stat_drv & SWAIT)                /* send stopped ? */
            {
              pdd->stat_drv &= ~SWAIT;                    /* mark as ok */
              restart(pdd);
            }
          }
          else pdd->stat_drv |= SWAIT;               /* mark as not ok */
        break;
    }
  }
  eoi(pdd->pic);
}
```

The function *device_close* disables all interrupts related to the device indicated. It also puts the modem control lines in an off-line state.

```
/*********************************************************************/
/* device_close() - deactivate serial port interrupts                */
/*********************************************************************/

void device_close(pdd)                              /* close a device */

register DEV_DAT *pdd;                          /* pointer to device data */

{

  outp(&pdd->base->modem_ctrl, 0);                    /* device offline */
  outp(&pdd->base->iebh, 0);                        /* 8250A UART interrupts off
*/
  imask(pdd->pic, pdd->ir_bit, OFF);                 /* mask the interrupt */
  pdd->stat_drv |= DEV_OFF;                       /* driver state = offline */
}
```

The function *put_buf* loops until it has stored the indicated character in an output buffer. After storing the character, it ensures that the transmit interrupt is enabled. The companion function *puts_buf* processes a null terminated string by calling put_buf at each character in the string. The null terminator is not processed.

```
/*************************************************************************/
/* put_buf() - put character in output buffer                            */
/*************************************************************************/

void put_buf(pdd, c)                          /* put char in output buf */

register DEV_DAT *pdd;                        /* pointer to device data */
char c;                                        /* char to put in buffer */

{

int flag;                                                  /* temporary */
int r_val;

  for(flag = -1; flag < 0;)                      /* wait until success */
  {
    flag = rb_in(pdd->prbo, c);                   /* attempt to store */
    restart(pdd);
  }
}

/*************************************************************************/
/* puts_buf() - put string in to output buffer                           */
/*************************************************************************/

void puts_buf(pdd, pc)                       /* string into output buf  */

register DEV_DAT *pdd;                        /* pointer to device data */
char *pc;                                          /* pointer to string */

{

int flag;

  while(*pc)                                   /* while string not done */
    put_buf(pdd, *pc++);                        /* do another character */
}
```

The function *get_buf* attempts to retrieve a character from the input buffer. If no characters are available, it returns a -1 value. It also handles the input handshake protocol.

```
/*************************************************************************/
/* get_buf() - get character from input buffer                           */
/*************************************************************************/

int get_buf(pdd)                                  /* get char from input buf */

register DEV_DAT *pdd;                             /* pointer to device data */

{

char c;                                            /* temp to hold character */
int flag;

  flag = rb_out(pdd->prbi, &c);                   /* get char from input buff */
  if(flag < 0) return(flag);                       /* no characters available */
  if(flag == 0)                                     /* room to restart flow ? */
  {                                                    /* and receive stopped */
    if(!(pdd->stat_drv & HHS))                       /* software handshake ? */
    {
      pdd->stat_drv |= F_XON;                                /* set x-on flag */
      restart(pdd);
    }
    else outp(&pdd->base->modem_ctrl, DTR|RTS|OUT2);
  }
  flag = c;
  return(flag & 0xff);                                    /* return character */
}

/*************************************************************************/
/* reserve storage for variables used in example serial driver           */
/*************************************************************************/
#define BUF_SIZ 100                                        /* size of buffers */
#define HIWATER 75                                  /* buffer near full value */
#define LOWATER 25                                 /* buffer near empty value */

#define COM1 (SERIAL *)0x03f8                        /* base address of com1 */
#define MODEM (SERIAL *)0x02f8                      /* base address of modem */
#define PRINTER (SERIAL *)0x0CA0                   /* base address of printer */

RING_BUFF rb[6];                                   /* ring buff ctrl structs */
char buff[3][2][BUF_SIZ];                                  /* ring buffers */
DEV_DAT devdat[3] =                                  /* device data tables */
```

```
{
{ COM1,     0, 4, 9600, 8, NO_PAR, 0, 1, &rb[0], &rb[1], 0 },
{ MODEM,    0, 3, 1200, 8, NO_PAR, 0, 1, &rb[2], &rb[3], 0 },
{ PRINTER, 1, 3, 4800, 8, NO_PAR, 0, 0, &rb[4], &rb[5], 0 },
};
```

The function *so* drives the examples by transmitting or receiving serial data. It initializes the devices and data structures, executes the example and then closes the device.

```
/*************************************************************************/
/* so() - example application that uses serial driver                    */
/*************************************************************************/
so()
{

static MESSAGE mso[] =                                    /* opening menu */
{
    {  3, 25, "Serial Communications Example" },
    {  5, 24, "F1.  Select the COM1 port" },
    {  6, 24, "F2.  Select the Modem (COM2) port" },
    {  7, 24, "F3.  Select the Printer port" },
    {  8, 24, "F4.  Transmit sample data" },
    {  9, 24, "F5.  Receive data" },
    { 10, 24, "F10. Return to Main menu" },
    {  0,  0,  0 },
};

register DEV_DAT *pdd;                          /* pointer to device data */
int i;                                                  /* loop control */
int s;                                                  /* loop control */
int col;                                          /* display position */
int flag;                                       /* hold state of CPU IF */
char line[512];                                   /* to hold input line */
unsigned int intr_flag;                         /* to hold CPU IF state */

  for(i = 0; i < 2; i++)
  {
    pdd = &devdat[i];                             /* get pointer to data */
    init_rb(pdd->prbi, &buff[i][0][0], BUF_SIZ, HIWATER, LOWATER);
    init_rb(pdd->prbo, &buff[i][1][1], BUF_SIZ, HIWATER, LOWATER);
    if(pdd->pic == 0) s = 0x08;           /* base vector for master pic */
    else s = 0x70;                         /* base vector for slave pic */
    intr_flag = int_off();                      /* no interrupts allowed */
    iv_init(s + pdd->ir_bit);                  /* init interrupt vectors */
    device_init(pdd);                             /* init serial device */
    device_open(pdd);                               /* activate device */
    int_on(intr_flag);                             /* allow interrupts */
  }
  line[0] = 0;                                                /* null line */
  while(1)                                          /* forever until F10 */
  {
```

```
disp_menu(mso);                    /* display menu for serial example */
line[0] = get_fkey();                        /* get menu selection */
switch(line[0])                             /* which function key ? */
{
  case F1:                                      /* select com1 port */
    pdd = &devdat[0];                        /* get pointer to data */
    break;

  case F2:                                /* select modem (COM2) port */
    pdd = &devdat[1];                        /* get pointer to data */
    break;

  case F3:                                      /* select printer port */
    pdd = &devdat[2];                        /* get pointer to data */
    break;

  case F4:                                    /* transmit sample text */
    disp_str(14, 0, "Press F4 again to cancel");
    i = 0;
    while(line[0] != 0 || line[1] != F4)          /* F4 terminates */
    {
      sprintf(&line[0], "This is line %d\r\n", i++);
      puts_buf(pdd, line);
      disp_str(15, 0, line);
      chk_dt();                          /* display date and time ? */
      if(get_key(&line[0])) /* scan code */
        while(!get_key(&line[1])) /* character code */
          ;
    }
    break;

  case F5:                                    /* receive serial data */
    disp_str(14, 0, "Press F5 again to cancel");
    col = 0;
    while(line[0] != 0 || line[1] != F5)          /* F5 terminates */
    {
      i = get_buf(pdd);
      if(i > -1)
      {
        i &= 0x7f;
        disp_t(16, col++, i, 0x07);
        if(i == '\r' || i == '\n') col = 0;
        if(col == 80) col = 0;
      }
      chk_dt();                          /* display date and time ? */
      if(get_key(&line[0])) /* scan code */
        while(!get_key(&line[1])) /* character code */
```

```
            ;
        }
        break;

      case F10:
        for(i = 0; i < 3; i++)
        {
          pdd = &devdat[i];                    /* get pointer to data */
          device_close(pdd);                  /* close device when done */
          if(pdd->pic == 0) s = 0x08;     /* base vector for master pic */
          else s = 0x70;                   /* base vector for slave pic */
          intr_flag = int_off();              /* no interrupts allowed */
          iv_rest(s + pdd->ir_bit);        /* restore interrupt vectors */
          int_on(intr_flag);                        /* allow interrupts */
        }
        return;
    }
  }
}
```

Chapter 10
Mouse Information

Introduction

The VAXmate mouse (part number VSXXX) is a pointing device with three input switches. The mouse has two encoders, one for the X axis and one for the Y axis. The encoders have a resolution of 200 counts per inch. When moved on a flat surface, the mouse monitors the motion relative to its position at the beginning of the motion. Thus, the mouse maintains positional data in the form of incremental X/Y encoder counts. Figure 10-1 shows the mouse in relation to its X/Y axes.

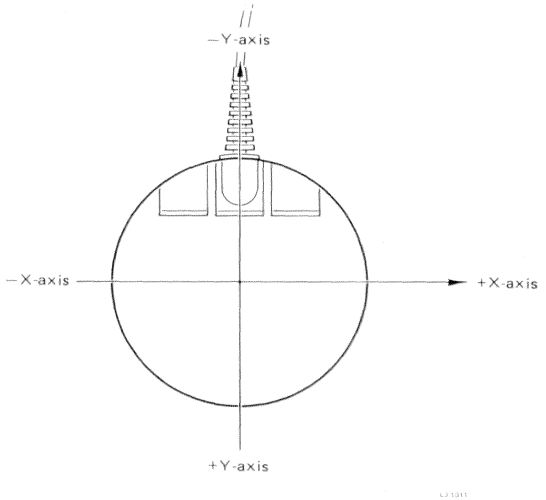


Figure 10-1 VAXmate Mouse (Part Number VSXXX)

Communication Requirements

The mouse communicates through an asynchronous serial interface at 4800 baud.
Data bytes have the following format:

- 1 start bit
- 8 data bits (least significant bit first)
- 1 parity bit (the mouse transmits odd parity, but ignores receive parity errors.)
- 1 stop bit

If a byte is sent to the mouse while the mouse is transmitting, the mouse aborts the transmission and processes the new command. If the mouse receives a byte between the characters of a multibyte report, the mouse is considered to be transmitting and aborts the report.

The VAXmate workstation communicates with the mouse through an asynchronous serial interface (Signetics SCN2661 Enhanced Programmable Communications Interface).

Additional Source of Information

The Signetics' document, *Microprocessor Data Manual 1986*, provides additional information on the SCN2661.

Mouse Commands

The Table 10-1 lists the mouse commands. The commands are issued by transmitting the appropriate command code.

Table 10-1 Mouse Command Summary

ASCII	HEX	Function
D	44H	Prompt Mode
R	52H	Incremental Stream Mode
P	50H	Request Mouse Position
T	54H	Invoke Self-test
Zx	5AH xx	Vendor Reserved function

Prompt Mode
Incremental Stream Mode

The mouse has two operating modes, prompt mode and incremental stream mode. In prompt mode, which is the powerup default, the mouse generates a report in response to a request mouse position command. In incremental stream mode, whenever the mouse moves it generates a report of that movement. It also reports a change in button position since the last report. No report is generated when the mouse is motionless and no buttons have been changed.

Request Mouse Position

The mouse responds to this command by sending a position report and switching to prompt mode.

Invoke Self-Test

The mouse responds to this command by executing a self-test and then sending a self-test report. Self-test leaves the mouse in the reset or powerup state. During the self-test, any data sent to the mouse is ignored until the last byte of the self-test report has been sent by the mouse. The 4-byte self-test report consists of a 2-byte identification code and a 2-byte status code.

Vendor Reserved Function

The vendor reserved function is a 2-byte command, the ASCII character 'Z' followed by any printable character. This command allows vendors to add special mouse functions. Normally, these functions are for quality control. The manufacturer determines these functions, which may include transmitting specialized reports. These commands may not include new modes. On completion of a vendor reserved function, the mouse must be restored to its previous state.

Mouse Reports

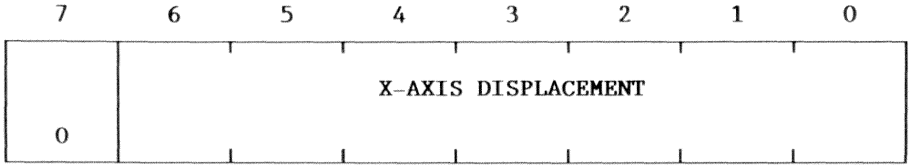
The mouse can transmit two reports, a 3-byte position report and a 4-byte self-test report.

Position Report - Byte 1

7	6	5	4	3	2	1	0
1	0	0	SIGN–X	SIGN–Y	LEFT BUTTON	MIDDLE BUTTON	RIGHT BUTTON

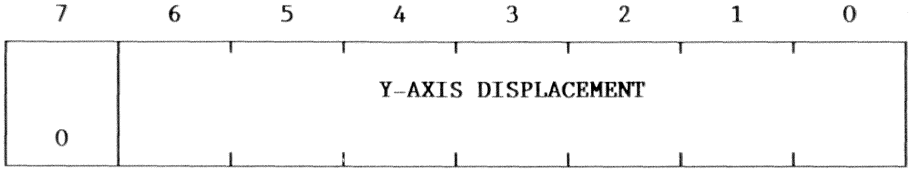
Bit	Description
7	Always 1
6-5	Always 0
4	SIGN-X (Sign bit for X-axis displacement) 0 = Negative X-axis displacement 1 = Positive X-axis displacement
3	SIGN-Y (Sign bit for Y-axis displacement) 0 = Negative Y-axis displacement 1 = Positive Y-axis displacement
2	LEFT BUTTON 0 = Switch open 1 = Switch closed
1	MIDDLE BUTTON 0 = Switch open 1 = Switch closed
0	RIGHT BUTTON 0 = Switch open 1 = Switch closed

Position Report - Byte 2



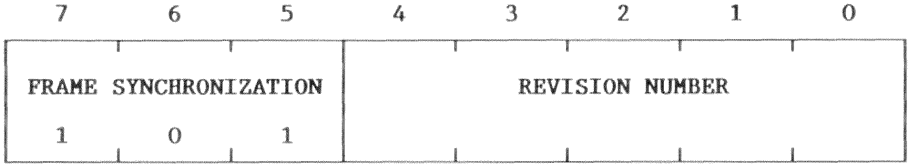
Bit	Description
7	Always 0
6-0	X-AXIS DISPLACEMENT The X-axis displacement is measured in encoder counts (200 per inch). The value returned in this byte is the distance moved since the last report. In prompt mode, if reports are not requested often enough, this value can overflow. If an overflow occurs, no indication is given. Bit 0 is the least significant bit.

Position Report - Byte 3



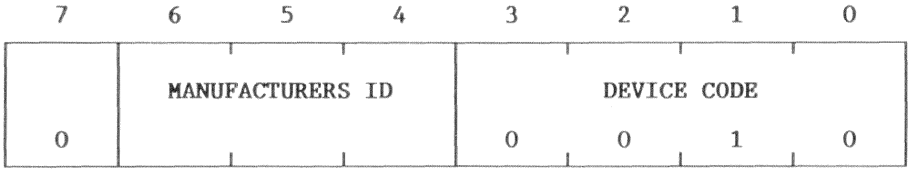
Bit	Description
7	Always 0
6-0	Y-AXIS DISPLACEMENT The Y-axis displacement is measured in encoder counts (200 per inch). The value returned in this byte is the distance moved since the last report. In prompt mode, if reports are not requested often enough, this value can overflow. If an overflow occurs, no indication is given. Bit 0 is the least significant bit.

Self-Test Report - Byte 1



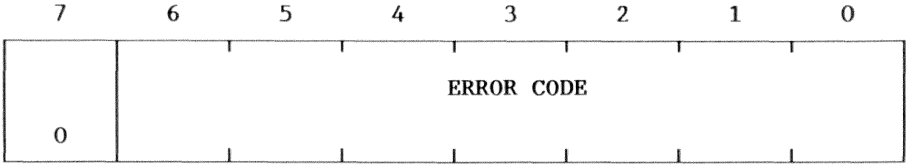
Bit	Description
7-5	FRAME SYNCHRONIZATION These bits are always 101. They provide a means of detecting the first byte of a self-test report.
4-0	REVISION NUMBER This is a hardware and software revision number for this design cycle.

Self-Test Report - Byte 2



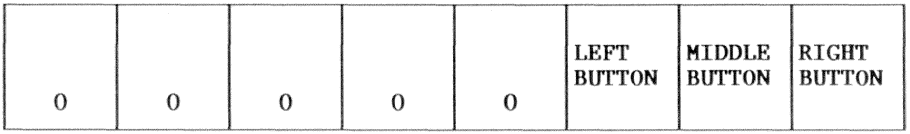
Bit	Description
7	Always 0
6-4	MANUFACTURERS ID
3-0	DEVICE CODE Always 0010

Self-Test Report - Byte 3



Bit	Description
7	Always 0
6-0	ERROR CODE 00H = No error 3EH = RAM or ROM checksum error 3DH = Button error

Self-Test Report - Byte 4



Bit	Description
7-3	Always 0
2	LEFT BUTTON 0 = Switch good 1 = Switch closed or failed
1	MIDDLE BUTTON 0 = Switch good 1 = Switch closed or failed
0	RIGHT BUTTON 0 = Switch good 1 = Switch closed or failed

Serial Interface

The serial interface is a SIGNETICS SCN2661 Enhanced Programmable Communications Interface. Table 10-2 lists the input/output (I/O) ports that access the serial interface registers.

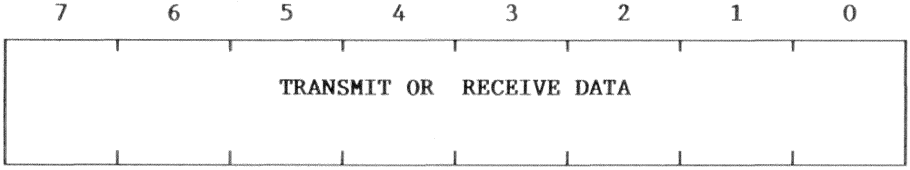
Table 10-2 Serial Interface Registers

Address	R/W	Register
0C40H	R W	Receive buffer Transmit holding register
0C41H	R W	Status register Syn1/Syn2/DLE registers *
0C42H	R/W	Mode register 1 and mode register 2 **
0C43H	R/W	Command register

* The Syn1, Syn2, and DLE registers are not used.

** Mode registers 1 and 2 are accessed at the same I/O address. Read mode register 1 and then read mode register 2, or write mode register 1 and then write mode register 2. Mode register 1 must be accessed to access mode register 2.

Transmit Holding Register and Receive Buffer (0C40H)



Bit	R/W	Description
7-0	R W	Accesses the receive data buffer Accesses the transmit holding register

Status Register (0C41H)

7	6	5	4	3	2	1	0
DATA SET READY 1	DATA CARRIER DETECT 1	FRAMING ERROR	OVERRUN	PARITY ERROR	DATA SET READY CHANGED	RxRDY	TxRDY

Bit	R/W	Description
7	R	DATA SET READY (always 1)
6	R	DATA CARRIER DETECT (always 1)
5	R	FRAMING ERROR 0 = Normal 1 = Framing error This bit is cleared by disabling the receiver, issuing the reset error command, or reading the status register.
4	R	OVERRUN 0 = Normal 1 = Overrun error This bit is cleared by disabling the receiver or issuing the reset error command.
3	R	PARITY ERROR 0 = Normal 1 = Parity error (if parity checking is enabled) This bit is cleared by disabling the receiver, issuing the reset error command, or receiving another character.
2	R	DATA SET READY CHANGED (always 0)
1	R	RxRDY - Receive Data Ready 0 = Receive buffer is empty 1 = Receive buffer contains data and an interrupt is pending This bit is cleared by reading the receive buffer or disabling the receiver (command register bit 2).
0	R	TxRDY - Transmit Holding Register Ready 0 = Transmit holding register busy 1 = Transmit holding register empty and an interrupt is pending This bit is cleared by writing the transmit holding register or disabling the transmitter (command register bit 0).

Mode Register 1 (0C42H)

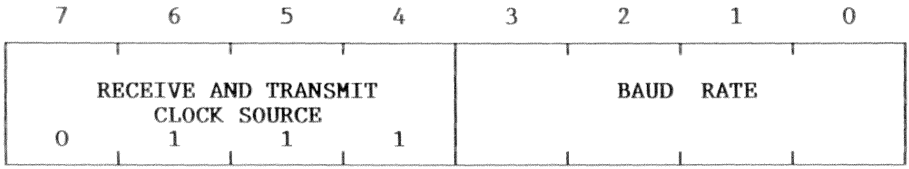
7	6	5	4	3	2	1	0
STOP BITS		PARITY TYPE	PARITY CONTROL	CHARACTER LENGTH		MODE AND BAUD RATE FACTOR	

Bit	R/W	Description
7-6	R/W	STOP BITS 00 = Invalid 01 = 1 stop bit 10 = 1-1/2 stop bits 11 = 2 stop bits
5	R/W	PARITY TYPE 0 = Odd parity 1 = Even parity
4	R/W	PARITY CONTROL 0 = Parity checking disabled 1 = Parity checking enabled
3-2	R/W	CHARACTER LENGTH 00 = 5 bits 01 = 6 bits 10 = 7 bits 11 = 8 bits
1-0	R/W	MODE AND BAUD RATE FACTOR 00 = Synchronous 1 X rate 01 = Asynchronous 1 X rate 10 = Asynchronous 16 X rate 11 = Asynchronous 64 X rate

Mode registers 1 and 2 are accessed at the same I/O address. Read mode register 1 and then read mode register 2, or write mode register 1 and then write mode register 2. Mode register 1 must be accessed to access mode register 2.

When programming mode register 1 on the VAXmate workstation, use a value 5EH.

Mode Register 2 (0C42H)



Bit	Description
7-4	RECEIVE AND TRANSMIT CLOCK SOURCE For the VAXmate workstation hardware, this value is fixed.
3-0	BAUD RATE See Table 10-3

Mode registers 1 and 2 are accessed at the same I/O address. Read mode register 1 and then read mode register 2, or write mode register 1 and then write mode register 2. Mode register 1 must be accessed to access mode register 2.

When programming mode register 2 on the VAXmate workstation, use a value 7CH.

Table 10-3 Baud Rate Table

Bits 3–0	Baud Rate	Bits 3–0	Baud Rate
0000	50	1000	1800
0001	75	1001	2000
0010	110	1010	2400
0011	134.5	1011	3600
0100	150	1100	4800
0101	300	1101	7200
0110	600	1110	9600
0111	1200	1111	19200

Command Register (0C43H)

7	6	5	4	3	2	1	0
OPERATING MODE		REQUEST TO SEND	RESET ERROR	SYNCH/ ASYNCH	RECEIVE CONTROL	DTR	XMIT CONTROL

Bit	R/W	Description
7-6	R/W	OPERATING MODE 00 = Normal operation 01 = Asynchronous (automatic echo mode) 10 = Local loop back 11 = Remote loop back
5	R/W	REQUEST TO SEND 0 = Force request to send output high (disables interrupt buffer) 1 = Force request to send output low (enables interrupt buffer) The 2661 EPCI has a buffer between the interrupt output line and the peripheral interrupt controller input. This buffer is controlled by bit 3. Enabling the buffer enables the 2661 EPCI interrupt output line.
4		RESET ERROR
	R	Always 0
	W	0 = No effect 1 = Reset error flags (parity, framing, overrun)
3	R/W	SYNCH/ASYNCH 0 = Normal 1 = Force break
2	R/W	RECEIVE CONTROL 0 = Disable receiver, receive interrupt, and status register bit 1 1 = Enable receiver, receive interrupt, and status register bit 0
1	R/W	DTR - Data Terminal Ready (output not connected) 0 = Force data terminal ready output high 1 = Force data terminal ready output low
0	R/W	XMIT CONTROL - Transmit Control 0 = Disable transmitter, transmit interrupt, and status register bit 0 1 = Enable transmitter, transmit interrupt, and status register bit 0

When programming the command register on the VAXmate workstation, use a

base value of 30H. In addition to the base value, bits 0 and 3 (transmit and receive control) must be applied as required.

Programming Example

The mouse programming example demonstrates:

- Communicating with the mouse
- Interpreting the motion
- Interpreting the buttons
- Scaling the mouse motion to the screen

The example provides routines as described in the following list:

mouse_init	Initializes the SCN2661 serial interface.
mouse_open	Prepares the serial interface for interrupt driven communications.
send_to_mouse	Sends commands to the mouse.
mouse_int	Is the serial interface interrupt handler.
mouse_close	Deactivates the serial interface.
mouse	Executes the example program.

> CAUTION
> Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

The include file *rb.h* defines the ring buffer structure used in the serial interface interrupt handler. The include files *kyb.h* and *example.h* support the example, but are not pertinent to the mouse section.

The constant value MOUSE_PORT defines the base address of the serial interface. The constant values MOUSE_PIC through MOUSE_HWI define the interrupt controller, interrupt input line, and the interrupt vector for the serial interface.

The constant values MMODE through CMND_REG define bit values for the serial interface registers. The constant values SELF_TEST through POS_REP define the mouse command bytes. Finally, the constant values TESTMASK through BUTTON_ERR define various values used in deciphering the mouse reports.

```
#include "rb.h"
#include "kyb.h"
#include "example.h"

/*************************************************************************/
/* define constant values used in example mouse driver                   */
/*************************************************************************/

#define MOUSE_PORT (MOUSE_UART *) 0x0C40        /* base address of mouse */
#define MOUSE_PIC 1                            /* PIC that handles mouse */
#define MOUSE_INT 4                            /* mouse int request line */
#define MOUSE_HWI 0x74                  /* harware int vector location */

/* SCN2661 mode register bit values */

#define MMODE   0x02                          /* Aynchronous 16X rate */
#define P_ENAB  0x10                                /* Enable parity */
#define P_EVEN  0x20                         /* Even parity select */
#define P_ODD   0x00                          /* Odd parity select */
#define WORD8   0x0C                           /* 8 bit characters */
#define WORD7   0x08                           /* 7 bit characters */
#define WORD6   0x04                           /* 6 bit characters */
#define WORD5   0x00                           /* 5 bit characters */
#define S_BIT1  0x40                              /* One stop bit */
#define S_BIT2  0xC0                             /* Two stop bits */
#define BD4800  0x0C                                /* 4800 baud */
#define CLKSPC  0x70                                /* 16X clock */

/* SCN2661 status register bit masks */

#define THRE 0x01                /* Transmit holding register is empty */
#define RDRDY 0x02                 /* Receive holding register is full */
#define ERRORS 0x38                /* Parity, overrun or framing error */
#define PARITYERR 0x08                                /* Parity error */
#define OVERRUNERR 0x10                              /* Overrun error */
#define FRAMINGERR 0x20                              /* Framing error */
```

```
/* SCN2661 command register bit values */

#define TxEN  0x01                              /* Enable transmit control */
#define DTR   0x02                            /* Disable data terminal ready */
#define RxEN  0x04                              /* Enable receive control */
#define BREAK 0x08                                        /* Disable break */
#define RESET 0x10                                 /* Enable reset status */
#define RTS   0x20                                   /* RTS (normally on) */
#define CMODE 0xC0                             /* Command mode (normally 0) */
#define CMND_REG RTS|RESET|RxEN  /* normal operation, RTS=1, rec enabled */

/* Define mouse commands */

#define SELF_TEST 'T'                                  /* self-test command */
#define P_MODE    'D'                                       /* prompt mode */
#define I_S_MODE  'R'                            /* incremental stream mode */
#define POS_REP   'P'                            /* request position report */

/* These values are used to check the mouse */

#define TESTMASK 0xe0                                /* mask any header byte */
#define HEADER_BYTE 0x80                           /* header byte indicator */
#define SELFTEST 0xa0                        /* self-test header byte mask */
#define POSREP 0x80                  /* position report header byte mask */
#define RIGHTBUTTON 0x01                             /* right button mask */
#define MIDDLEBUTTON 0x02                             /* left button mask */
#define LEFTBUTTON 0x04                             /* middle button mask */
#define XSIGN 0x10                                /* X-axis sign bit mask */
#define YSIGN 0x08                                /* Y-axis sign bit mask */
#define ROMRAM_ERR 0x3e           /* self-test byte # 2 error type mask */
#define BUTTON_ERR 0x3d           /* self-test byte # 2 error type mask */
```

```
/*************************************************************************/
/* define structures used in example mouse driver                        */
/*************************************************************************/

typedef struct
{
  unsigned char hr;              /* transmit/receive holding register */
  unsigned char status;                          /* status register */
  unsigned char mode;                              /* mode register */
  unsigned char command;                        /* command register */
} MOUSE_UART;

typedef struct mouse_dat
{
  MOUSE_UART     *base;                  /* base i/o address of device */
  RING_BUFF      *prbi;      /* pointer to input ring buffer structure */
  RING_BUFF      *prbo;     /* pointer to output ring buffer structure */
} MOUSE_DAT;

/*************************************************************************/
/* reserve storage for variables used in example mouse driver            */
/*************************************************************************/
#define BUF_SIZ 100                                    /* size of buffers */
#define HIWATER 75                               /* buffer near full value */
#define LOWATER 25                              /* buffer near empty value */

RING_BUFF rb[2];                                /* ring buff ctrl structs */
char buff[2][BUF_SIZ];                                  /* ring buffers */
unsigned char cmnd_reg;          /* value to write to command register */
MOUSE_DAT mouse_data = { MOUSE_PORT, &rb[0], &rb[1] };

int quietmouse = OFF;                                /* mouse state flag */
```

```
/*************************************************************************/
/* mouse_init() - establish a known state for the mouse and SCN2661      */
/*************************************************************************/

void mouse_init()                          /* initialize mouse and port */
{

register MOUSE_DAT *pdd;                     /* pointer to mouse data */
register MOUSE_UART *ps;                   /* pointer to mouse struct */

  pdd = &mouse_data;                         /* point to driver data */
  ps = pdd->base;                             /* assign base address */

/* initialize ring buffer structures */
  init_rb(pdd->prbi, &buff[0][0], BUF_SIZ, HIWATER, LOWATER);
  init_rb(pdd->prbo, &buff[1][0], BUF_SIZ, HIWATER, LOWATER);
/*  async 16x, enable parity, odd parity, eightbit data, one stop bit  */
  outp(&ps->mode, MMODE | P_ENAB | P_ODD | WORD8 | S_BIT1);
  outp(&ps->mode, BD4800 | CLKSPC);           /*  16x clock, 4800 baud */
  cmnd_reg = CMND_REG;                      /* init to normal contents */
  outp(&ps->command, cmnd_reg);               /* reset status errors */
}
```

NOTE
During the initialization process, it is possible to transmit or receive garbage characters. The mouse initialization function does not account for this possibility.

```
/*************************************************************************/
/* mouse_open() - activate mouse interrupts                              */
/*************************************************************************/

void mouse_open()                                  /* open the mouse */

{
  imask(MOUSE_PIC, MOUSE_INT, ON);          /* enable interrupt input */
}
```

```
/*********************************************************************/
/*  send_to_mouse() - write data to mouse                            */
/*********************************************************************/

send_to_mouse(c)

unsigned char c;                              /* byte value to transmit */

{

register MOUSE_DAT *pdd;                       /* pointer to mouse data */
register MOUSE_UART *ps;                     /* pointer to mouse struct */
int intr_flg;                                  /* hold state of CPU IF */

  pdd = &mouse_data;                           /* point to driver data */
  ps = pdd->base;                               /* assign base address */
  while(rb_in(pdd->prbo, c) < 0 )          /* wait until stored in buffer */
    ;
  intr_flg = int_off();                        /* disable CPU interrupt */
  cmnd_reg |= TxEN;                            /* to enable transmitter */
  outp(&ps->command, cmnd_reg);                  /* enable transmitter */
  int_on(intr_flg);                            /* enable CPU interrupt */
}
```

```
/*************************************************************************/
/* mouse_int() - interrupt handler for mouse serial port                 */
/*************************************************************************/

void mouse_int()                                    /* interrupt handler */

{
register MOUSE_DAT *pdd;                          /* pointer to mouse data */
register MOUSE_UART *ps;                    /* pointer to MOUSE_UART struct */
unsigned char c;
unsigned char s;

  pdd = &mouse_data;
  ps = pdd->base;                                   /* assign base address */

  s = inp(&ps->status);                             /* read status of port */

  if(s & (PARITYERR | FRAMINGERR))                  /* garbage character ? */
    s = inp(&ps->hr);                             /* read garbage character */
  else if(s & RDRDY)                          /* is there anything to read ? */
  {                                          /* read and store in ring buffer */
    if(rb_in(pdd->prbi, inp(&ps->hr)) < 1 )    /* buffer getting full ? */
    {
      send_to_mouse(P_MODE);                    /* put mouse in prompt mode */
      quietmouse = ON;
    }
  }
  if(s & THRE)                                      /* ready to transmit ? */
  {
    if(rb_out(pdd->prbo, &c) < 0)          /* any characters to transmit ? */
      cmnd_reg &=   TxEN;                            /* disable transmitter */
    else outp(&ps->hr, c);                           /* write the character */
  }
  outp(&ps->command, cmnd_reg);                     /* reset status errors */
  eoi(MOUSE_PIC);                      /* send EOI to interrupt controller */
}
```

NOTE

This routine could check for overrun errors, but it does not. Because each mouse report has a fixed byte count, missing characters are detected in the record collection part of the mouse() function.

```
/*************************************************************************/
/* mouse_close() - deactivate mouse port interrupts                      */
/*************************************************************************/

void mouse_close()                              /* deactivate the mouse */

{

register MOUSE_DAT *pdd;                          /* pointer to mouse data */
register MOUSE_UART *ps;                  /* pointer to MOUSE_UART struct */

  pdd = &mouse_data;                               /* point to driver data */
  send_to_mouse(P_MODE);                       /* put mouse in prompt mode */
  while(pdd->prbo->count)                   /* wait until ring buffer empty */
    ;
  cmnd_reg = CMND_REG &   RxEN;       /* disable receiver and transmitter */
  ps = pdd->base;                                    /* assign base address */
  outp(&ps->command, cmnd_reg);                        /* write new command */
  imask(MOUSE_PIC, MOUSE_INT, OFF);             /* disable interrupt input */
}
```

```
/*************************************************************************/
/* example application that uses mouse driver                            */
/*************************************************************************/

mouse()
{

static MESSAGE mmouse[] =                                   /* mouse menu */
{
  {  3, 34, "Mouse Example" },
  {  5, 18, "Move the mouse to see X and Y displacements." },
  {  6, 16, "Move cursor to a box and select with left button." },
  {  8, 24, "[ ] End Mouse Example" },
  {  9, 24, "[ ] Increase X Scale" },
  { 10, 24, "[ ] Decrease X Scale" },
  { 11, 24, "[ ] Increase Y Scale" },
  { 12, 24, "[ ] Decrease Y Scale" },
  { 14, 24, "Left button status:   Up  " },
  { 15, 24, "Middle button status: Up  " },
  { 16, 24, "Right button status:  Up  " },
  { 18, 24, "X encoder counts:     0" },
  { 19, 24, "Y encoder counts:     0" },
  { 20, 24, "X Scale:  12" },
  { 21, 24, "Y Scale:  52" },
  {  0,  0,  0 },
};

register MOUSE_DAT *pdd;                        /* pointer to mouse data */
register MOUSE_UART *ps;

unsigned char i_buff[80];                                 /* input buffer */
char o_buff[80];                                         /* output buffer */
char kb;
unsigned char *pb;                             /* pointer to input buffer */
int row = 0;                             /* row position of cursor or text */
int tmp;                                          /* temporary variable */
int pos_rep = 0;                            /* mouse position report flag */
int end_me = FALSE;                             /* end mouse example flag */
int left_button = FALSE;                          /* state of left button */
int buff_state;                                /* input buffer state flag */
int x_abs = 40;                  /* absolute position of cursor in X-axis */
int y_abs = 12;                  /* absolute position of cursor in Y-axis */
int x_cnts = 0;                     /* accumulated X-axis encoder counts */
int y_cnts = 0;                     /* accumulated Y-axis encoder counts */
int x_scale = 12;                          /* encoder counts per column */
int y_scale = 52;                             /* encoder counts per row */
int moved = 0;           /* flag to indicate that mouse reported motion */
```

```
int intr_flg;                                   /* hold state of CPU IF */
int cnt_req;                              /* byte count required for report */

extern int time_flag;                           /* defined in RTC example */

  pdd = &mouse_data;                              /* get pointer to data */
  ps = pdd->base;                                 /* assign base address */
  intr_flg = int_off();                          /* disable CPU interrupt */
  iv_init(MOUSE_HWI);                           /* init interrupt vectors */
  mouse_init();                                              /* init mouse */
  int_on(intr_flg);                             /* enable CPU interrupt */
  mouse_open();                                        /* activate mouse */
  send_to_mouse(P_MODE);                 /* ensure mouse is in prompt mode */
  while(rb_out(pdd->prbi, &i_buff[0]) >= 0)  /* while buffer not empty */
    ;                                                    /* dump characters */
  send_to_mouse(SELF_TEST);                    /* issue self-test command */
  intr_flg = int_off();                          /* disable CPU interrupt */
  time_flag = 0;                                 /* reset RTC second flag */
  int_on(intr_flg);                             /* enable CPU interrupt */
  for(tmp = 0; time_flag < 3 && tmp < 4; )
  {
    if(rb_out(pdd->prbi, &i_buff[tmp]) >= 0)  /* try to read character */
    {
      if((i_buff[tmp] & TESTMASK) == SELFTEST)/* self-test header byte */
        tmp = 1;                                  /* first byte of report */
      else if(tmp) tmp++;                      /* additional report bytes */
    }
  }
  tmp = 0;
  if(time_flag >= 3)                          /* check for time-out error */
  {
     strcpy(&o_buff[0], "Mouse time-out error");       /* error message */
     tmp = 1;                                       /* set error indicator */
  }
  else if(i_buff[2] == 0x3e)             /* check for mouse ROM/RAM error */
  {
     strcpy(&o_buff[0], " Mouse ROM/RAM error");       /* error message */
     tmp = 1;                                       /* set error indicator */
  }
  else if(i_buff[2] == 0x3d)             /* check for mouse button errors */
  {
     strcpy(&o_buff[0], " Mouse button error");        /* error message */
     tmp = 1;                                       /* set error indicator */
  }
  if(tmp)                                         /* test error indicator */
  {
    clear_vid_mem();
```

```
  disp_str(12, 35, &o_buff[0]);                    /* show error message */
  disp_str(13, 35, "Press F10 to continue");       /* show help message */
  while(1) if(get_key(&kb) == 1 && kb == F10) break;
}
else
{
  disp_menu(mmouse);                          /* display the mouse menu */
  cursor_on(y_abs, x_abs);                      /* make cursor visible */
  send_to_mouse(I_S_MODE);          /* reset to incremental stream mode */
  cnt_req = -1;                /* set byte count required to below zero */
  pos_rep = FALSE;                            /* no position report yet */
  pb = &i_buff[0];               /* initialize pointer to input buffer */
  while(end_me == FALSE)
  {
    chk_dt();                           /* check date and time for update */
    buff_state = rb_out(pdd->prbi, pb);          /* try to read mouse */
    if(buff_state >= 0)                /* did the mouse send anything ? */
    {
      if(*pb & HEADER_BYTE)                      /* is it a header byte ? */
      {
        i_buff[0] = *pb;                 /* move to beginning of buffer */
        pb = &i_buff[1];          /* reset to next byte in input buffer */
        if((i_buff[0] & TESTMASK) == POSREP) /* discover header type */
        {
          cnt_req = 2;                      /* remaining count required */
          pos_rep = TRUE;     /* have header byte for position report */
        }
        else                                /* anthing else is an error */
        {
          cnt_req = -1;       /* set byte count required to below zero */
          pos_rep = FALSE;                        /* no position report */
        }
      }
      else if(pos_rep)  /* if received a position report header byte */
      {
        if(++pb > &i_buff[10])              /* increment buffer pointer */
        {     /* if pointer test is true, unexpected error condition */
          pb = &i_buff[0];                      /* reset buffer pointer */
          pos_rep = FALSE;                /* cannot be a position report */
          cnt_req = -1;       /* set byte count required to below zero */
        }
        if(--cnt_req == 0)                             /* end of report ? */
        {
          /* get position increments */

          moved = 0;                          /* clear mouse motion flag */
          tmp = i_buff[1];                      /* get X-axis increment */
```

```
            if(!(i_buff[0] & XSIGN)) tmp = -tmp;      /* check sign bit */
            x_cnts += tmp;           /* accumulate X-axis encoder counts */
            sprintf(o_buff, "%4d", tmp);          /* convert to a string */
            disp_str(18, 42, o_buff);           /* show X-axis increment */
            tmp = x_cnts / x_scale;           /* enough to show motion ? */
            if(tmp)
            {
              x_cnts -= x_scale * tmp;           /* remove scaled counts */
              x_abs += tmp;                    /* add to absolute position */
              if(x_abs < 0) x_abs = 0;           /* no off-screen motion */
              if(x_abs > 79) x_abs = 79;         /* no off-screen motion */
              moved = 1;           /* set flag to update cursor position */
            }
            tmp = i_buff[2];                       /* get Y-axis increment */
            if(!(i_buff[0] & YSIGN)) tmp = -tmp;      /* check sign bit */
/* y-axis encoder counts are accumulated negatively to invert motion */
            y_cnts -= tmp;           /* accumulate Y-axis encoder counts */
            sprintf(o_buff, "%4d", tmp);          /* convert to a string */
            disp_str(19, 42, o_buff);           /* show Y-axis increment */
            tmp = y_cnts / y_scale;           /* enough to show motion ? */
            if(tmp)
            {
              y_cnts -= y_scale * tmp;           /* remove scaled counts */
              y_abs += tmp;                    /* add to absolute position */
              if(y_abs < 0) y_abs = 0;           /* no off-screen motion */
              if(y_abs > 24) y_abs = 24;         /* no off-screen motion */
              moved = 1;           /* set flag to update cursor position */
            }
            if(moved) mv_cursor(y_abs, x_abs);          /* update cursor */

            /* display state of mouse buttons */

            for(tmp = LEFTBUTTON, row = 14; row < 17; row++, tmp >>= 1)
            {
              if(i_buff[0] & tmp) disp_str(row, 46, "Down");
              else disp_str(row, 46, "Up  ");
            }
            if(i_buff[0] & LEFTBUTTON)      /* test for valid selection */
            {
              if(!left_button) /* must release button from last select */
              {
                left_button = TRUE;              /* left button pressed */
                if(x_abs == 25)
                {
                  switch(y_abs)
                  {
                    case 8:                          /* end mouse example */
```

```
                    end_me = TRUE;                          /* set to true */
                    break;

                  case 9:                                   /* increase X scale */
                    if(x_scale < 1000)                       /* arbitrary value */
                      x_scale += 2;                       /* arbitrary increment */
                    break;

                  case 10:                                  /* decrease X scale */
                    if(x_scale > 2)                           /* cannot be zero */
                      x_scale -= 2;                       /* arbitrary decrement */
                    break;

                  case 11:                                  /* increase Y scale */
                    if(y_scale < 1000)                       /* arbitrary value */
                      y_scale += 2;                       /* arbitrary increment */
                    break;

                  case 12:                                  /* decrease Y scale */
                    if(y_scale > 2)                           /* cannot be zero */
                      y_scale -= 2;                       /* arbitrary decrement */
                    break;
                }
                sprintf(o_buff, "%3d", x_scale);                /* convert */
                disp_str(20, 33, o_buff);               /* show new X-scale */
                sprintf(o_buff, "%3d", y_scale);                /* convert */
                disp_str(21, 33, o_buff);               /* show new Y-scale */
              }
            }
          }
          else left_button = FALSE;             /* reset left button state */
          pos_rep = FALSE;                            /* no position report */
        }
      }
    }
/* If mouse disabled because the ring buffer was full, turn it back on */
    if(quietmouse && buff_state == 0) send_to_mouse(I_S_MODE);
  }
 }
 mouse_close();                                          /* close the mouse */
 intr_flg = int_off();                                /* no interrupts allowed */
 iv_rest(MOUSE_HWI);                                    /* restore old vectors */
 int_on(intr_flg);                                       /* allow interrupts */
}
```

Chapter 11
Diskette Drive Controller

Introduction

The diskette drive controller interfaces the VAXmate system bus and the diskette drives. The diskette drive controller supports the following drives and media:

Drive Type	Media
5¼ Inch - High capacity	1.2 Megabyte - 80 Track - High capacity
	800 Kbyte - 80 Track - Standard
	360 Kbyte - 40 Track - Standard (with double stepping)

The diskette drive controller operates in either DMA or non-DMA mode. In DMA mode, the processor initializes the DMA controller and issues the transfer command to the diskette controller. The diskette controller and the DMA controller transfer the data unattended. In non-DMA mode, the diskette controller generates interrupts to the processor each time the controller transfers a data byte.

Diskette Drive Controller Registers

The diskette drive controller has five 8-bit registers that are accessed through four port addresses. Table 11-1 lists the registers.

Table 11-1: Diskette Drive Controller Registers

Address	R/W	Register
03F2H	W	Control register
03F4H	R	Main status register
03F5H	R/W	Data register
03F6H	W	Transfer rate register
03F6H	R	Change register

Control Register (03F2H)

7	6	5	4	3	2	1	0
0	0	MOTOR B	MOTOR A	DMA ENABLE	RESET	0	DRIVE SELECT

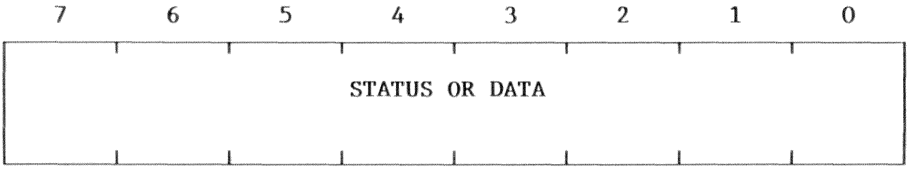
Bit	R/W	Description
7-6	W	Always 0
5	W	MOTOR B 0 = Drive B motor off and disable bit 0 1 = Drive B motor on and enable bit 0
4	W	MOTOR A 0 = Drive A motor off and disable bit 0 1 = Drive A motor on and enable bit 0
3	W	DMA ENABLE 0 = Disable the diskette drive controllers DMA request, DMA acknowledge, and interrupt request 1 = Enable the diskette drive controllers DMA request, DMA acknowledge, and interrupt request
2	W	RESET 0 = Reset the diskette drive controller 1 = Enable the diskette drive controller
1	W	Always 0
0	W	DRIVE SELECT 0 = Select Drive A 1 = Select Drive B This bit is enabled or disabled by bits 5-4.

Main Status Register (03F4H)

7	6	5	4	3	2	1	0
REQUEST FOR MASTER	DATA I/O DIR	NON-DMA MODE	CONTROL BUSY	DRIVE 3 BUSY	DRIVE 2 BUSY	DRIVE 1 BUSY	DRIVE 0 BUSY

Bit	R/W	Description
7	R	REQUEST FOR MASTER 0 = Data register not ready 1 = Data register is ready to be read or written by processor
6	R	DATA I/O DIR - Data I/O Direction 0 = Transfer data from processor to data register 1 = Transfer data from data register to processor
5	R	NON-DMA MODE 0 = Result phase (execution phase ended) 1 = Execution phase
4	R	CONTROL BUSY 0 = Controller ready to accept new command 1 = Controller processing a read or write command
3	R	DRIVE 3 BUSY 0 = Drive 3 not seeking 1 = Drive 3 seeking new track
2	R	DRIVE 2 BUSY 0 = Drive 2 not seeking 1 = Drive 2 seeking new track
1	R	DRIVE 1 BUSY 0 = Drive 1 not seeking 1 = Drive 1 seeking new track
0	R	DRIVE 0 BUSY 0 = Drive 0 not seeking 1 = Drive 0 seeking new track

Data Register (03F5H)



Bit	R/W	Description
7-0	R/W	Status or data

This register accesses several internal diskette drive controller registers. The internal register accessed depends on the state of the diskette drive controller. The internal registers and the diskette drive controller states are discussed later in this chapter in the section **Diskette Drive Controller Programming**.

Data Transfer Rate Register (03F6H)

7	6	5	4	3	2	1	0
0	0	0	0	0	0	TRANSFER RATE	

Bit	R/W	Description
7-2	W	Always 0
1-0	W	TRANSFER RATE 00 = 500 KBits per second 01 = 250 KBits per second * 10 = 250 KBits per second 11 = Not used (selects 250 KBits per second)

* The industry-standard transfer rate for the bit values (01) is 300 KBits per second.

On power-up, this register defaults to 250 KBits per second.

Change Register (03F6H)

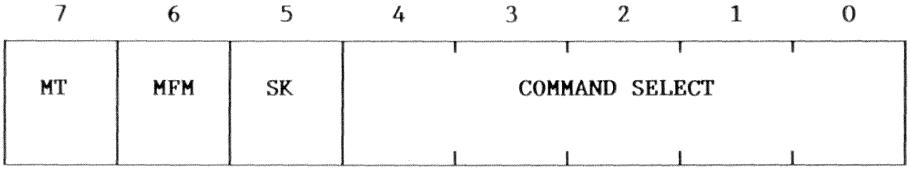
7	6	5	4	3	2	1	0
CHANGE STATUS	0	0	0	0	0	0	0

Bit	R/W	Description
7	R	CHANGE STATUS 0 = Since the last time this register was read, the diskette in the selected drive has not been removed. 1 = Since the last time this register was read, the diskette in the selected drive was removed.
6-0	R	Always 0

Diskette Drive Controller Internal Registers

The diskette drive controller has several internal registers that are read or written through the data register as a series of command or status bytes. The usage of these registers is dependent on the command. Tables 11-3 through 11-17 define the command specific usage.

Internal Register - Command



Bits	R/W	Description
7	W	MT - Multi-track (Must be 0 for some commands) 0 = Disable multi-track 1 = Enable multi-track (Accessing both sides of the diskette automatically)
6	W	MFM - Modified Frequency Modulation 0 = Use FM (Frequency Modulation) for reading and writing the diskette 1 = Use MFM for reading and writing the diskette
5	W	SK - Skip (Must be 0 for some commands) 0 = Do not skip sectors containing a DELETED DATA ADDRESS MARK 1 = Skip sectors containing a DELETED DATA ADDRESS MARK
4-0	W	COMMAND SELECT 00010 = Read Track 00011 = Specify 00100 = Sense Drive Status 00101 = Write Data 00110 = Read Data 00111 = Recalibrate 01000 = Sense Interrupt Status 01001 = Write Deleted Data 01010 = Read ID 01100 = Read Deleted Data 01101 = Format Track 01111 = Seek 10001 = Scan Equal 11001 = Scan Low or Equal 11101 = Scan High or Equal

Internal Register - Head/Unit Select

7	6	5	4	3	2	1	0
0	0	0	0	0	HEAD SELECT	UNIT SELECT	

Bit	R/W	Description
7-3	W	Always 0
2	W	HEAD SELECT 0 = Select head on side 0 1 = Select head on side 1
1-0	W	UNIT SELECT 00 = Select drive 0 01 = Select drive 1 10 = Select drive 2 11 = Select drive 3 Because the outputs are not connected, these bits are ineffective. Use bits 5, 4, and 0 of the control register to select the drive.

Internal Register - Status Register 0

7	6	5	4	3	2	1	0
INTERRUPT CODE		SEEK END	EC	NOT READY	HEAD ADDRESS	UNIT SELECT	

Bit	R/W	Description
7-6	R	INTERRUPT CODE 00 = Command completed successfully 01 = Command started but did not complete successfully 10 = Command was never started 11 = Abnormal termination (disk drive ready signal changed state during command execution)
5	R	SEEK END 0 = Seek not complete 1 = Seek complete
4	R	EC - Equipment Check 0 = No error detected 1 = Fault signal detected or, during a recalibrate, the track 0 signal was not detected after 77 step pulses
3	R	NOT READY 0 = Drive was ready 1 = Drive not ready signal was detected
2	R	HEAD ADDRESS 0 = Side 0 selected 1 = Side 1 selected
1-0	R	UNIT SELECT 00 = Drive 0 selected 01 = Drive 1 selected 10 = Drive 2 selected 11 = Drive 3 selected

Internal Register - Status Register 1

7	6	5	4	3	2	1	0
EN	0	DATA ERROR	OVERRUN	0	NO DATA	NW	MISSING ADDRESS MARK

Bit	R/W	Description
7	R	EN - End of Cylinder 0 = No error 1 = Controller attempted to access a sector beyond the last sector of a cylinder
6	R	Always 0
5	R	DATA ERROR 0 = No error 1 = Controller detected a cyclic redundancy check (CRC) error in the ID or data field
4	R	OVERRUN 0 = No error 1 = During a data transfer in non-DMA mode, the processor did not service the controller within the required time interval
3	R	Always 0
2	R	NO DATA 0 = No error 1 = One of the following conditions occurred: During execution of a read data, a write-deleted data, or a scan command, the controller could not find the specified sector. During execution of a read ID command, the controller could not read the ID field. During execution of a read track command, the starting sector could not be found.

Bit	R/W	Description (Status Register 1 - cont.)
1	R	NW - Not Writable 0 = No error 1 = During a write data, write-deleted data, or format track command, the controller detected a write-protect signal from the disk drive.
0	R	MISSING ADDRESS MARK 0 = No error 1 = One of the following conditions occurred: The controller had detected the index hole twice, but had not detected the ID field ADDRESS MARK. The controller could not detect the DATA ADDRESS MARK or the DELETED DATA ADDRESS MARK. When this bit is set, status register 2 bit 0 (MD) is set.

Internal Register - Status Register 2

7	6	5	4	3	2	1	0
0	CONTROL MARK	DATA ERROR IN DATA FIELD	WC	SCAN HIT EQUAL	SN	BC	MD

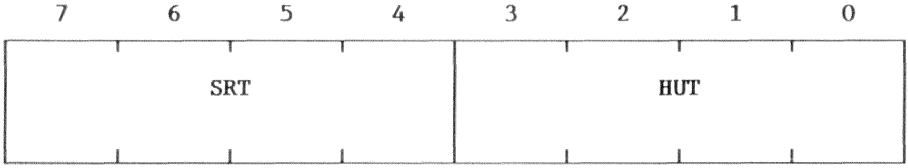
Bit	R/W	Description
7	R	Always 0
6	R	CONTROL MARK 0 = DELETED DATA ADDRESS MARK not detected 1 = During a read data or scan command, the controller found a DELETED DATA ADDRESS MARK.
5	R	DATA ERROR IN DATA FIELD 0 = No error 1 = Controller detected a cyclic redundancy check (CRC) error in the data field.
4	R	WC - Wrong Cylinder 0 = No error 1 = Cylinder number in the ID field does not match the cylinder number in the internal register
3	R	SCAN EQUAL HIT 0 = No match 1 = During the execution of a scan command, the equal condition was satisfied.
2	R	SN - Scan Not Satisfied 0 = No error 1 = During the execution of a scan command, the controller could not find a sector on the cylinder that met the condition.
1	R	BC - Bad Cylinder 0 = No error 1 = Cylinder number in the ID field is FFH and does not match the cylinder number in the internal register
0	R	MD - Missing ADDRESS MARK in Data Field 0 = No error 1 = During execution of a read command, the controller could not find a DATA ADDRESS MARK or DELETED DATA ADDRESS MARK.

Internal Register - Status Register 3

7	6	5	4	3	2	1	0
FAULT	WRITE PROTECT	READY	TRACK 0	TWO SIDE	HEAD ADDRESS	UNIT SELECT	

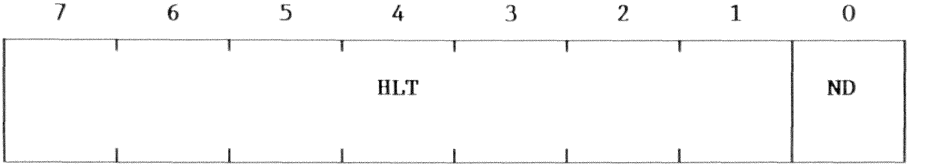
Bit	R/W	Description
7	R	FAULT 0 = No error 1 = Diskette drive fault signal detected
6	R	WRITE PROTECT 0 = Diskette not write protected 1 = Diskette drive write protect signal detected
5	R	READY 0 = Drive not ready 1 = Drive ready
4	R	TRACK 0 0 = Read/Write heads not over track 0 1 = Read/Write heads over track 0
3	R	TWO SIDE 0 = Diskette is single sided 1 = Diskette is double sided
2	R	HEAD ADDRESS 0 = Side 0 selected 1 = Side 1 selected
1-0	R	UNIT SELECT 00 = Drive 0 selected 01 = Drive 1 selected 10 = Drive 2 selected 11 = Drive 3 selected

Internal Register - SRT/HUT



Bit	R/W	Description			
7-4	W	SRT - Step Rate			
		0000 =	16 ms	1000 =	8 ms
		0001 =	15 ms	1001 =	7 ms
		0010 =	14 ms	1010 =	6 ms
		0011 =	13 ms	1011 =	5 ms
		0100 =	12 ms	1100 =	4 ms
		0101 =	11 ms	1101 =	3 ms
		0110 =	10 ms	1110 =	2 ms
		0111 =	9 ms	1111 =	1 ms
3-0	W	HUT - Head Unload Time			
		0000 =	0	1000 =	128 ms
		0001 =	16 ms	1001 =	144 ms
		0010 =	32 ms	1010 =	160 ms
		0011 =	48 ms	1011 =	176 ms
		0100 =	64 ms	1100 =	192 ms
		0101 =	80 ms	1101 =	208 ms
		0110 =	96 ms	1110 =	224 ms
		0111 =	112 ms	1111 =	240 ms

Internal Register - HLT/ND



Bit	R/W	Description
7-1	W	HLT - Head Load Time 0000000 = No head load time 0000001-1111111 = 2 ms to 254 ms in 2 ms steps
0	W	ND - Non-DMA Mode 0 = DMA mode enabled 1 = DMA mode disabled

Internal Register - C

This 8-bit register specifies the currently selected cylinder/track number. To ensure that it is at the correct cylinder/track, the diskette controller compares this cylinder/track number to the cylinder/track in the sector header.

Internal Register - H

This 8-bit register specifies the currently selected read/write head. To ensure that it is on the correct side of the diskette, the diskette controller compares this head address to the head address in the sector header. Only 0 and 1 are valid values.

Internal Register - R

This 8-bit register specifies the desired sector number.

Internal Register - N

This 8-bit register specifies the number of data bytes per sector as follows:

Value	Bytes per Sector
00H	128
01H	256
02H	512
03H	1024
04H	2048
05H	4096
06H	8192

Internal Register - EOT

This 8-bit register specifies the last sector of a read/write operation. The value written to this register is the last desired sector plus 1.

For example, to read or write one sector (sector number 5), internal register R would contain 05H and internal register EOT would contain 06H. To read or write five sectors (starting at sector 1), internal register R would contain 01H and internal register EOT would contain 06H.

Internal Register - GPL

This 8-bit register specifies the gap between sectors. When executing the format-track command, use a value of 54H. Otherwise, use a value of 1BH. These are the values specified by the ROM BIOS. See Interrupt 13H in Chapter 15.

Internal Register - DTL

When internal register N contains 00H, this 8-bit register specifies the number of bytes to be read from or written into a sector. For this register, the ROM BIOS defines a value of FFH. See Interrupt 13H in Chapter 15.

Internal Register - SC

For the format-track command, this 8-bit register specifies the number sectors per track.

Internal Register - D

For the format-track command, this 8-bit register specifies the value used as a fill byte. For this register, the ROM BIOS defines a value of F6H. See Interrupt 13H in Chapter 15.

Internal Register - STP

For the scan commands, this register specifies contiguous sectors (interleave of 1) or alternate sectors (interleave of 2).

Internal Register - PCN

For the sense-interrupt-status command, this 8-bit register returns the resulting present-cylinder number

Internal Registers - NCN

For the seek command, this 8-bit register specifies the desired cylinder/track number (new cylinder number).

Diskette Drive Controller Programming

The diskette drive controller has three operational states, command, execution, and result. The current state is determined by bit 7 (REQUEST FOR MASTER) and bit 6 (I/O DIR) of the main status register. If bit 7 is equal to 0, the diskette drive controller is in the execution state. Otherwise, the diskette drive controller is in a command or result state. Bit 6 determines whether the diskette drive controller is in the command or result state. If bit 6 is equal to 0, the diskette drive controller is in the command state. Otherwise, the diskette drive controller is in the result state.

Command State

The diskette drive controller accepts a series of 1 to 9 command bytes that are written to the data register. Each command has a fixed set of data bytes that are required to initiate the command. For correct results, the set must not be shortened.

On acceptance of a command, the diskette drive controller enters the execution state. If a command is not accepted as a valid command, the diskette drive controller sets the internal register, status register 0, equal to 80H.

The diskette drive controller has fifteen commands. Table 11-2 lists the diskette drive controller commands. The commands listed in Table 11-2 are described later in this chapter. The four internal status registers, 0-3, are described in the section on *Diskette Drive Controller Internal Registers.*

Table 11-2: **Diskette Drive Controller Commands**

Command	Description
Read Data	Multi-sector read of sectors with DATA ADDRESS MARK in header (at the current track)
Write Data	Multi-sector write at the current track (writes a DATA ADDRESS MARK in header)
Read Deleted Data	Multi-sector read at the current track (including those with a DELETED DATA ADDRESS MARK in header)
Write Deleted Data	Multi-sector write at the current track (writes a DELETED DATA ADDRESS MARK)
Read Track	Read all sectors at the current track
Read ID	Reads the ID field of the first sector encountered at the current track
Format Track	Formats sectors in the track as indicated
Scan Equal	Data on the diskette is compared for equality to data in memory (8-bit data)
Scan Low or Equal	Data on the diskette is compared for equality or a value less than the data in memory (8-bit data)
Scan High or Equal	Data on the diskette is compared for equality or a value greater than the data in memory (8-bit data)
Recalibrate	Read/Write heads retract to track 0
Sense Interrupt Status	Returns the internally stored status registers
Specify	Sets the diskette controller parameters HEAD LOAD, HEAD UNLOAD, and STEP RATE
Sense Drive Status	Loads the current drive status into the internal register, status register 3
Seek	Read/Write heads move to the specified track

Execution State

The diskette drive controller executes the command as instructed. When the operation is complete, the diskette drive controller generates an interrupt to the processor and enters the result state.

NOTE
The seek and recalibrate commands do not have a result state.

The overlapped seek or recalibration capability, described in the following explanation, is not supported by VAXmate diskette drive controllers or industry-standard diskette drive controllers.

The floppy disk controller chip supports overlapped seeks and recalibrations. That is, issuing a seek or recalibrate command to two or more drives before the previous seek or recalibrate commands have completed. To provide this feature, the result state was eliminated. After issuing one or more seek or recalibrate commands, the controlling program must monitor the main status register. Bits 3-0 of the main status register reflect the status of the corresponding drive.

Result State

On completion of a command, the diskette drive controller provides a series of status bytes that are read from the data register. These status bytes represent the states of corresponding internal registers. Each command has a fixed set of status bytes that result from a command. Until all of the status bytes have been read, the diskette drive controller will not accept a new command.

Command and Result Register Sets

Each command has a specific set of internal registers that must be written through the data register. During the result state, each command has a specific set of internal registers that must be read through the data register. Tables 11-3 through 11-17 define the command and result register sets for the various commands.

Invalid command codes produce a result state that contains only status register 3.

Table 11-3 Register Sets for Read Data Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	DTL	Data Length
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-4 Register Sets for Write Data Command

State	Order	Register	Comment
Command	1	Command	SK must be 0
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	DTL	Data Length
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-5 Register Sets for Read Deleted Data Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	DTL	Data Length
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-6 Register Sets for Write Deleted Data Command

State	Order	Register	Comment
Command	1	Command	SK must be 0
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	DTL	Data Length
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-7 Register Sets for Read Track Command

State	Order	Register	Comment
Command	1	Command	MT must be 0
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	DTL	Data Length
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-8 Register Sets for Read ID Command

State	Order	Register	Comment
Command	1	Command	MT and SK must be 0
	2	Head/Unit Select	
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-9 Register Sets for Format Track Command

State	Order	Register	Comment
Command	1	Command	MT and SK must be 0
	2	Head/Unit Select	
	3	N	Sector Size
	4	SC	Sectors per track
	5	GPL	Gap length
	6	D	Fill data
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-10 Register Sets for Scan Equal Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	STP	Interleave (1 or 2)
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-11 Register Sets for Scan Low or Equal Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	STP	Interleave (1 or 2)
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-12 Register Sets for Scan High or Equal Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	C	Cylinder
	4	H	Head address
	5	R	Sector number
	6	N	Sector Size
	7	EOT	Last sector for operation
	8	GPL	Gap length
	9	STP	Interleave (1 or 2)
Result	1	Status Register 0	
	2	Status Register 1	
	3	Status Register 2	
	4	C	Cylinder
	5	H	Head address
	6	R	Sector number
	7	N	Sector Size

Table 11-13 Register Sets for Recalibrate Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
Result		None	Issue a sense interrupt status command

Table 11-14 Register Sets for Sense Interrupt Status Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
Result	1	Status Register 0	
	2	PCN	Present cylinder number

Table 11-15 Register Sets for Specify Command

State	Order	Register	Comment
Command	1	Command	
	2	SRT/HUT	
	3	HLT/ND	
Result		None	Command does not have a result state

Table 11-16 Register Sets for Sense Drive Status Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
Result		Status Register 3	

Table 11-17 Register Sets for Seek Command

State	Order	Register	Comment
Command	1	Command	
	2	Head/Unit Select	
	3	NCN	New cylinder number
Result		None	Issue a sense interrupt status command

Programming Example

The following programming example demonstrates:

- Initializing the diskette drive controller
- Using DMA data transfers
- Recalibrating the diskette drive
- Seeking to a track
- Hard formatting a diskette

CAUTION
Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

```
#include "kyb.h"
#include "example.h"

/*************************************************************************/
/* define constants used in diskette controller example                  */
/*************************************************************************/

/* define bit values for diskette controller control register (DCCR)   */

#define DRV_SEL      0x01                  /* bit mask for drive select */
#define FDC_ON       0x04                 /* bit value allows fdc to run
                                        if this bit not set, fdc is reset */
#define DMA_INT_ON   0x08  /* value to enable DMA and interrupts to CPU */
#define DRVA_MOTOR   0x10          /* bit value to turn on drive a motor */
#define DRVB_MOTOR   0x20          /* bit value to turn on drive b motor */

/* define bit values for data transfer rate register */

#define DTR_500      0x00        /* bit value for 500 Kbit transfer rate */
#define DTR_300      0x01            /* VAXmate = 250 Kbit transfer rate */
#define DTR_250      0x10        /* bit value for 250 Kbit transfer rate */

/* define disk change register bit */

#define DISK_CHG     0x80                     /* diskette changed if set */

/* define bit values for FDC main status register */

#define FDD0_BUSY    0x01            /* diskette drive 0 busy doing seek */
#define FDD1_BUSY    0x02            /* diskette drive 1 busy doing seek */
#define FDD2_BUSY    0x04            /* diskette drive 2 busy doing seek */
#define FDD3_BUSY    0x08            /* diskette drive 3 busy doing seek */
#define FDD_BUSY     FDD0_BUSY | FDD1_BUSY | FDD2_BUSY | FDD3_BUSY
#define FDC_CB       0x10                              /* controller busy */
#define FDC_NDM      0x20      /* in non-DMA mode = execution phase busy */
#define DIO_RD       0x40    /* indicates processor should read data reg */
#define RQM          0x80      /* data register ready to send or receive */
```

```
/* define status register 0 bit values */

#define SR0_US0     0x00   /* at interrupt time, unit select = drive 0 */
#define SR0_US1     0x01   /* at interrupt time, unit select = drive 1 */
#define SR0_US2     0x02   /* at interrupt time, unit select = drive 2 */
#define SR0_US3     0x03   /* at interrupt time, unit select = drive 3 */
#define SR0_HD      0x04             /* head address at interrupt time */
#define SR0_NR      0x08                  /* diskette drive not ready */
#define SR0_EC      0x10   /* equipment check, could not reach track 0 */
#define SR0_SE      0x20                    /* seek command completed */
#define SR0_IC_AT   0x40     /* interrupt code = abnormal termination */
#define SR0_IC_IC   0x80          /* interrupt code = invalid command */
#define SR0_IC_NR   0xc0          /* interrupt code = drive not ready */
#define SR0_IC_NT   0x00       /* interrupt code = normal termination */

/* define status register 1 bit values */

#define SR1_MA      0x01                        /* missing address mark */
#define SR1_NW      0x02                /* write protect signal detected */
#define SR1_ND      0x04  /* couldn't find sector, or couldn't read ID */
#define SR1_OR      0x10                /* did not receive data in time */
#define SR1_DE      0x20           /* data field or ID field CRC error */
#define SR1_EN      0x80  /* tried to access sector at end of cylinder */

/* define status register 2 bit values */

#define SR2_MD      0x01         /* missing address mark in data field */
#define SR2_BC      0x02                                /* bad cylinder */
#define SR2_SN      0x04       /* scan command could not find a sector */
#define SR2_SH      0x08                              /* scan equal hit */
#define SR2_WC      0x10                              /* wrong cylinder */
#define SR2_DD      0x20                    /* CRC error in data field */
#define SR2_CM      0x40            /* deleted data address mark found */

/* define status register 3 bit values */

#define SR3_US0     0x00                       /* unit select - drive 0 */
#define SR3_US1     0x01                       /* unit select - drive 1 */
#define SR3_US2     0x02                       /* unit select - drive 2 */
#define SR3_US3     0x03                       /* unit select - drive 3 */
#define SR3_HD      0x04                               /* head address */
#define SR3_TS      0x08                    /* drive signal - two side */
#define SR3_TO      0x10                     /* drive signal - track 0 */
#define SR3_RDY     0x20                       /* drive signal - ready */
#define SR3_WP      0x40                /* drive signal - write protect */
#define SR3_FT      0x80                       /* drive signal - FAULT */
```

```
/* define base values of fdc commands */

#define FDC_RD      0x06                                /* read data */
#define FDC_RDD     0x0c                          /* read deleted data */
#define FDC_WD      0x05                                /* write data */
#define FDC_WDD     0x09                         /* write deleted data */
#define FDC_RT      0x02                                /* read track */
#define FDC_ID      0x0a                                  /* read ID */
#define FDC_FT      0x0d                              /* format track */
#define FDC_SE      0x11                                /* scan equal */
#define FDC_SLE     0x19                          /* scan low or equal */
#define FDC_SHE     0x1d                         /* scan high or equal */
#define FDC_RECAL   0x07                           /* recalibrate drive */
#define FDC_SIS     0x08                     /* sense interrupt status */
#define FDC_SPE     0x03                                  /* specify */
#define FDC_SDS     0x04                        /* sense drive status */
#define FDC_SEEK    0x0f                                    /* seek */
#define FDC_MT      0x80                               /* multi-track */
#define FDC_MFM     0x40           /* modified frequency modulation */
#define FDC_SK      0x20          /* skip deleted data address mark */

/*************************************************************************/
/* define some general constants                                         */
/*************************************************************************/

#define RETRY_COUNT 4                                 /* maximum retries */

/*************************************************************************/
/* define some error codes                                               */
/*************************************************************************/

#define ERR_FATAL   0xffff            /* fatal error of unknown origin */
#define ERR_FAT_RD  0xfffe       /* fdc was expecting write not read */
#define ERR_FAT_WR  0xfffd       /* fdc was expecting read not write */
#define ERR_TO      0xfffc                          /* time out error */
#define ERR_DNR     0xfffb                        /* drive not ready */
#define ERR_RECAL   0x0001                     /* recalibrate error */
#define ERR_SEEK    0x0002                           /* seek error */
```

```
/*************************************************************************/
/* declare structures used in diskette controller example                */
/*************************************************************************/

typedef struct
{
  unsigned char dccr;           /* diskette controller control register */
  unsigned char reserved1;          /* I/O space not used by controller */
  unsigned char fdc_stat;  /* diskette controller main status register */
  unsigned char fdc_data;          /* diskette controller data register */
  unsigned char reserved2;          /* I/O space not used by controller */
  unsigned char dtr;          /* read <------- diskette change register */
                              /* write ---> data transfer rate register */
} FDC;

#define FDC_BASE (FDC *)0x03F2        /* base address of FDC structure */

typedef struct
{
  unsigned char mt;                                      /* multi-track */
  unsigned char mfm;                                         /* mfm/fm */
  unsigned char sk;                                           /* skip */
  unsigned char last_cmd;                    /* last command sent to fdc */
  int busy;                                                /* busy flag */
  int retry;                                              /* retry count */
  unsigned char dccr;                                    /* dccr contents */
  unsigned char dtr;                                /* data transfer rate */
  int hsd;                                           /* head settle delay */
  int msd;                                       /* motor start up delay */
  int mod;                                            /* motor off delay */
  unsigned char ds;                                 /* drive select 0 - 3 */
  unsigned char c;                                   /* cylinder number */
  unsigned char h;                                        /* head side */
  unsigned char r;                                   /* sector number */
  unsigned char n;                                  /* bytes per sectors */
  unsigned char eot;                                    /* end of track */
  unsigned char sgpl;                               /* sector gap length */
  unsigned char fgpl;                               /* format gap length */
  unsigned char sc;                                   /* sector count */
  unsigned char d;                                  /* format fill byte */
  unsigned char dtl;                                    /* data length */
  unsigned char stp;                          /* scan skip sector flag */
  unsigned char srt;                                  /* step rate time */
  unsigned char hlt;                                  /* head load time */
  unsigned char hut;                                /* head unload time */
  unsigned char nd;                                     /* non-DMA mode */
} FDC_CMD;
```

```
typedef struct
{
  unsigned char mt;                          /* multi-track */
  unsigned char mfm;                             /* mfm/fm */
  unsigned char sk;                                /* skip */
  unsigned char dtr;                   /* data transfer rate */
  int hsd;                              /* head settle delay */
  int msd;                            /* motor start up delay */
  unsigned char c;                       /* cylinder number */
  unsigned char h;                             /* head side */
  unsigned char r;                       /* sector number */
  unsigned char n;                     /* bytes per sectors */
  unsigned char eot;                       /* end of track */
  unsigned char sgpl;                  /* sector gap length */
  unsigned char fgpl;                  /* format gap length */
  unsigned char sc;                        /* sector count */
  unsigned char dtl;                       /* data length */
  unsigned char srt;                     /* step rate time */
  unsigned char hlt;                     /* head load time */
  unsigned char hut;                   /* head unload time */
} FDD;

typedef struct
{
  unsigned char st0;                   /* status register 0 */
  unsigned char st1;                   /* status register 1 */
  unsigned char st2;                   /* status register 2 */
  unsigned char st3;                   /* status register 3 */
  unsigned char c;                       /* cylinder number */
  unsigned char h;                             /* head side */
  unsigned char r;                       /* sector number */
  unsigned char n;                     /* bytes per sectors */
  unsigned char pcn;             /* present cylinder number */
  unsigned int error;                  /* error code/status */
  unsigned char change;         /* diskette change register */
} FDC_RESULT;
```

```
/*************************************************************************/
/* declare some external timers                                          */
/*************************************************************************/

extern int head_settle;          /* head settle and motor startup timer */
extern int motor_flag;               /* automatic motor shut off timer */

/*************************************************************************/
/* declare space for fdc parameter data                                  */
/*************************************************************************/

FDC_CMD fdc_cmd =
{
  0x00,                                        /* not multi-track to start */
  FDC_MFM,                                                 /* always mfm */
  0x00,                                                  /* not skipping */
  0x00,                                             /* no last command yet */
  FALSE,                                                   /* not busy yet */
  0x00,                                                 /* current retries */
  0x00,                                 /* nothing enabled until fdc is reset */
  DTR_500,                                /* data transfer rate is 500 Kbits */
  5,                 /* 3.90625 ms * 5 = 19.5312 ms = head settle delay */
  128,                 /* 3.90625 ms * 128 = 500 ms motor startup delay */
  512,                  /* 3.90625 ms * 512 = 2 seconds motor off delay */
  0x00,                                              /* no drive selected */
  0x00,                                             /* cylinder 0 to start */
  0x00,                                              /* head zero to start */
  0x01,                                                /* sector 1 to start */
  0x02,                                           /* 512 bytes per sectors */
  0x10,                                         /* end of track at sector 16 */
  0x1b,                                                /* sector gap length */
  0x54,                                                /* format gap length */
  0x0f,                                            /* 15 sectors per track */
  0xf6,                                                 /* format fill byte */
  0xff,                                                     /* data length */
  0x00,                                  /* not skipping sectors during scan */
  0x0d,     /* 3 ms step rate (1 - 16 ms in 1 ms increment) 0x0f = 1 ms */
  0x32,         /* 50 ms head load time ( 0x01 = 2 ms, 0x02 = 4 ms ...) */
  0x08,     /* 128 ms head unload time ( 0x01 = 16 ms, 0x02 = 32ms ...) */
  0x00,       /* select dma mode (0x00 = dma mode, 0x01 = non-dma mode) */
};

FDD fdd[2];                           /* place to store diskette parameters */
FDC_RESULT fdc_result;             /* place to store result and error codes */
```

```
/*************************************************************************/
/* motor_off() - turn diskette drive motors off                          */
/*************************************************************************/
motor_off()
{
FDC *pfdc = FDC_BASE;
int intr_flag;                                  /* to hold CPU IF state */

  fdc_cmd.dccr &=   (DRVA_MOTOR | DRVB_MOTOR);       /* both motors off */
  outp(&pfdc->dccr, fdc_cmd.dccr);                    /* turn them off */
  intr_flag = int_off();                          /* no interrupts please */
  motor_flag = 0;                                    /* clear motor timer */
  int_on(intr_flag);                                  /* allow interrupts */
}

/*************************************************************************/
/* select() - select the desired drive and turn on motor                 */
/*************************************************************************/
select()
{
FDC *pfdc = FDC_BASE;
int intr_flag;                                  /* to hold CPU IF state */

  if((fdc_cmd.dccr & DRV_SEL) != fdc_cmd.ds ||       /* if not current */
     (fdc_cmd.dccr & (DRVA_MOTOR | DRVB_MOTOR)) == 0)/* all motors off */
  {
    fdc_cmd.dccr &=   DRV_SEL;                        /* deselect drives */
    fdc_cmd.dccr |= fdc_cmd.ds;                        /* select drive */
    fdc_cmd.dccr &=   (DRVA_MOTOR | DRVB_MOTOR);     /* both motors off */
    if(!fdc_cmd.ds) fdc_cmd.dccr |= DRVA_MOTOR;    /* desired motor on */
    else fdc_cmd.dccr |= DRVB_MOTOR;
    outp(&pfdc->dccr, fdc_cmd.dccr);              /* write the register */
    intr_flag = int_off();                        /* no interrupts please */
    motor_flag = 0;                                  /* clear motor timer */
    int_on(intr_flag);                                /* allow interrupts */
  }
  if(!motor_flag)                                  /* has motor stopped ? */
  {
    head_settle = fdc_cmd.msd;   /* head settle timer to time start up */
    while(head_settle)              /* wait until motor is up to speed */
      ;
  }
  intr_flag = int_off();                          /* no interrupts please */
  motor_flag = fdc_cmd.mod;          /* write the motor off delay time */
  int_on(intr_flag);                                /* allow interrupts */
}
```

```
/*************************************************************************/
/* fdc_in() - read data from fdc data register                            */
/*************************************************************************/

fdc_in()
{
FDC *pfdc = FDC_BASE;
int i;

    head_settle = 2;       /* clk cycle = 3.9 ms, must read in 2 cycles */
  i = inp(&pfdc->fdc_stat);                /* read fdc status register */
    while(!(i & RQM))                                  /* fdc ready ? */
    {
      if(!head_settle)                               /* time out error ? */
      {
        fdc_result.error = ERR_TO;                          /* mark error */
        return;
      }
      else i = inp(&pfdc->fdc_stat);       /* read fdc status register */
    }
  if(i & DIO_RD)                          /* data direction = cpu read ? */
    return(inp(&pfdc->fdc_data));                    /* return data read */
  else fdc_result.error = ERR_FAT_RD;                /* mark fatal error */
}
```

```
/*********************************************************************/
/* fdc_out() - write data to fdc data register                       */
/*********************************************************************/

fdc_out(value)

unsigned char value;            /* value to write to fdc data register */

{
FDC *pfdc = FDC_BASE;
int i;

    head_settle = 2;      /* clk cycle = 3.9 ms, must read in 2 cycles */
  i = inp(&pfdc->fdc_stat);              /* read fdc status register */
    while(!(i & RQM))                                  /* fdc ready ? */
    {
      if(!head_settle)                              /* time out error ? */
      {
        fdc_result.error = ERR_TO;                        /* mark error */
        return;
      }
      else i = inp(&pfdc->fdc_stat);       /* read fdc status register */
    }
  if((i & DIO_RD) == 0)               /* data direction = cpu write ? */
    outp(&pfdc->fdc_data, value);          /* write fdc data register */
  else fdc_result.error = ERR_FAT_WR;              /* mark fatal error */
}

/*********************************************************************/
/* waitcc() - wait for command to complete                           */
/*********************************************************************/

waitcc()
{
  while(fdc_cmd.busy)                   /* wait until command complete */
  {
    if(!motor_flag)                                    /* time out ? */
    {
      fdc_result.error = ERR_TO;                    /* time out error */
      return;
    }           /* do something useful while waiting, like check date */
    chk_dt();         /* and time for update. ROM BIOS does an INT 15H */
  }
```

```
  if(fdc_result.error) return;                              /* error ? */
  switch(fdc_cmd.last_cmd)              /* discover last command issued */
  {
    case FDC_RECAL:                                    /* recalibrate ? */
    case FDC_SEEK:                                          /* seek ? */
      head_settle = fdc_cmd.hsd;          /* set head settle delay timer */
      while(head_settle)              /* wait for head settle to time out */
        ;
      fdc_cmd.last_cmd = FDC_SIS;              /* set last command issued */
      fdc_out(FDC_SIS);                         /* sense interrupt status */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.st0 = fdc_in();                     /* read st0 results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.pcn = fdc_in();          /* read present cylinder number */
      if(fdc_result.error) return;                          /* error ? */
      break;

    default:/* all other commands except FDC_SPE, FDC_SIS, and FDC_SDS */
      fdc_result.st0 = fdc_in();                     /* read st0 results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.st1 = fdc_in();                     /* read st1 results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.st2 = fdc_in();                     /* read st2 results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.c = fdc_in();                  /* read cylinder results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.h = fdc_in();                    /* read head results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.r = fdc_in();                  /* read sector results */
      if(fdc_result.error) return;                          /* error ? */
      fdc_result.n = fdc_in();             /* read bytes/sector results */
      if(fdc_result.error) return;                          /* error ? */
      break;
  }
}
```

```
/*************************************************************************/
/* specify() - set the diskette drive characteristics                    */
/*************************************************************************/

specify()
{
FDC *pfdc = FDC_BASE;
unsigned char uc;                                   /* temporary variable */

  outp(&pfdc->dtr, fdc_cmd.dtr);                  /* set data transfer rate */
  outp(&pfdc->dccr, fdc_cmd.dccr);                  /* set control register */
  fdc_cmd.last_cmd = FDC_SPE;                    /* last command is specify */
  fdc_out(FDC_SPE);                               /* issue specify command */
  if(fdc_result.error) return;                                  /* error ? */
  uc = fdc_cmd.srt << 4;                             /* specify step rate */
  uc |= fdc_cmd.hut;                            /* specify head unload time */
  fdc_out(uc);                  /* issue step rate and head unload time */
  if(fdc_result.error) return;                                  /* error ? */
  uc = fdc_cmd.hlt << 1;                           /* specify head load time */
  uc |= fdc_cmd.nd;                                   /* specify dma mode */
  fdc_out(uc);                     /* issue head load time and dma mode */
}
```

```
/*************************************************************************/
/* fdc_issue() - issue all fdc commands except specify and sis           */
/*************************************************************************/

fdc_issue(cmd, drv)

int cmd;                                          /* desired command */
int drv;                                        /* desired drive 0 or 1 */

{
FDC *pfdc = FDC_BASE;
char oline[20];

  fdc_cmd.ds = drv;                                /* indicate the drive */
  select(drv);                                   /* select appropriate drive */
  fdc_out(FDC_SDS);                                /* sense drive status */
  if(fdc_result.error) return;                                 /* error ? */
  fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);          /* second byte of SDS */
  if(fdc_result.error) return;                                 /* error ? */
  fdc_result.st3 = fdc_in();                        /* read st3 results */
  if(fdc_result.error) return;                                 /* error ? */
  sprintf(oline, "%04x %04x", pfdc, &pfdc->dtr);
  disp_str(16, 1, oline);
  fdc_result.change = inp(&pfdc->dtr) & 0x80;    /* read disk change reg */
  if(fdc_result.st3 & SR3_RDY)                          /* drive ready ? */
  {
    fdc_cmd.last_cmd = cmd;                  /* set last command issued */
    switch(cmd)
    {
      case FDC_SDS:
      return;
      break;

      case FDC_RECAL:                                /* recalibrate ? */
        fdc_out(cmd);                                 /* issue byte 1 */
        if(fdc_result.error) return;                       /* error ? */
        fdc_out(fdc_cmd.ds);                          /* issue byte 2 */
        break;

      case FDC_SEEK:                                         /* seek ? */
        fdc_out(cmd);                                 /* issue byte 1 */
        if(fdc_result.error) return;                       /* error ? */
        fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);       /* issue byte 2 */
        if(fdc_result.error) return;                       /* error ? */
        fdc_out(fdc_cmd.c);                           /* issue byte 3 */
        break;
```

```
case FDC_ID:                                        /* read ID ? */
  fdc_out(fdc_cmd.mfm | cmd);                     /* issue byte 1 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);         /* issue byte 2 */
  break;

case FDC_FT:                                      /* format track ? */
  fdc_out(fdc_cmd.mfm | cmd);                     /* issue byte 1 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);         /* issue byte 2 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.n);                             /* issue byte 3 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.sc);                            /* issue byte 4 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.fgpl);                          /* issue byte 5 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.d);                             /* issue byte 6 */
  if(fdc_result.error) return;                        /* error ? */
  break;

case FDC_RD:                                         /* read data ? */
case FDC_RDD:                                /* read deleted data ? */
case FDC_WD:                                        /* write data ? */
case FDC_WDD:                               /* write deleted data */
  fdc_out(fdc_cmd.mt | fdc_cmd.mfm | fdc_cmd.sk | cmd);/* byte 1 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);         /* issue byte 2 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.c);                             /* issue byte 3 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.h);                             /* issue byte 4 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.r);                             /* issue byte 5 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.n);                             /* issue byte 6 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.eot);                           /* issue byte 7 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.sgpl);                          /* issue byte 8 */
  if(fdc_result.error) return;                        /* error ? */
  fdc_out(fdc_cmd.dtl);                           /* issue byte 9 */
  break;
```

```
    case FDC_RT:                                    /* read a track ? */
      fdc_out(fdc_cmd.mfm | fdc_cmd.sk | cmd);      /* issue byte 1 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);       /* issue byte 2 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.c);                           /* issue byte 3 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.h);                           /* issue byte 4 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.r);                           /* issue byte 5 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.n);                           /* issue byte 6 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.eot);                         /* issue byte 7 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.sgpl);                        /* issue byte 8 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.dtl);                         /* issue byte 9 */
      break;

    case FDC_SE:                                    /* scan equal ? */
    case FDC_SHE:                                /* scan high or equal ? */
    case FDC_SLE:                                 /* scan low or equal ? */
      fdc_out(fdc_cmd.mt | fdc_cmd.mfm | fdc_cmd.sk | cmd);/* byte 1 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out((fdc_cmd.h << 2) | fdc_cmd.ds);       /* issue byte 2 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.c);                           /* issue byte 3 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.h);                           /* issue byte 4 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.r);                           /* issue byte 5 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.n);                           /* issue byte 6 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.eot);                         /* issue byte 7 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.sgpl);                        /* issue byte 8 */
      if(fdc_result.error) return;                      /* error ? */
      fdc_out(fdc_cmd.stp);                         /* issue byte 9 */
      break;
  }
}
```

```
  else
  {
    fdc_result.error = ERR_DNR;              /* drive not ready error */
    return;
  }
  if(fdc_result.error) return;                              /* error ? */
  fdc_cmd.busy = TRUE;
  waitcc();                                /* wait for command complete */
  disp_status(&fdc_result, &fdc_cmd);        /* display result status */
}

/*************************************************************************/
/* fdc_init() - initialize diskette controller                           */
/*************************************************************************/

fdc_init()
{
FDC *pfdc = FDC_BASE;
int intr_flag;

  intr_flag = int_off();
  outp(&pfdc->dccr, DMA_INT_ON);                        /* reset the fdc */
  fdc_cmd.dccr = DMA_INT_ON | FDC_ON;                   /* fdc not reset */
  outp(&pfdc->dccr, fdc_cmd.dccr);              /* allow communications */
  iv_init(0x0e);                     /* initialize the interrupt vector */
  int_on(intr_flag);
  imask(0, 6, 1);                                  /* enable PIC input */
}

/*************************************************************************/
/* fdc_rest() - restore diskette controller and interrupt vector         */
/*************************************************************************/

fdc_rest()
{
FDC *pfdc = FDC_BASE;

  outp(&pfdc->dccr, 0);                                 /* reset the fdc */
  imask(0, 6, 0);                                 /* disable PIC input */
  fdc_cmd.dccr = DMA_INT_ON | FDC_ON;                   /* fdc not reset */
  outp(&pfdc->dccr, fdc_cmd.dccr);              /* allow communications */
  iv_rest(0x0e);                       /* restore the interrupt vector */
}
```

```
/*************************************************************************/
/* fdc_int_hand() - fdc interrupt handler                                */
/*************************************************************************/

fdc_int_hand()
{
  fdc_cmd.busy = FALSE;                              /* no longer busy */
  eoi(0);
}

/*************************************************************************/
/* fdc() - execute diskette controller examples                          */
/*************************************************************************/

fdc()
{
static MESSAGE mfdc[] =                                  /* fdc menu */
{
  {  3, 27, "Diskette Controller Example" },
  {  5, 27, "F1.  Turn drive A motor on" },
  {  6, 27, "F2.  Turn drive A motor off" },
  {  7, 27, "F3.  Recalibrate" },
  {  8, 27, "F4.  Seek track 40" },
  {  9, 27, "F5.  Format diskette" },
  { 10, 27, "F6.  Read ID" },
  { 12, 27, "F10. Return to Main menu" },
  {  0,  0,  0 },
};

unsigned char tmp;                          /* to hold CMOS byte read */
unsigned char sum;                       /* to hold calculated checksum */
char line[512];                             /* to hold input line */
char oline[512];                            /* to hold output line */
int i;                                   /* to hold menu selection */
int r;                                             /* temp value */
FDC *pfdc = FDC_BASE;
char *pc;
char far *fpc = oline;
long l = (long)fpc;
long l1;
long l2;
int pr;
int pa;

#define ROW 16
#define COL 17
```

```
  disp_menu(mfdc);                              /* display the fdc menu */
l1 = l & 0x0000ffff;    /* build address of buffer for DMA controller */
l2 = l >> 16;
l2 <<= 4;
l = l2 + l1;
l1 = l >> 16;
pr = (int)l1;                                   /* pointer to buffer */
l1 = l & 0x0000ffff;
pa = (int)l1;                                  /* page register value */
specify();
line[0] = 0;                                     /* null terminated */
while(1)                                        /* forever (see F10) */
{
  disp_status(&fdc_result, &fdc_cmd);        /* display result status */
  line[0] = get_fkey();      /* get a function key for menu selection */
  switch(line[0])                          /* determine menu selection */
  {
    case F1:                                     /* turn drive motor on */
     fdc_cmd.ds = 0;
      select();
      break;

    case F2:                                    /* turn drive motor off */
      motor_off();
      break;

    case F3:                                       /* recalibrate drive */
      select(0);
      fdc_issue(FDC_RECAL, 0);
      fdc_issue(FDC_RECAL, 0);
      fdc_issue(FDC_ID, 0);
      break;

    case F4:                                        /* seek to track 40 */
      select(0);
      fdc_cmd.c = 40;
      fdc_issue(FDC_SEEK, 0);
      fdc_issue(FDC_ID, 0);
      break;
```

```
case F5:                /* hard format a diskette - interleave = 1 */
  select(0);
  fdc_issue(FDC_RECAL, 0);
  fdc_issue(FDC_RECAL, 0);
  fdc_issue(FDC_ID, 0);
  for(i = 0; i < 80; i++)
  {
    fdc_cmd.c = i;
    fdc_issue(FDC_SEEK, 0);
    fdc_cmd.h = 0;                                  /* side 0 */
    pc = oline;
    for(r = 1; r < 16; r++)             /* build sector table */
    {
      *pc++ = (char)i;
      *pc++ = '\000';
      *pc++ = (char)r;
      *pc++ = '\002';
    }
    r = (int)(1 >> 16);
    dma_transfer(2, pr, pa, 60, 8);              /* setup DMA */
    fdc_issue(FDC_FT, 0);                      /* format track */
    fdc_cmd.h = 1;                                  /* side 1 */
    pc = oline;
    for(r = 1; r < 16; r++)             /* build sector table */
    {
      *pc++ = (char)i;
      *pc++ = '\001';
      *pc++ = (char)r;
      *pc++ = '\002';
    }
    r = (int)(1 >> 16);
    dma_transfer(2, pr, pa, 60, 8);              /* setup DMA */
    fdc_issue(FDC_FT, 0);                      /* format track */
  }
  break;
```

```
      case F6:                                    /* read any sector ID */
        select(0);
        fdc_issue(FDC_RECAL, 0);
        fdc_issue(FDC_RECAL, 0);
        fdc_issue(FDC_ID, 0);
        for(i = 0; i < 80; i++)
        {
          fdc_cmd.c = i;
          fdc_issue(FDC_SEEK, 0);
          fdc_cmd.h = 0;
          fdc_issue(FDC_ID, 0);
          fdc_cmd.h = 1;
          fdc_issue(FDC_ID, 0);
        }
        break;

      case F10:                             /* return to caller (main menu) */
        return;
    }
  }
}
```

```
disp_status(pres, pcmd)                                    /* display status */

FDC_RESULT *pres;
FDC_CMD *pcmd;

{
char oline[50];

  sprintf(oline, "Error status : 0x%04x", pres->error);
  disp_str(19,  1, oline);
  sprintf(oline, "Status reg 0 :   0x%02x", pres->st0);
  disp_str(20,  1, oline);
  sprintf(oline, "Status reg 1 :   0x%02x", pres->st1);
  disp_str(21,  1, oline);
  sprintf(oline, "Status reg 2 :   0x%02x", pres->st2);
  disp_str(22,  1, oline);
  sprintf(oline, "Status reg 3 :   0x%02x", pres->st3);
  disp_str(23,  1, oline);
  sprintf(oline, "Present Cyl Num: 0x%02x", pres->pcn);
  disp_str(24,  1, oline);
  sprintf(oline, "Change reg :   0x%02x", pres->change);
  disp_str(19, 41, oline);
  sprintf(oline, "Cylinder (C):   0x%02x", pres->c);
  disp_str(20, 41, oline);
  sprintf(oline, "Head (H):   0x%02x", pres->h);
  disp_str(21, 41, oline);
  sprintf(oline, "Sector (R): 0x%02x", pres->r);
  disp_str(22, 41, oline);
  sprintf(oline, "Sectors per track (N): 0x%02x", pres->n);
  disp_str(23, 41, oline);
  sprintf(oline, "Last command : 0x%04x", pcmd->last_cmd);
  disp_str(24, 41, oline);
}
```

Chapter 12
Hard Disk Drive Controller

Introduction

The hard disk controller provides an interface between a hard disk drive and the workstation microprocessor. The hard disk controller supports the following features:

- A 16-bit data path and an 8-bit input/output (I/O) path
- ECC correction
- Programmed I/O data transfers
- Field formatting with unlimited sector interleave
- Drives with a maximum of 16 heads and 1024 cylinders

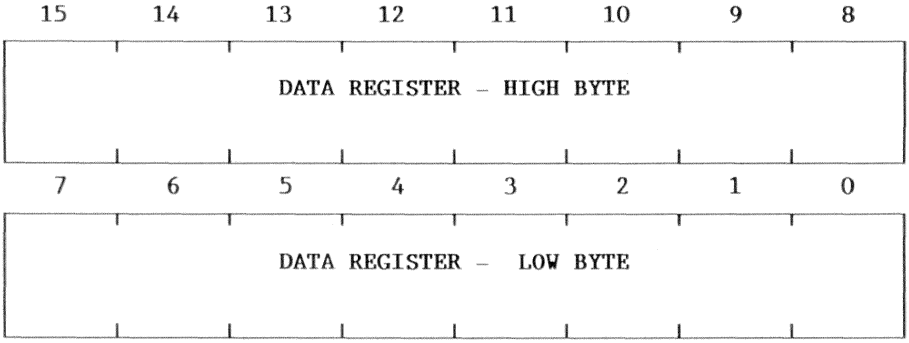
Hard Disk Controller Registers

The hard disk controller has 13 registers that control hard disk operations and provide status information. These registers are mapped to a set of primary or secondary I/O addresses. Table 12-1 lists the registers and the corresponding primary and secondary I/O addresses.

Table 12-1 Hard Disk Controller Registers

Primary Address	Secondary Address	R/W	Register
01F0H	0170H	R/W	Data register (16 bits)
01F1H	0171H	W	Write Precompensation Cylinder
01F1H	0171H	R	Error Register
01F2H	0172H	R/W	Sector Count
01F3H	0173H	R/W	Sector Number
01F4H	0174H	R/W	Cylinder Number - Low Byte
01F5H	0175H	R/W	Cylinder Number - High Byte
01F6H	0176H	R/W	SDH (sector size, drive, and head)
01F7H	0177H	W	Command Register
01F7H	0177H	R	Status Register
03F6H	0376H	W	Hard Disk Register
03F6H	0376H	R	Alternate Status Register
03F7H	0377H	R	Digital Input Register

Data Register (01F0H/0170H)



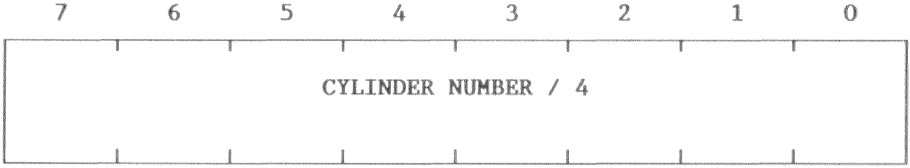
Bit	R/W	Description
15-0	R/W	DATA

The data register provides a 16-bit data path to the sector buffer. This register is accessible only during execution of a read or write command.

Except for the four ECC bytes of a read long or write long command, all data transfers are 16-bit transfers.

After transferring the data of a read long or write long command, the four ECC bytes are transferred one byte at a time. The ECC bytes are transferred through the high byte. The hard disk controller requires a minimum of 2 μs between ECC byte transfers. To transfer an ECC byte, the data request (DRQ) status bit (status register bit 3) must be set. The status register is defined later in this chapter.

Write Precompensation Register (01F1H/0171H)



Bit	R/W	Description
7-0	W	This register specifies the cylinder at which the hard disk controller begins applying write precompensation. The value written to this register is the desired cylinder number divided by 4.

Error Register (01F1H/0171H)

7	6	5	4	3	2	1	0
BAD BLOCK DETECT	ECC ERROR	0	ID NOT FOUND	0	ABORTED COMMAND DETECT	TRACK 0 ERROR	DAM NOT FOUND

Bit	R/W	Description
7	R	BAD BLOCK DETECT 0 = No error 1 = The controller read a sector ID field that contained a bad block mark
6	R	ECC ERROR 0 = No error 1 = An ECC syndrome error was detected
5	R	Always 0
4	R	ID NOT FOUND 0 = No Error 1 = The hard disk controller failed to find the desired cylinder, head, sector, or size parameter within eight revolutions of the disk, or an ID field CRC error has occurred.
3	R	Always 0
2	R	ABORTED COMMAND DETECT 0 = No error 1 = The hard disk controller aborted a command If a command is issued while the status signal, DRIVE READY L, is inactive or the status signal, WRITE FAULT L is active, the hard disk controller sets this bit.
1	R	TRACK 0 ERROR 0 = No error 1 = The hard disk controller executed a restore command, issued 2047 step pulses, and did not detect track 0.
0	R	DAM NOT FOUND - Data Address Mark Not Found 0 = No error 1 = The hard disk controller read the correct sector ID field, but it did not contain a data address mark.

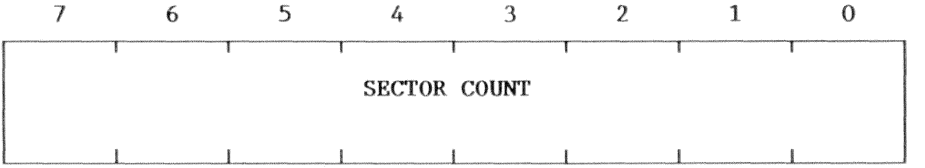
For the following conditions, the error register contains valid data:

- After a command completion interrupt, if the status register error bit (bit 0) equals 1, this register indicates the specific error condition.
- On power-up or receiving a diagnose command, the hard disk controller executes a set of diagnostic tests. On completion of those tests, the hard disk controller places a result code in this register. The result code is one of the codes listed in Table 12-2. For this condition, the status register error bit (bit 0) is ignored.

Table 12-2 Hard Disk Controller Diagnostic Result Codes

Value	Meaning
01H	No error
02H	WD1015/WD2010 controller error
03H	Sector Buffer RAM data error
04H	WD1015/WD11C00A-22 Register access error
05H	WD1015 ROM checksum or RAM data error

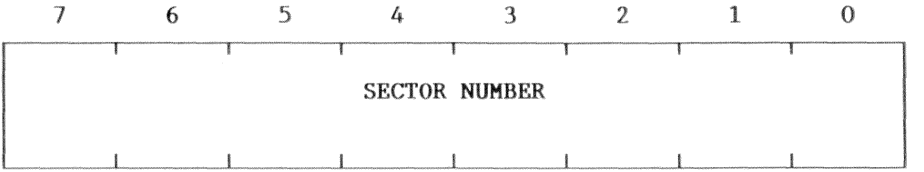
Sector Count Register (01F2H/0172H)



Bit	R/W	Description
7-0	R/W	The sector count for an operation A value of 00H indicates a 256 sector transfer.

For a read, write, or read verify command, this register specifies the number of sectors transferred. For the format command, this register specifies the number of sectors to format. During a multiple sector operation, after each sector is transferred or formatted, the hard disk controller decrements the sector count.

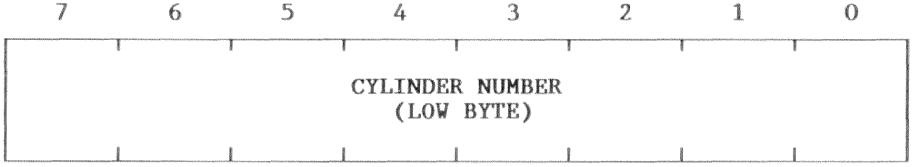
Sector Number Register (01F3H/0173H)



Bit	R/W	Description
7-0	R/W	The first sector for an operation

For a read, write, read verify, or format command, this register specifies the first sector of an operation. During multiple sector operations, after each sector is transferred or formatted, the hard disk controller increments the sector number.

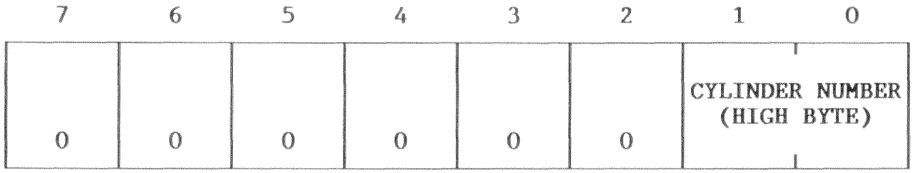
Cylinder Number Low Register (01F4H/0174H)



Bit	R/W	Description
7-0	R/W	Lower 8 bits of the 10-bit cylinder number

This register specifies the 8 least significant bits of the 10-bit cylinder number. The cylinder number high register specifies the 2 most significant bits.

Cylinder Number High Register (01F5H/0175H)



Bit	R/W	Description
7-2	R/W	Always 0
1-0	R/W	Upper 2 bits of the 10-bit cylinder number

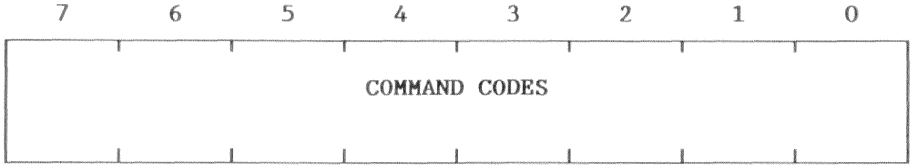
This register specifies the 2 most significant bits of the 10-bit cylinder number. The cylinder number low register specifies the 8 least significant bits.

SDH Register (01F6H/0176H)

7	6	5	4	3	2	1	0
ECC SELECT	0	DATA SIZE	DRIVE SELECT	HEAD SELECT			

Bit	R/W	Description
7	R/W	ECC SELECT 0 = CRC used to check the data field of a sector 1 = ECC used to check the data field of a sector (ROM BIOS requires ECC)
6	R/W	Always 0
5	R/W	DATA SIZE 0 = 256 byte records (read operations only) 1 = 512 byte records
4	R/W	DRIVE SELECT 0 = Hard disk drive 0 selected 1 = Hard disk drive 1 selected
3-0	R/W	HEAD SELECT The binary value in bits 3-0 selects the corresponding head. To use bit 3 of this field, the ENABLE-HEAD-SELECT-BIT-3 (bit 3 of the hard disk register) must equal 1. Otherwise, bit 3 is ineffective and bits 2-0 have a range of 0-7.

Command Register (01F7H/0177H)



Bit	R/W	Description
7-0	W	Hard disk controller command codes

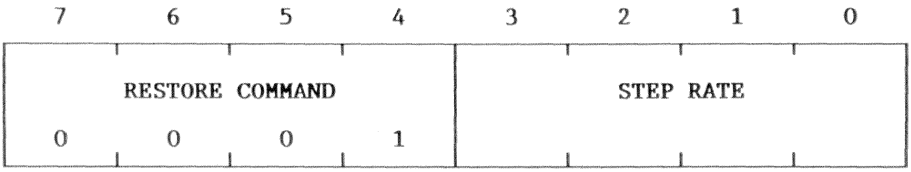
When command codes are written to this register, the hard disk controller executes the command immediately. If the hard disk controller is busy (status register bit 7 equals 1), command codes cannot be written to the command register. When the hard disk controller detects a command error condition, the hard disk controller aborts the command and sets the ABORTED COMMAND DETECT bit (error register bit 2). Some of the common command error conditions are:

- WRITE FAULT is active
- DRIVE READY is inactive
- SEEK COMPLETE is inactive
- Command code was invalid

The hard disk controller has the following commands:

- Restore
- Seek
- Read Sector
- Write Sector
- Format Track
- Read Verify
- Diagnose
- Set Parameter

Restore Command



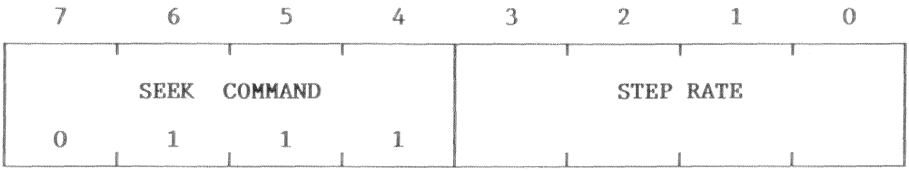
Bit	Description	
7-4	RESTORE COMMAND (Always 0001)	
3-0	STEP RATE	
	0000 = 35.0 μs	1000 = 4.0 ms
	0001 = 0.5 ms	1001 = 4.5 ms
	0010 = 1.0 ms	1010 = 5.0 ms
	0011 = 1.5 ms	1011 = 5.5 ms
	0100 = 2.0 ms	1100 = 6.0 ms
	0101 = 2.5 ms	1101 = 6.5 ms
	0110 = 3.0 ms	1110 = 3.2 μs
	0111 = 3.5 ms	1111 = 16.0 μs

Until a terminating condition occurs, the hard disk controller issues step pulses to the selected drive. The terminating conditions are:

- The TRACK 0 signal from the drive becomes active, which indicates a successful completion of the command. In this case, the hard disk controller sets SEEK COMPLETE (status register bit 4) to 1.
- The hard disk controller issues 2047 step pulses, which causes a TRACK 0 ERROR (error register bit 3).
- The DRIVE READY signal becomes inactive, which causes an ABORTED COMMAND DETECT (error register bit 2).
- The WRITE FAULT signal becomes active, which causes an ABORTED COMMAND DETECT (error register bit 2).

While a restore command is in progress, the SEEK COMPLETE signal from the drive controls the step rate. Bits 3-0 set the implied seek step rate. On termination of this command, the hard disk controller generates an interrupt to the CPU.

Seek Command



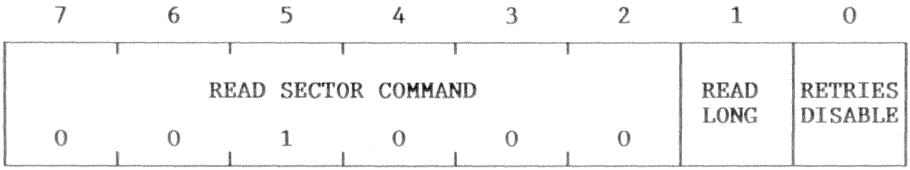
Bit	Description	
7-4	SEEK COMMAND (Always 0111)	
3-0	STEP RATE	
	0000 = 35.0 μs	1000 = 4.0 ms
	0001 = 0.5 ms	1001 = 4.5 ms
	0010 = 1.0 ms	1010 = 5.0 ms
	0011 = 1.5 ms	1011 = 5.5 ms
	0100 = 2.0 ms	1100 = 6.0 ms
	0101 = 2.5 ms	1101 = 6.5 ms
	0110 = 3.0 ms	1110 = 3.2 μs
	0111 = 3.5 ms	1111 = 16.0 μs

Until a terminating condition occurs, the hard disk controller issues step pulses to the selected drive. The terminating conditions are:

- The SEEK COMPLETE signal from the drive becomes active, which indicates a successful completion of the command. In this case, the hard disk controller sets SEEK COMPLETE (status register bit 4) to 1.
- The DRIVE READY signal becomes inactive, which causes an ABORTED COMMAND DETECT (error register bit 2).
- The WRITE FAULT signal becomes active, which causes an ABORTED COMMAND DETECT (error register bit 2).

The seek command moves the read/write heads to the cylinder specified by the 10-bit value in the registers cylinder low and cylinder high. Bits 3-0 set the implied seek step rate and control the step rate of the seek command. After the hard disk controller starts the seek, the hard disk controller generates an interrupt to the CPU. This interrupt indicates that the hard disk controller is ready to accept another command. It does not indicate that the seek is complete. To determine when the seek is complete, the application program or driver must monitor status register bit 4. When status register bit 4 equals 1, the seek is complete.

Read Sector Command



Bit	Description
7-2	READ SECTOR COMMAND (Always 001000)
1	READ LONG 0 = After transferring a sector of data, the hard disk controller calculates and checks the data field ECC. If a correctable error occurs, the hard disk controller sets CORRECTED DATA (status register bit 2). If an uncorrectable error occurs, the hard disk controller sets ECC ERROR (error register bit 6) and ERROR (status register bit 0). If an uncorrectable error occurs during a multiple sector read, the multiple sector read is terminated. 1 = The hard disk controller does not calculate the data field ECC. After transferring a sector of data, the hard disk controller transfers the four data field ECC bytes. The four ECC bytes are transferred one at a time in the high byte of the word. Until DRQ (status register bit 3) equals 1, an ECC byte is not ready for transfer. The hard disk controller requires at least 2 μs between ECC byte transfers.
0	RETRIES DISABLE 0 = Retries enabled 1 = Retries disabled

To read a sector from the hard disk, the hard disk controller must first find the sector. To find the correct sector, the hard disk controller selects the indicated head, performs any implied seek, and waits for the index pulse. On receipt of the index pulse, the hard disk controller begins scanning the track for a valid sector ID field. A valid sector ID field contains the cylinder, head, sector number, and size information that the hard disk controller is searching for. The hard disk controller counts each index pulse as an attempt to read a valid sector ID.

Bit	Description (Read Sector Command - cont.)
	If retries are enabled and the hard disk controller fails to read a valid ID field within ten attempts, the hard disk controller performs an auto-seek and an auto-scan. If the hard disk controller fails to read a valid ID field within an additional ten attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0). If retries are disabled and the hard disk controller fails to read a valid ID field within two attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0). Also, the hard disk controller does not perform an auto-seek or an auto-scan.

The read sector command transfers the specified number of sectors from the selected drive. If the heads are not positioned at the specified cylinder, the controller performs an implied seek to the proper cylinder. The step rate used is that of the most recently executed restore or seek command. Multiple sector reads can cross head and cylinder boundaries. If the DRIVE READY signal becomes inactive, or the WRITE FAULT signal becomes active, the hard disk controller terminates the command and sets ABORTED COMMAND DETECT (error register bit 2) and ERROR (status register bit 0).

When each sector is stored in the sector buffer and ready to be read by the CPU, the hard disk controller generates an interrupt request to the CPU. The hard disk controller does not generate an interrupt request to indicate command completion.

Write Sector Command

7	6	5	4	3	2	1	0
WRITE SECTOR COMMAND						WRITE LONG	RETRIES DISABLE
0	0	1	1	0	0		

Bit	Description
7-2	WRITE SECTOR COMMAND (Always 001100)
1	WRITE LONG 0 = The hard disk controller calculates the data field ECC and appends the ECC to the data field. 1 = The hard disk controller does not calculate the data field ECC. After accepting a sector of data, the hard disk controller accepts the four data field ECC bytes. The four ECC bytes are transferred one at a time in the high byte of the word size data register. Until DRQ (status register bit 3) equals 1, an ECC byte is not ready for transfer. The hard disk controller requires at least 2 μs between ECC byte transfers.
0	RETRIES DISABLE 0 = Retries enabled 1 = Retries disabled

To write a sector from the hard disk, the hard disk controller must first find the sector. To find the correct sector, the hard disk controller selects the indicated head, performs any implied seek, and waits for the index pulse. On receipt of the index pulse, the hard disk controller begins scanning the track for a valid sector ID field. A valid sector ID field contains the cylinder, head, sector number, and size information that the hard disk controller is searching for. The hard disk controller counts each index pulse as an attempt to read a valid sector ID.

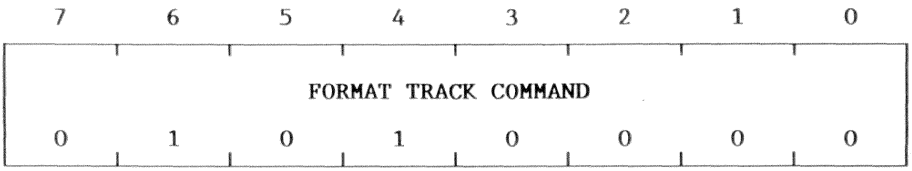
If retries are enabled and the hard disk controller fails to read a valid ID field within ten attempts, the hard disk controller performs an auto-seek and an auto-scan. If the hard disk controller fails to read a valid ID field within an additional ten attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0).

Bit	Description (Write Sector Command - cont.)
	If retries are disabled and the hard disk controller fails to read a valid ID field within two attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0). Also, the hard disk controller does not perform an auto-seek or an auto-scan.

The write sector command transfers the specified number of sectors to the selected drive. If the heads are not positioned at the specified cylinder, the controller performs an implied seek to the proper cylinder. The step rate used is that of the most recently executed restore or seek command. Multiple sector writes can cross head and cylinder boundaries. If the DRIVE READY signal becomes inactive, or the WRITE FAULT signal becomes active, the hard disk controller terminates the command and sets ABORTED COMMAND DETECT (error register bit 2) and ERROR (status register bit 0).

After the write sector command has been issued, immediately write the first sector of data to the data buffer (monitor DRQ, status register bit 3). When each sector is written to the hard disk, the hard disk controller generates an interrupt request to the CPU. The hard disk controller does not generate an interrupt request to indicate command completion.

Format Track Command



Bit	Description
7-0	FORMAT TRACK COMMAND (Always 01010000)

Using data from the internal registers and the sector buffer, the format track command formats a single track at the specified cylinder and head location.

Before formatting a hard disk, the SDH register must be loaded and a restore command must be issued. Assuming that a series of sequential format track commands are issued to format the hard disk, only one restore command is required. Prior to each format track command, the sector count register must be loaded with the number of sectors per track and the cylinder registers must be loaded with the cylinder number.

Instead of providing normal read/write data, the sector buffer provides sector header information. Also, the organization of this sector header information specifies the sector interleave. Table 12-3 lists the data (in memory order) that is required for an interleave factor of 2. The data transfer using this table is 256 words long. A bad block is specified by replacing the 00H (normal block mark) with an 80H (bad block mark). The sector interleave table is loaded into the sector buffer in the same manner as a sector write. That is, the format track command is issued and the sector buffer is loaded immediately (monitor DRQ, status register bit 3).

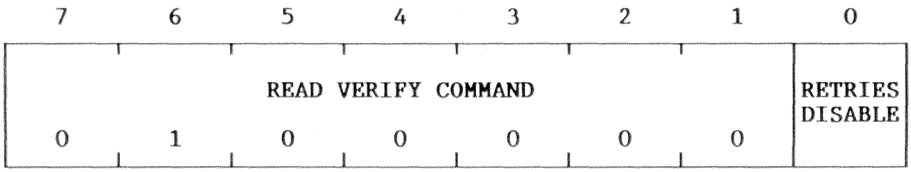
When a track has been formatted, the hard disk controller clears the sector buffer to zeros and issues a command completion interrupt. Following the next format track command, the sector buffer must be reloaded.

The hard disk controller does not produce any error reports for this command.

Table 12-3 Memory Image of a Sector Interleave Table

Offset	2-Byte Sequence	
	Sector Number (High Byte)	Sector Status (00H = Good, 80H = Bad) (Low Byte)
00H	01H	00H
02H	0AH	00H
04H	02H	00H
06H	0BH	00H
08H	03H	00H
0AH	0CH	00H
0CH	04H	00H
0EH	0DH	00H
10H	05H	00H
12H	0EH	00H
14H	06H	00H
16H	0FH	00H
18H	07H	00H
1AH	10H	00H
1CH	08H	00H
1EH	11H	00H
20H	09H	00H
22H	FFH	FFH
.	FFH	FFH
.	FFH	FFH
.	FFH	FFH
FFH	FFH	FFH

Read Verify Command



Bit	Description
7-1	READ VERIFY COMMAND (Always 0100000)
0	RETRIES DISABLE 0 = Retries enabled 1 = Retries disabled

To read a sector from the hard disk, the hard disk controller must first find the sector. To find the correct sector, the hard disk controller selects the indicated head, performs any implied seek, and waits for the index pulse. On receipt of the index pulse, the hard disk controller begins scanning the track for a valid sector ID field. A valid sector ID field contains the cylinder, head, sector number, and size information that the hard disk controller is searching for. The hard disk controller counts each index pulse as an attempt to read a valid sector ID.

If retries are enabled and the hard disk controller fails to read a valid ID field within ten attempts, the hard disk controller performs an auto-seek and an auto-scan. If the hard disk controller fails to read a valid ID field within an additional ten attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0).

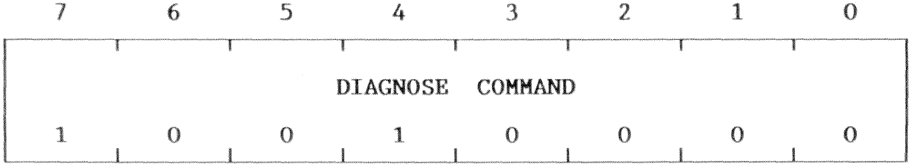
If retries are disabled and the hard disk controller fails to read a valid ID field within two attempts, the hard disk controller sets ID Not Found (error register bit 4) and ERROR (status register bit 0). Also, the hard disk controller does not perform an auto-seek or an auto-scan.

The read verify command scans the data in the specified number of sectors of the selected drive. As the hard disk controller scans the data of each sector, it calculates the ECC bytes and verifies that the ECC bytes calculated from the data and ECC bytes read from the disk agree. If the heads are not positioned at the specified cylinder, the controller performs an implied seek to the proper cylinder. The step rate used is that of the most recently executed restore or seek command. Multiple sector reads may cross head and cylinder boundaries. If the DRIVE READY signal becomes inactive, or the WRITE FAULT signal becomes active, the hard disk controller terminates the command and sets

ABORTED COMMAND DETECT (error register bit 2) and ERROR (status register bit 0).

To indicate command completion or an error condition, the hard disk controller generates an interrupt request to the CPU.

Diagnose Command



Bit	Description
7-0	DIAGNOSE COMMAND (Always 10010000)

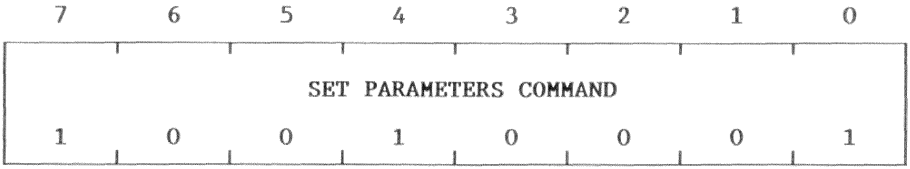
On receipt of this command, the hard disk controller executes a set of diagnostic tests. If a problem exists, the hard disk controller reports the results in the error register (see Table 12-4). The internal ROM, internal RAM, ECC circuitry, and data path circuitry are tested. When the diagnostic tests complete, the hard disk controller loads the appropriate code into the error register. To indicate command completion, the hard disk controller generates an interrupt request to the CPU.

For the diagnose command, the code in the error register is interpreted differently than normal error register encoding. Table 12-4 lists the diagnose command result codes.

Table 12-4 Hard Disk Controller Diagnostic Result Codes

Value	Meaning
01H	No error
02H	WD1015/WD2010 controller error
03H	Sector Buffer RAM data error
04H	WD1015/WD11C00A-22 Register access error
05H	WD1015 ROM checksum or RAM data error

Set Parameters Command



Bit	Description
7-0	SET PARAMETERS COMMAND (Always 10010001)

This command sets the drive parameters for the maximum number of heads and sectors per track. Prior to issuing this command, the drive selection must be specified in the Sector Size/Drive Select/Head Select task register, and the Sector Count Register must be set. This command must be issued before any multiple sector operations are performed. To indicate command completion, the hard disk controller generates an interrupt request to the CPU.

Status Register (01F7H/0177H)

7	6	5	4	3	2	1	0
BUSY	DRIVE READY	WRITE FAULT	SEEK STATUS	DRQ	CD	INDEX	ERROR

Bit	R/W	Description
7	R	BUSY 0 = Hard disk controller is ready to accept commands. The status and error registers contain valid data. 1 = Hard disk controller is busy. The error register contains invalid data. Only this status register bit is valid. This bit must be polled before reading any register.
6	R	DRIVE READY 0 = Drive is not ready. Read, write, and seek commands are inhibited. 1 = Drive is ready. If bit 4 equals 1, read, write, and seek commands are possible.
5	R	WRITE FAULT 0 = No error 1 = Improper operation of the drive. All commands will be terminated with an aborted command error.
4	R	SEEK STATUS - Seek Complete 0 = Seek operation, direct or implied, is incomplete. 1 = Seek operation is complete (read/write heads are in position).
3	R	DRQ - Data Request 0 = Do not read or write the sector buffer 1 = Read or write the sector buffer as indicated by the last command issued.
2	R	CD - Corrected Data 0 = No error 1 = The sector read from the drive resulted in a correctable ECC error. Correctable errors do not cause multiple sector transfers to terminate.

Bit	R/W	Description (Status Register - cont.)
1	R	INDEX 0 = Index not in position and not detected 1 = Index in position and detected
0	R	ERROR 0 = No error 1 = A nonrecoverable error has occurred. The error register contains the error status. The next command issued to the hard disk controller, clears this bit. If the hard disk controller sets this bit during a multiple sector transfer, the transfer is terminated.

Alternate Status Register (03F6H/0376H)

This read-only register duplicates the status register (01F7H/0177H).

Hard Disk Register (03F6H/0376H)

7	6	5	4	3	2	1	0
0	0	0	0	ENABLE HEAD SELECT BIT 3	RESET	INTRPT ENABLE	0

Bit	R/W	Description
7-4	W	Always 0
3	W	ENABLE HEAD SELECT BIT 3 0 = Reduced write current enabled 1 = Head Select Bit 3 (SDH register bit 3) enabled. See the SDH register description.
2	W	RESET 0 = Hard disk controller enabled. 1 = When set, the hard disk controller enters the reset state and remains there until this bit is cleared. This bit must stay set for a minimum of 5 μs. When this bit is cleared, the hard disk controller performs a set of power-up diagnostic tests and the busy bit (status register bit 7) remains set until tests are completed.
1	W	INTRPT ENABLE - Interrupt Enable 0 = Hard disk controller interrupt buffer is enabled. Setting this bit does not clear the hard disk controller interrupt output. When this bit is cleared, any pending interrupts will occur. 1 = Hard disk controller interrupt buffer is disabled. This bit enables or disables a buffer between the hard disk controller output and the peripheral interrupt controller input.
0	W	Always 0

Digital Input Register (03F7H/0377H)

7	6	5	4	3	2	1	0
RESERVE	WRITE GATE SIGNAL	1	HEAD SELECT SIGNAL 2	HEAD SELECT SIGNAL 1	HEAD SELECT SIGNAL 0	DRIVE SELECT SIGNAL 1	DRIVE SELECT SIGNAL 0

Bit	R/W	Description
7	R	Reserved (Value undefined)
6	R	WRITE GATE SIGNAL 0 = Write gate signal is active. 1 = Write gate signal is inactive.
5	R	Always 1
4	R	HEAD SELECT SIGNAL 2 0 = Head select signal 2 is active. 1 = Head select signal 2 is inactive.
3	R	HEAD SELECT SIGNAL 1 0 = Head select signal 1 is active. 1 = Head select signal 1 is inactive.
2	R	HEAD SELECT SIGNAL 0 0 = Head select signal 0 is active. 1 = Head select signal 0 is inactive.
1	R	DRIVE SELECT SIGNAL 1 0 = Drive select signal 1 is active. 1 = Drive select signal 1 is inactive.
0	R	DRIVE SELECT SIGNAL 0 0 = Drive select signal 0 is active. 1 = Drive select signal 0 is inactive.

Programming Example

The following programming example demonstrates:

- Initializing the hard disk controller
- Recalibrating the hard disk drive
- Seeking to a track
- Hard formatting a hard disk

CAUTION

Improper programming or improper operation of this device can cause the VAXmate workstation to malfunction. The scope of the programming example is limited to the context provided in this manual. No other use is intended.

```
#include "kyb.h"
#include "example.h"

/*********************************************************************/
/* define constants used in hard disk controller example             */
/*********************************************************************/
#define PRI_BASE    0x01f0  /* primary base address of controller regs */
#define ALT_BASE    0x0170/* alternate base address of controller regs */

#define HDR_ASR     0x03f6            /* address of hard disk register */
                                      /* and alternate status register */
#define DIG_INP     0x03f7        /* address of digital input register */

/* define constant divisor for write precompensation cylinder */

#define WPC_DIV     0x04

/* define bit values for error status register */

#define ERR_BBD     0x80                        /* bad block detected */
#define ERR_DFE     0x40              /* CRC/ECC error in data field */
#define ERR_ID      0x10        /* could not locate correct ID field */
#define ERR_ABR     0x04                          /* command aborted */
#define ERR_TZE     0x02                         /* track zero error */
#define ERR_DAMNF   0x01               /* data address mark not found */

/* define bit values for hard disk register */

#define HEAD_SEL3   0x04             /* enables third head select bit */
#define RESET_HDC   0x02               /* resets hard disk controller */
#define INT_ENA     0x01             /* enables hdc interrupts to CPU */

/* define bit values for sdh register */

#define SDH_ECC     0x80            /* select ecc mode for data field */

#define SDH_256     0x00                          /* sector size = 256 */
#define SDH_512     0x20                          /* sector size = 512 */
```

```
/* define bit values for status register */

#define STAT_BSY    0x80                              /* hdc busy */
#define STAT_RDY    0x40                              /* drdy pin */
#define STAT_WF     0x20                          /* write fault */
#define STAT_SC     0x10                         /* seek complete */
#define STAT_DRQ    0x08                          /* data request */
#define STAT_DWC    0x04                    /* data was corrected */
#define STAT_ERR    0x01                  /* nonrecoverable error */

/* define bit values for command register */

#define SR_35US     0x00                       /* step rate = 35 us */
#define SR_0_5MS    0x01                      /* step rate = 0.5 ms */
#define SR_1_0MS    0x02                      /* step rate = 1.0 ms */
#define SR_1_5MS    0x03                      /* step rate = 1.5 ms */
#define SR_2_0MS    0x04                      /* step rate = 2.0 ms */
#define SR_2_5MS    0x05                      /* step rate = 2.5 ms */
#define SR_3_0MS    0x06                      /* step rate = 3.0 ms */
#define SR_3_5MS    0x07                      /* step rate = 3.5 ms */
#define SR_4_0MS    0x08                      /* step rate = 4.0 ms */
#define SR_4_5MS    0x09                      /* step rate = 4.5 ms */
#define SR_5_0MS    0x0a                      /* step rate = 5.0 ms */
#define SR_5_5MS    0x0b                      /* step rate = 5.5 ms */
#define SR_6_0MS    0x0c                      /* step rate = 6.0 ms */
#define SR_6_5MS    0x0d                      /* step rate = 6.5 ms */
#define SR_3_2US    0x0e                      /* step rate = 3.2 us */
#define SR_1_6US    0x0f                      /* step rate = 1.6 us */

#define CMD_IC      0x08                        /* interrupt control */
#define CMD_MSF     0x04                   /* multiple sector flag */
#define CMD_LM      0x02                /* long mode read and write */
#define CMD_TRY     0x01                        /* disable retries */

#define CMD_REST    0x10                        /* restore command */
#define CMD_SEEK    0x70                           /* seek command */
#define CMD_READ    0x20                           /* read command */
#define CMD_WRITE   0x30                          /* write command */
#define CMD_SCAN    0x40                 /* read/verify command */
#define CMD_FRMAT   0x50                          /* format command */
#define CMD_CC      0x08              /* compute correction command */
#define CMD_PARM    0x91                   /* set parameter command */
```

```
/*********************************************************************/
/* declare structures used in hard disk controller example           */
/*********************************************************************/

typedef struct
{
  unsigned char sec_buf;                          /* (R/W) sector buffer */
  unsigned char err_wpc;                             /* (R) error reg */
                                  /* (W) write precompensation cylinder */
  unsigned char sc;                                /* (R/W) sector count */
  unsigned char sn;                                /* (R/W) sector number */
  unsigned char cyl_low;       /* (R/W) lower 8 bits of cylinder number */
  unsigned char cyl_hi;        /* (R/W) upper 2 bits of cylinder number */
  unsigned char sdh;                               /* (R/W) sdh register */
  unsigned char csr;                 /* (R) status reg, (W) command reg */
} HDC;

typedef struct
{
  HDC *base;     /* primary or alternate base address of controller reg */
  int busy;                                                  /* busy flag */
  int retry;                                               /* retry count */
  unsigned int cyl;                                     /* cylinder number */
  unsigned char last_cmd;                     /* last command sent to hdc */
  unsigned char ecm;                           /* error correction method */
  unsigned char srt;                                    /* step rate time */
  unsigned char ds;                                  /* drive select 0 - 3 */
  unsigned char hs;                                /* selected head number */
  unsigned char sc;                                     /* sector count */
  unsigned char sn;                                    /* sector number */
  unsigned char eot;                                    /* end of track */
  unsigned char fgpl;                               /* format gap length */
  unsigned char nd;                                     /* non-DMA mode */
  unsigned char ss;                                     /* sector size */
  unsigned char t;                                       /* retry flag */
} HDC_CMD;
```

```
typedef struct
{
  unsigned int cc;                                  /* cylinder count */
  unsigned int wpc;                  /* write precompensation cylinder */
  unsigned int hc;                                    /* head count */
  unsigned int spt;                                /* sectors per track */
} HDD;

typedef struct
{
  unsigned int cyl;                                 /* cylinder number */
  unsigned char error;                             /* error code/status */
  unsigned char sc;                                   /* sector count */
  unsigned char sn;                                  /* sector number */
  unsigned char sdh;                                  /* sdh register */
  unsigned char status;                             /* status register */
} HDC_RESULT;



/************************************************************************/
/* declare space for hdc parameter data                                 */
/************************************************************************/

HDD hdd[2] =                                      /* hard disk descriptors */
{
  { 615, 300, 4, 17 },
  { 615, 300, 4, 17 },
};
HDC_CMD hdc_cmd;                                     /* command data */
HDC_RESULT hdc_result;                               /* result data */
int hdc_buff[256];                                 /* hdc I/O buffer */

/************************************************************************/
/* hds() - select the desired drive                                     */
/************************************************************************/

hds()  /* select error correction method, sector size, drive, and head */
{
HDC *phdc = hdc_cmd.base;           /* pointer to controller registers */

  outp(&phdc->sdh, hdc_cmd.ecm | hdc_cmd.ss | (hdc_cmd.ds << 4) | hdc_cmd.hs);
}
```

```
/*************************************************************************/
/* wait_hdc() - wait for hard disk command to complete                   */
/*************************************************************************/

wait_hdc()
{
HDC *phdc = hdc_cmd.base;           /* pointer to controller registers */

  while(hdc_cmd.busy)                    /* wait until interrupt occurs */
    ;                  /* busy bit should be clear, but just in case, */
  while(inp(&phdc->csr) & STAT_BSY)    /* wait until busy bit is clear */
    ;
  if(inp(&phdc->csr) & STAT_ERR)                 /* if error condition */
    hdc_result.error = inp(&phdc->err_wpc);     /* read error register */
  else hdc_result.error = 0;                          /* mark no error */

  switch(hdc_cmd.last_cmd)             /* discover last command issued */
  {
    case CMD_REST :
      while((inp(&phdc->csr) & (STAT_RDY | STAT_SC)) != (STAT_RDY | STAT_SC))
        ;
      break;

    case CMD_SEEK :
      while((inp(&phdc->csr) & (STAT_RDY | STAT_SC)) != (STAT_RDY | STAT_SC))
        ;
      break;

    case CMD_FRMAT:
     break;

    default :
      break;
  }
  hdc_result.sc = inp(&phdc->sc);         /* read sector count register */
  hdc_result.sn = inp(&phdc->sn);        /* read sector number register */
  hdc_result.cyl = inp(&phdc->cyl_low);           /* low 8 bits of cyl */
  hdc_result.cyl |= inp(&phdc->cyl_hi) << 8;     /* high 2 bits of cyl */
  hdc_result.sdh = inp(&phdc->sdh);               /* read sdh register */
  hdc_result.status = inp(&phdc->csr);          /* read status register */
  dsp_hdc_stat();                              /* display result status */
}
```

```
/*************************************************************************/
/* hdc_issue() - issue all hdc commands                                  */
/*************************************************************************/

hdc_issue(cmd)

int cmd;                                          /* desired command */

{
HDC *phdc = hdc_cmd.base;
char oline[20];

  hdc_cmd.last_cmd = cmd;                  /* set last command issued */
  switch(cmd)
  {
    case CMD_REST:                                  /* recalibrate ? */
      hdc_cmd.cyl = 0;
      hdc_cmd.hs = 0;
      cmd |= hdc_cmd.srt;                        /* restore command */
      break;

    case CMD_SEEK:                                        /* seek ? */
      cmd |= hdc_cmd.srt;                       /* seek command */
      break;

    case CMD_SCAN:                                    /* read ID ? */
      cmd |= hdc_cmd.t;                         /* scan id command */
      break;

    case CMD_READ:                                   /* read data ? */
    case CMD_WRITE:                                  /* write data ? */
      break;

    default:
      break;
  }
  outp(&phdc->err_wpc, hdd[hdc_cmd.ds].wpc >> 2); /* write the wpc reg */
  outp(&phdc->sc, hdc_cmd.sc);                 /* write the sector count */
  outp(&phdc->sn, hdc_cmd.sn);                /* write the sector number */
  outp(&phdc->cyl_low, hdc_cmd.cyl);                 /* write 8 low bits */
  outp(&phdc->cyl_hi, hdc_cmd.cyl >> 8);            /* write 2 high bits */
  hds();                                    /* select appropriate drive */
  outp(&phdc->csr, cmd);                             /* write the command */
  hdc_cmd.busy = TRUE;
}
```

```
/*************************************************************************/
/* hdc_init() - initialize hard disk controller and interrupt vector   */
/*************************************************************************/

hdc_init()
{

  outp(HDR_ASR, RESET_HDC);                          /* reset the hdc */
                              /* must be reset for minimum of 5 us */
  iv_init(0x76);                /* initialize the interrupt vector */
  imask(1, 6, ON);                                  /* enable PIC input */
  outp(HDR_ASR, INT_ENA);          /* hdc not reset and allow interrupt */
}

/*************************************************************************/
/* hdc_rest() - restore hard disk controller and interrupt vector      */
/*************************************************************************/

hdc_rest()
{
  outp(HDR_ASR, RESET_HDC);                          /* reset the hdc */
  imask(1, 6, OFF);                                /* disable PIC input */
  iv_rest(0x76);                       /* restore the interrupt vector */
  outp(HDR_ASR, INT_ENA);          /* hdc not reset and allow interrupt */
}

/*************************************************************************/
/* hdc_int_hand() - hdc interrupt handler                               */
/*************************************************************************/

hdc_int_hand()
{
HDC *phdc = hdc_cmd.base;            /* pointer to controller registers */

  hdc_result.status = inp(&phdc->csr);          /* read status register */
  hdc_cmd.busy = FALSE;                              /* no longer busy */
  eoi(1);
}
```

```
/*************************************************************************/
/* hdc() - execute hard disk controller examples                         */
/*************************************************************************/

hdc()
{
static MESSAGE mhdc[] =                                    /* hdc menu */
{
  {  3, 27, "Hard Disk Controller Example" },
  {  5, 27, "F1.  Recalibrate" },
  {  6, 27, "F2.  Head +" },
  {  7, 27, "F3.  Head -" },
  {  8, 27, "F4.  Seek +" },
  {  9, 27, "F5.  Seek -" },
  { 10, 27, "F6.  Read ID" },
  { 11, 27, "F7.  ** Format Disk ** (Erases data)" },
  { 12, 27, "F10. Return to Main menu" },
  {  0,  0,  0 },
};

char line[512];                                  /* to hold input line */
char oline[512];                                /* to hold output line */
unsigned int cyl;                                     /* loop control */
unsigned int head;                                    /* loop control */
HDC *phdc;
int *pi;
int i;

  disp_menu(mhdc);                              /* display the hdc menu */
  hdc_cmd.base = (HDC *)PRI_BASE;   /* controller addressed at primary */
  hdc_cmd.ecm = SDH_ECC;
  hdc_cmd.ss = SDH_512;
  hdc_cmd.ds = 0;
  hdc_cmd.hs = hdd[0].hc;
  hdc_cmd.sc = hdd[0].spt;
  hdc_cmd.srt = SR_1_5MS;                          /* assign step rate */
  phdc = hdc_cmd.base;
  hdc_issue(CMD_PARM);                               /* set parameters */
  wait_hdc();
  hdc_cmd.cyl = hdc_result.cyl;
  dsp_hdc_stat();
  hdc_cmd.hs = 0;
```

```
line[0] = 0;                                        /* null terminated */
while(1)                                           /* forever (see F10) */
{
  line[0] = get_fkey();      /* get a function key for menu selection */
  switch(line[0])                          /* determine menu selection */
  {
    case F1:                                                /* recalibrate */
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      hdc_issue(CMD_REST);
      wait_hdc();                          /* wait for command complete */
      break;

    case F2:                                             /* head select + 1 */
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      if(hdc_cmd.hs < hdd[hdc_cmd.ds].hc) hdc_cmd.hs++;
      if(hdc_cmd.hs == hdd[hdc_cmd.ds].hc) hdc_cmd.hs--;
      hdc_issue(CMD_SEEK);
      wait_hdc();                          /* wait for command complete */
      break;

    case F3:                                             /* head select -1 */
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      if(hdc_cmd.hs) hdc_cmd.hs--;
      hdc_issue(CMD_SEEK);
      wait_hdc();                          /* wait for command complete */
      break;

    case F4:                                           /* seek + 1 cylinder */
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      if(hdc_cmd.cyl < hdd[hdc_cmd.ds].cc) hdc_cmd.cyl++;
      if(hdc_cmd.cyl == hdd[hdc_cmd.ds].cc) hdc_cmd.cyl--;
      hdc_issue(CMD_SEEK);
      wait_hdc();                          /* wait for command complete */
      break;
```

```
case F5:                                        /* seek - 1 cylinder */
  hdc_cmd.ecm = SDH_ECC;
  hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
  hdc_cmd.sn = 0;
  if(hdc_cmd.cyl) hdc_cmd.cyl--;
  hdc_issue(CMD_SEEK);
  wait_hdc();                          /* wait for command complete */
  break;

case F6:                                                  /* scan id */
  hdc_cmd.ecm = SDH_ECC;
  hdc_cmd.sc = 1;
  for(i = 1; i <= hdd[hdc_cmd.ds].spt; i++)
  {
    hdc_cmd.sn = i;
    hdc_issue(CMD_SCAN);
    wait_hdc();                        /* wait for command complete */
  }
  break;

case F7:                                        /* hard format disk */
  hdc_issue(CMD_REST);
  wait_hdc();                          /* wait for command complete */
  interleave(hdc_buff, 256, 17, 2, 0xffff);
  for(cyl = 0; cyl < hdd[hdc_cmd.ds].cc; cyl++) /* all cylinders */
  {
    hdc_cmd.cyl = cyl;                          /* current cylinder */
    for(head = 0; head < hdd[hdc_cmd.ds].hc; head++)/* all heads */
    {
      hdc_cmd.hs = head;
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      hdc_issue(CMD_SEEK);             /* seek to current cylinder */
      wait_hdc();                      /* wait for command complete */
      hdc_cmd.ecm = SDH_ECC;
      hdc_cmd.sc = hdd[hdc_cmd.ds].spt;
      hdc_cmd.sn = 0;
      hdc_issue(CMD_FRMAT);
      while(!(inp(&phdc->csr) & STAT_DRQ))
        ;
      for(i = 0; i < 256; i++)
        outw(&phdc->sec_buf, hdc_buff[i]);
      wait_hdc();                      /* wait for command complete */
    }
  }
    break;
```

```
      case F8:
        interleave(hdc_buff, 256, 17, 2, 0xffff);
        for(cyl = 0, head = 0; head < 30; head++)
        {
          sprintf(oline, "%04x ", hdc_buff[head]);
          disp_str(13 + (head / 10), (head % 10) * 5, oline);
        }
        break;

      case F10:                          /* return to caller (main menu) */
        return;
    }
  }
}

dsp_hdc_stat()
{
HDC_RESULT *pres = &hdc_result;
char oline[50];

  sprintf(oline, "Error Reg     : 0x%02x", pres->error);
  disp_str(21,  1, oline);
  sprintf(oline, "Sector Count  : 0x%02x", pres->sc);
  disp_str(21, 41, oline);
  sprintf(oline, "Sector Number: 0x%02x", pres->sn);
  disp_str(22,  1, oline);
  sprintf(oline, "Cylinder Num  : 0x%04x", pres->cyl);
  disp_str(22, 41, oline);
  sprintf(oline, "SDH Register  : 0x%02x", pres->sdh);
  disp_str(23,  1, oline);
  sprintf(oline, "Status Reg    : 0x%02x", pres->status);
  disp_str(23, 41, oline);
  sprintf(oline, "Last command  : 0x%02x", hdc_cmd.last_cmd);
  disp_str(24,  1, oline);
}
```

```
interleave(pb, size, cnt, il, fill)

int *pb;                                        /* pointer to buffer */
int size;                                          /* size of buffer */
int cnt;                                        /* number of sectors */
int il;                                              /* interleave:1 */
int fill;                                               /* fill byte */

{

int *pi;                                                /* temp pointer */
int *pf;                                               /* start of fill */
int x, y;                                               /* temp counter */

  x = size - cnt;                                          /* fill size */
  pf = pi = pb + cnt;                                   /* offset to fill */
  while(x--) *pi++ = fill;                              /* set fill bytes */
  for(x = 1; x <= cnt; x++)
  {
    if(pi > pf) pi = pb++;
    *pi++ = x << 8;
    for(y = 1; y < il; y++) pi++;
  }

}
```

Chapter 13
Network Hardware Interface

Introduction to the LANCE

The network hardware interface is located on the VAXmate workstation I/O board. When the back of the VAXmate workstation is open and no options are plugged in, the I/O board is visible.

VAXmate workstations may contain different versions of the LANCE, which exhibit subtle differences on large networks with high levels of traffic. The different versions of hardware require special treatment by software. Digital Equipment Corporation recommends that you avoid programming the hardware interface directly. Instead, applications should use the MS-Network session level interface, DECnet-DOS session level interface, and the Data Link interface to access the network. For additionl information about these software interfaces, see Chapter 18 in this manual.

The remainder of this chapter discusses the following topics:

- Functional description of the Network Hardware Interface
- Programming sequence
- Register descriptions
- Physical description
- Physical address field
- Logical address filter field
- Buffer management
- Receive descriptor rings
- Transmit descriptor rings
- Network interface external interconnect
- Network Interface system bus interconnect

Additional Source of Information

Additional information about the LANCE can be found in the Advanced Micro Devices document:

- *MOS Microprocessors and Peripherals 1985 Data Book*

Functional Description of the Network Hardware Interface

The network hardware interface is connected to the system bus. The interface contains address, data, and control lines capable of handling the controller functions. The interface is part of the standard communications and I/O functions of the VAXmate workstation.

The Network Interface (NI) consists of three main integrated circuits. The hardware also includes a small number of discrete components, system bus interfacing devices, and a female BNC connector mounted directly on VAXmate I/O module printed circuit board. The BNC connector connects the Network Interface to the network.

The three integrated circuits in the NI are the:

- Coax Transceiver Interface (CTI)
- Serial Interface Adapter (SIA)
- Local Area Network Controller for Ethernet (LANCE)

The Coax Transceiver Interface

The CTI interfaces to the coaxial cable by a BNC connector. The CTI performs transmit, receive, and collision detect functions for the network controller. The CTI is electrically isolated from the other devices as required by IEEE 802.3 specifications.

The Serial Interface Adapter

The SIA interfaces to the CTI through an isolation transformer. The SIA performs manchester phase encoding and decoding of the data transferred over the network. The SIA also interfaces with the LANCE. The SIA filters noise and interprets collisions for the LANCE circuit.

The Local Area Network Controller

The LANCE converts data beteen the network serial format and the system byte-wide format. The LANCE is the primary interface between the network hardware and the rest of the VAXmate workstation.

In receive mode, the LANCE:

1. Receives information from the SIA.
2. Converts the serial network bit stream into a parallel (8-bit wide) byte stream.
3. Strips the Ethernet preamble and synchronization pattern.
4. Checks and removes the CRC bits.
5. Uses direct memory access and a 24-bit-wide physical address to place the information in memory located in the CPU address space.

In transmit mode, the LANCE:

1. Uses DMA to read data from system memory.
2. Converts the data to a serial bit stream.
3. Adds a preamble and sync pattern.
4. Calculates and adds the CRC at the end of the data packet.
5. Passes the data packet to the SIA for transmission on the ThinWire Ethernet.

Programming the LANCE

This section defines the control registers, status registers, and the data structures that are used to program the hardware interface.

The LANCE is controlled by a set of four control and status registers (CSRs) and three data structures. The CSRs are accessible within the CPU I/O address space and the data structures located in the CPU memory address space. After the LANCE is enabled, it acquires additional operating parameters by accessing the data structures. The LANCE accesses system memory through bus arbitration between the LANCE, CPU, memory refresh controller, and the DMA controller.

After the LANCE is programmed, it independently manages the data structures and transfers data packets on the ThinWire Ethernet.

The three data structures accessed by the LANCE are:

- Initialization Block
- Receive and Transmit Descriptor Rings
- Data Buffers

Initialization Block

The initialization block is 12 contiguous words of memory starting on a word boundary. The controlling program stores the operating parameters in the initialization block. To acquire the operating parameters, which are necessary for device operation, the LANCE reads the initialization block. The initialization block contains the:

- Mode of Operation
- Physical Address
- Logical Address Mask
- Location of Receive and Transmit Descriptor Rings
- Number of Entries in Receive and Transmit Descriptor Rings

Receive and Transmit Descriptor Rings

The receive and transmit descriptor rings are two ring structures: one for incoming and the other for outgoing packets. Each entry in the rings is four words, and must start on a quadword boundary. The descriptor rings contain the buffer address, length, and status.

Data Buffers

Data buffers are contiguous portions of memory reserved for buffering packets. Data buffers can begin on arbitrary byte boundaries. Entries in the Receive and Transmit Descriptor Rings point to the data buffers.

Programming Sequence

The general programming sequence of the LANCE is:

1. Load CSR1 and CSR2, which identifies the location of an initialization block in memory.
2. Load CSR3, which sets the operating mode of the LANCE.
3. Set the INIT and STRT bits in CSR0, which causes the LANCE to load the information from the initialization block and to access the descriptor rings.

NOTE
The ThinWire Ethernet interface conforms to the IEEE 802.3 specification, which requires a frequency accuracy of $\pm 0.01\%$. The crystal oscillator in the VAXmate network hardware requires 1 Ms after power on to achieve $\pm 10\%$ accuracy, and 10 seconds to achieve $\pm 0.01\%$ accuracy. Therefore, the network interface is considered operational 10 seconds after power-on of the VAXmate workstation.

On powerup, the VAXmate diagnostics ensure the required 10 second settling time.

Register Description

The Network Interface uses three physical I/O ports as registers: two in the LANCE, and one in the interface logic. The registers in the LANCE comprise five logical registers, for a total of six logical registers in the Network Interface (NI).

The LANCE internal CSRs are accessed through its two bus addressable ports, a register address port (RAP), and a register data port (RDP). The internal CSRs are read (or written) in a two step operation. The address of the CSR is written into the address port (RAP). Then the data is read from (or written into) the selected CSR through the RDP. Once written, the address in RAP remains unchanged until rewritten.

Table 13-1 describes the registers and their addressing.

Table 13-1 Network Interface Registers

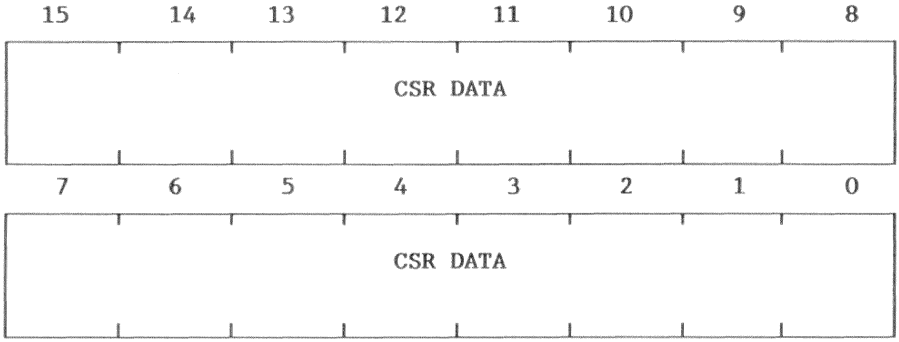
Primary I/O Address	Alias I/O Address	R/W	Register Width	Description
0C60H	C064H	R/W	16-bit	LANCE Register Data Port (RDP)
0C62H	C066H	R/W	16-bit	LANCE Register Address Port (RAP)
RAP = 0		R/W	16-bit	LANCE CSR 0
RAP = 1		R/W	16-bit	LANCE CSR 1
RAP = 2		R/W	16-bit	LANCE CSR 2
RAP = 3		R/W	16-bit	LANCE CSR 3
0C68H	0C69H-0C6FH	W	8-bit	NI CSR

The logical address is the contents of RAP bits 1-0, and applies to references to LANCE internal CSR registers 3-0. These registers are physically addressed at the LANCE RDP address on the VAXmate I/O bus, and individually selected by the contents of LANCE RAP bits 1-0.

The VAXmate address decode logic creates alias I/O port addresses for certain registers. The alias I/O addresses are listed in Table 13-1.

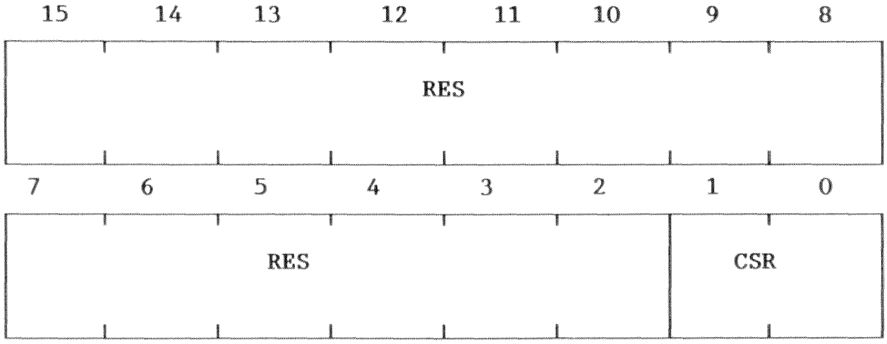
> **NOTE**
> Because future revisions of hardware may not implement these aliases, programs should not use the alias addresses.

Register Data Port (RDP)



Bit	R/W	Description
15-0	R	CSR DATA Writing data into RDP writes the data into the CSR selected in RAP. Reading the data from RDP reads the data from the CSR selected in RAP. CSR1, CSR2, and CSR3 are accessible only when the STOP bit of CSR0 is set. If the STOP bit is not set while attempting to access CSR1, CSR2, or CSR3, the LANCE returns READY, but a read operation returns undefined data and a write operation is ignored.

Register Address Port (RAP)



Bit	R/W	Description
15-2		RES - Reserved (always 0)
1-0	R/W	CSR - Control/Status Register Select 0 = CSR0 accessed through RDP 1 = CSR1 accessed through RDP 2 = CSR2 accessed through RDP 3 = CSR3 accessed through RDP

The register address port is cleared by STOP or Bus RESET.

Control And Status Register 0 (RAP = 0)

15	14	13	12	11	10	9	8
ERR	BABL	CERR	MISS	MERR	RINT	TINT	IDON
7	6	5	4	3	2	1	0
INTR	INEA	RXON	TXON	TDMD	STOP	STRT	INIT

Bit	R/W	Description
15	R	ERR - Error 0 = The four bits BABL, CERR, MISS and MERR are all equal to 0. 1 = One or more of the four bits, BABL, CERR, MISS, or MERR are equal to 1.
14		BABL - Babble
	R	0 = Less than 1519 bytes of data have been transmitted. 1 = 1519 bytes of data, or more, have been transmitted. BABL indicates a transmitter error, caused by the transmitter being on longer than the time required to send the maximum length packet. The LANCE transmits data until the transmit buffer byte count equals zero. The LANCE assumes the Ethernet transmission is limited by the physical channel interface. If INEA = 1, an interrupt is generated when BABL is set.
	W	0 = No effect 1 = Clears this bit Bable is read/clear only. BABLE is cleared by Bus RESET or setting the STOP bit.

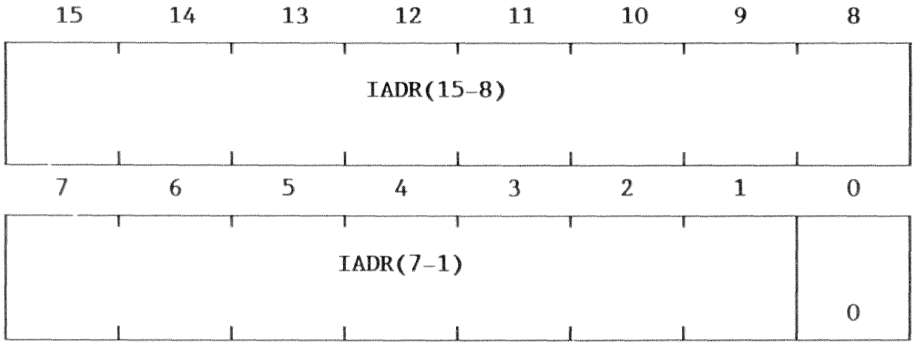
Bit	R/W	Description (CSR0 - cont.)
13		CERR - Collision Error
	R	0 = No collision error 1 = Collision error
		The collision input failed to activate within 2 μs after a transmission, started by the controller, was completed. Collision after transmission is a test feature of the transceiver.
	W	0 = No effect 1 = Clears this bit
		CERR is READ/CLEAR ONLY
		CERR is cleared by Bus RESET or setting the STOP bit.
12		MISS - Missed Packet
	R	0 = Receiver owns a receive buffer or the SILO has not overflowed 1 = The receiver loses a packet because it does not own a receive buffer and the SILO has overflowed, indicating loss of data. SILO overflow is not reported because there is no receive ring entry in which to write the status.
	W	0 = No effect 1 = Clears the bit
		If INEA equals 1 and MISS equals 1, an interrupt is generated.
		MISS is READ/CLEAR ONLY. MISS is cleared by Bus RESET or setting the STOP bit.
11		MERR - Memory Error
	R	0 = No memory error 1 = LANCE is the Bus Master and has not received READY within 25.6 μs after asserting the address on the DAL lines. When a Memory Error is detected, the receiver and transmitter are turned off and an interrupt is generated if INEA = 1.
	W	0 = No effect 1 = Clears the bit
		MERR is READ/CLEAR ONLY. MERR is cleared by Bus RESET or setting the STOP bit.

Bit	R/W	Description (CSR0 - cont.)
10		RINT - Receiver Interrupt
	R	0 = No receiver interrupt 1 = Receiver interrupt
	W	0 = No effect 1 = Clears the bit
		RINT = 1 when the status for an entry in the Receive Descriptor Ring is updated.
		If INEA = 1, an interrupt is generated when RINT is set.
		RINT is READ/CLEAR ONLY RINT is cleared by Bus RESET or setting the STOP bit.
9		TINT - Transmitter Interrupt
	R	0 = No transmitter interrupt 1 = When the status for an entry in the Transmit Descriptor Ring is updated.
	W	0 = No effect 1 = Clears the bit
		If INEA = 1, an interrupt is generated when TINT is set.
		TINT is READ/CLEAR ONLY
		TINT is cleared by Bus RESET or setting the STOP bit.
8	R	IDON - Initialization Done 0 = Initialization procedure not completed 1 = LANCE has completed the initialization procedure started by setting the INIT bit. When IDON is set, the LANCE has read the initialization block from memory and stored the new parameters.
		If INEA = 1, an interrupt is generated when IDON is set.
		IDON is READ/CLEAR ONLY
	W	0 = No effect 1 = Clears the bit
		IDON is cleared by Bus RESET or setting the STOP bit.
7	R	INTR - Interrupt Flag 0 = BABL, MISS, MERR, RINT, TINT, and IDON are all equal 0 1 = One or more of the following interrupt causing conditions has occurred: BABL, MISS, MERR, RINT, TINT, IDON. If INEA =1 and INTR =1 the INTR L I/O pin will be low.
		INTR is cleared by Bus RESET or setting the STOP bit. INTR is also cleared by clearing the conditions that caused the interrupt.

Bit	R/W	Description (CSR0 - cont.)
6	R/W	INEA - Interrupt Enable 0 = The INTR L I/O pin will be high, regardless of the state of INTR. 1 = If INTR =1, the INTR L I/O pin will be low. INEA allows the INTR L I/O pin to be driven low when the Interrupt Flag is set. If INEA =0 INEA is cleared by Bus RESET or setting the STOP bit.
5	R	RXON - Receiver ON 0 = Receiver disabled 1 = Receiver enabled RXON = 0 when IDON = 1, if DRX = 1 in the MODE register or a memory error (MERR =1) has occurred. RXON = 1 when STRT is set in CSR0, if DRX 0 in the MODE field of the initialization block is IDON = 1. RXON is cleared by Bus RESET or setting the STOP bit.
4	R	TXON - Transmitter On 0 = Transmitter enabled 1 = Transmitter disabled When TXON = 0, IDON = 1, and DTX = 1 in the MODE register, an Underflow or Retry error has occurred during transmission or a memory error (MERR) has occurred. TXON = 1 when the INIT bit = 1, and when STRT =1 if DTX = 0 in the MODE register in the initialization block. TXON is cleared by Bus RESET or setting the STOP bit.
3	W R	TDMD - Transmit Demand 0 = The packet is sent subject to the 1.6-millisecond polling interval delay. 1 = Packet is transmitted without the typical delay of the polling interval, 1.6 milliseconds. W 0 = No effect 1 = Clears this bit TDMD, when set, causes the LANCE to access the Transmit Descriptor Ring without waiting for the poll time interval to elapse. TDMD need not be set to transmit a packet, it merely hastens the LANCE response to a Transmit Descriptor Ring entry insertion by the software. TDMD is cleared by Bus RESET or setting the STOP bit. TDMD is cleared by the LANCE after the Transmit Descriptor Ring is accessed.

Bit	R/W	Description (CSR0 - cont.)
2	R/W	STOP 0 = No effect 1 = LANCE disabled from all external activity and internal logic cleared STOP set is the equivalent of asserting Bus RESET L. The LANCE remains inactive and STOP remains set until the STRT or INIT bit is set. If STRT, INIT, and STOP are all set together, STOP will override the other bits and only STOP will be set. Stop will terminate all LANCE activities asynchronously. STOP is cleared by setting INIT or STRT.
1	R/W	STRT - Start STRT enables the LANCE to send and receive packets, perform direct memory access and do buffer management. Setting STRT clears the STOP bit. If STRT and INIT are set together, the INIT function will be executed first. STRT is READ/WRITE WITH ONE ONLY. Writing a 0 into this bit has no effect. STRT is cleared by Bus RESET or setting the STOP bit.
0	R/W	INIT - Initialize 0 = No effect 1 = Starts LANCE initialization and reads initialization block. Setting INIT clears the STOP bit. If STRT and INIT are set together, the INIT function will be executed first. INIT is cleared by Bus RESET or setting the STOP bit.

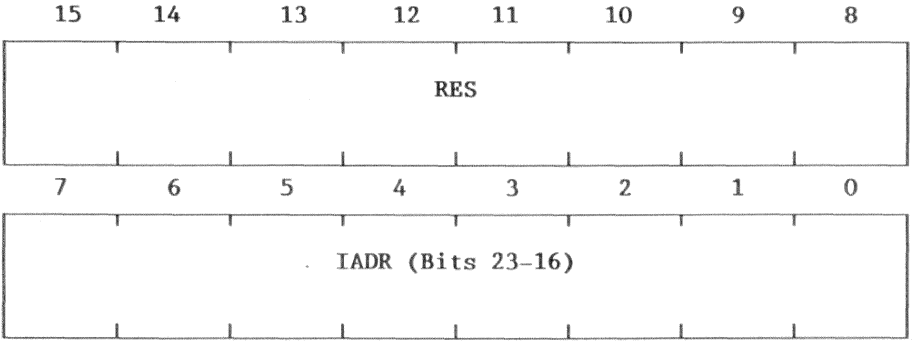
Control And Status Register 1 (RAP = 1)



Bit	Description
15-1	IADR The low-order 16 bits of the address of the first word (lowest address) in the initialization block.
0	Always 0

Accessible only when the STOP bit of CSR0 equals 1. If STOP = 0, an access returns READY, a Read operation returns undefined data and a Write operation is ignored. CSR1 is unaffected by Bus RESET L.

Control And Status Register 2 (RAP = 2)

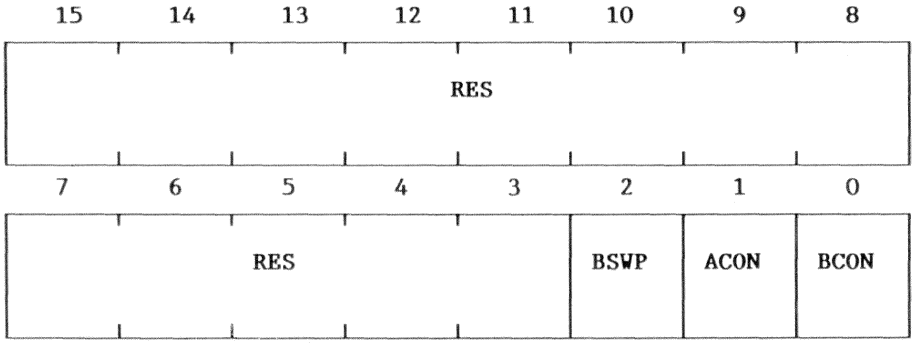


Bit	Description
15-8	RES Reserved.
7-0	IADR (Bits 23-16) The high order 8 bits of the address of the first word of the initialization block.

Accessible only when STOP equals 1 (CSR0). If STOP = 0, an access returns READY, a Read operation returns undefined data and a Write operation is ignored.

CSR2 is unaffected by Bus RESET L.

Control And Status Register 3 (RAP = 3)



Bit	R/W	Description
15-3		RES - Reserved (Always 0)
2	R/W	BSWP - Byte Swap 0 = The LANCE does not swap high and low bytes 1 = The LANCE swaps the high and low bytes on DMA data transfers between the SILO and bus memory. BSWP allows the LANCE to operate in systems that consider bits 15-8 to be the least significant byte and bits 7-0 to be the most significant byte. Only data from Silo transfers is swapped, the initialization block data and the Ring Descriptor entries are NOT swapped. BSWP is cleared by Bus RESET or setting the STOP bit in CSR0.
1	R/W	ACON - ALE Control
	R/W	0 = ALE is asserted high. 1 = ALE is asserted low. ACON defines the assertive state of ALE when the LANCE is a Bus Master. ACON is cleared by Bus RESET or setting the STOP bit in CSR0.

Bit	R/W	Description (CSR3 - cont.)
0	R/W	BCON - Byte Control

BCON	I/O Pin 16	I/O Pin 15	I/O Pin 17
0	BM 1 L	BM 0 L	HOLD L
1	BUSAKO L	BYTE H	BUSRQ L

BCON redefines the Byte Mask and Hold I/O pins.

BCON is cleared by Bus RESET or setting the STOP bit in CSR0.

Table 13-2 lists the value required for each function for CSR3.

Table 13-2 LANCE CSR3 Required Values For The VAXmate Workstation

Bit	Function	Value Required
15-3	Undefined	should be zero
2	BSWP (Byte Swap)	0
1	ACON (ALE Control)	1
0	BCON (Byte Mask Control)	0

CSR3, which allows redefinition of the bus master interface, contains three bits. They are used to customize certain hardware interface signals provided by the LANCE when it is in bus master mode. The programming of these bits is hardware implementation dependent. The VAXmate workstation values are in Table 13-2.

Accessible only when the STOP bit of CSR0 equals 1. If STOP = 0, an access returns READY, a Read operation returns undefined data and a Write operation is ignored.

Bus RESET L or setting the STOP bit in CSR0, clears CSR3.

NI CSR (0C68H)

7	6	5	4	3	2	1	0
LED SET	RES	RES	LED CLEAR	INT CLEAR	RES	RES	INT SET

Bit	R/W	Description
7		LED SET - Diagnostic LED Indicator Set
	W	0 = No effect
		1 = LED on
6		RES - Reserved (undefined)
5		RES - Reserved (undefined)
4		LED CLEAR - Diagnostic LED Indicator Clear
	W	0 = No effect
		1 = LED off (Power-on default)
3		INT CLEAR - Network Interface Interrupt Enable Clear
	W	0 = No effect
		1 = Disable interrupts (Power-on default)
2		RES - Reserved (undefined)
1		RES - Reserved (undefined)
0		INT SET - Network Interface Interrupt Enable Set
	W	0 = No effect
		1 = Enable interrupts

The NI CSR supports individual bit-set and bit-clear capability in a write-only register. Writing a 1 to a bit invokes its particular set or clear operation on the designated function. Writing a 0 to a bit does not invoke the particular set or clear operation for the designated function. Writing 0 to both set and clear bits for a designated function leaves the state of that function unchanged. Because writing 1 to both the set and clear bits of a designated function will toggle the state of that function, writing 1 to both the set and clear bits is not recommended.

Initialization Block

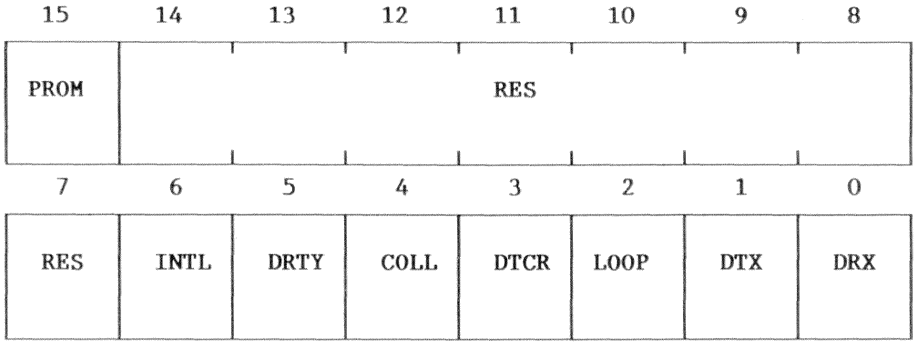
During initialization, the LANCE reads operating parameters from the initialization block.

When the INIT bit in CSR0 is set, the LANCE begins reading the initialization block. To ensure proper parameter initialization and LANCE operation, the controlling program should set the INIT bit and then set the STRT bit. After the LANCE reads the initialization block, the LANCE sets IDON in CSR0 and if INEA equals 1, the LANCE also generates an interrupt.

Higher Addresses	TLEN-TDRA (23-16)	IADR +22
	TDRA (15-00)	IADR +20
	RLEN-RDRA (23-16)	IADR +18
	RDRA (15-00)	IADR +16
	LADRF (63-48)	IADR +14
	LADRF (47-32)	IADR +12
	LADRF (31-16)	IADR +10
	LADRF (15-00)	IADR +08
	PADR (47-32)	IADR +06
	PADR (31-16)	IADR +04
	PADR (15-00)	IADR +02
Base Address of Block	MODE (15-00)	IADR +00

The base address, BASE ADDR, is formed from CSR2 bits 7-0 and CSR1 bits 15-1.

Mode Field
(Initialization Block, Offset 00H)



Bit	Description
15	PROM - Promiscuous 0 = Incoming packets matching physical or logical address filter are accepted. 1 = All incoming packets are accepted.
14-7	RES - Reserved
6	INTL - Internal Loopback INTL is used with the LOOP (bit 2) to determine where the loopback is to be done. Internal loopback allows the LANCE to receive its own transmitted packet. The maximum transmit packet size allowed during loopback is 32 data bytes. The LANCE automatically adds 4 bytes of CRC to the packet if DTCRC=0 and therefore the receive packet could be 36 bytes. INTL is only valid if LOOP = 1, otherwise it is ignored.

INTL	Loop	LOOPBACK
X	0	No loopback, normal operation
0	1	external
1	1	internal

Bit	Description (Mode Field - cont.)
5	DRTY - Disable Retry 0 = LANCE attempts 15 retransmissions of a packet after the first collision before reporting a Retry error. 1 = LANCE attempts only one transmission of a packet. If there is a collision on the first transmission attempt, a Retry Error (RTRY) is reported in Transmit Message Descriptor 3 (TMD3).
4	COLL - Force Collision 0 = Collision not forced 1 = Collision forced during subsequent transmission attempt COLL allows the collision logic to be tested. For COLL to be valid, the lance must be in internal loopback mode. When COLL = 1, 16 transmissions are attempted, and a retry error is reported in TMD3.
3	DTCR - Disable Transmit CRC 0 = The transmitter generates and appends a CRC to the transmitted packet. 1 = The CRC logic is allocated to the receiver and no CRC is generated and sent with the transmitted packet. During loopback, DTCR = 0 generates a CRC on the transmitted packet. CRC logic, shared by the receiver and the transmitter, cannot generate and check CRC at the same time. Therefore, the receiver does the CRC check. The generated CRC and data are written into memory and can be checked by the software. If DTCR = 1 during loopback, the software must append a CRC value to the transmit data in the transmit buffer. The receiver checks the CRC on the received data and reports any errors.

Bit	Description (Mode Field - cont.)
2	LOOP - Loopback 0 = LANCE operates in normal mode 1 = LANCE operates in full duplex mode for test purposes LOOP allows the LANCE to operate in full duplex loopback mode for test purposes. The maximum transmit packet size is limited to 32 data bytes. The received packet may be 36 bytes because the LANCE automatically adds 4 CRC bytes if DTCRC=0. During loopback, the runt packet filter is disabled because the maximum packet is forced to be smaller than the minimum size Ethernet packet of 64 Bytes. LOOP = 1 allows simultaneous transmission and reception for a message constrained to fit within the SILO. The LANCE waits until the entire message is in the SILO before serial transmission begins. The incoming data stream fills the SILO from behind as it is being emptied. Moving the received message out of the SILO to memory does not begin until reception has ceased. In loopback mode, transmit data chaining is not possible. Receive data chaining is allowed, regardless of the receive buffer length. In normal operation, not loopback, the receive buffers must be at least 64 bytes long to allow time for buffer lookahead.
1	DTX - Disable the Transmitter 0 = LANCE can access the Transmit Descriptor Ring 1 = LANCE does not access the Transmit Descriptor Ring, therefore no transmissions are attempted. DTX = 1 clears the TXON bit in CSR0 when initialization is complete.
0	DRX - Disable the Receiver 0 = Receives incoming packets 1 = LANCE rejects incoming packets and does not access the Receive Descriptor Ring. DRX = 1 clears the RXON bit in CSR0 when initialization is complete.

The Mode field allows alteration of the LANCE operating parameters. In normal operation, the Mode Register clear.

Physical Address Field (Initialization Block, Offset 02H)

PADR, the Network Physical Address, is the unique 48-bit physical address assigned to the LANCE. PADR(0) must be zero.

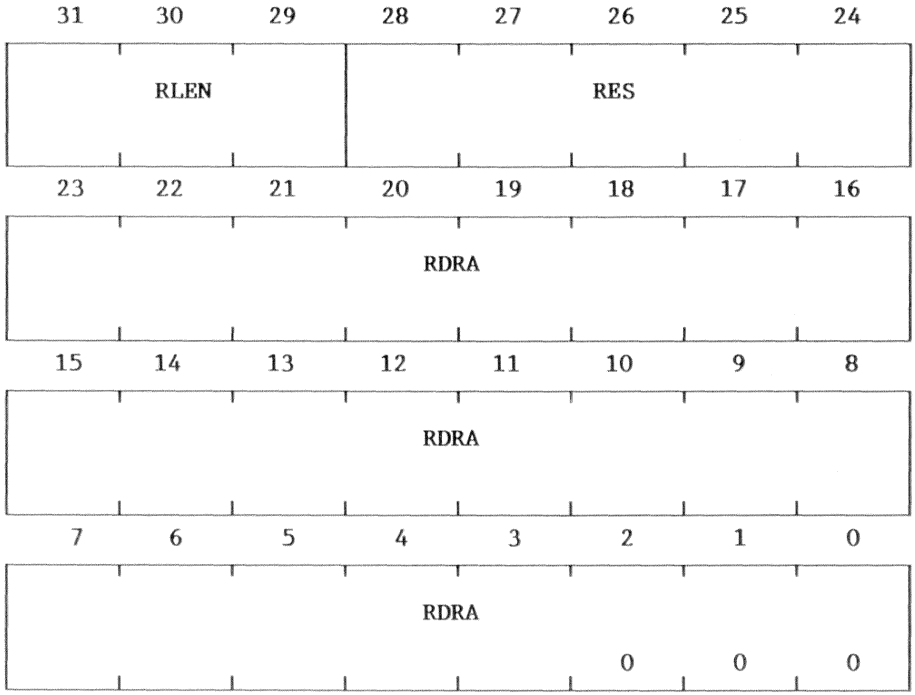
Logical Address Filter Field (Initialization Block, Offset 08H)

LADRF, the Logical Address Filter Field, is a 64-bit mask used by LANCE to accept the logical addresses.

The Logical Address Filter is a 64-bit mask that accepts incoming Logical Addresses sent through the CRC circuit. After all 48 bits of the address have travelled through the CRC circuit, the high-order 6 bits of the resultant CRC are strobed into a register. This register selects one of the 64-bit positions in the Logical Address Filter. If the selected filter bit equals 1, the address is accepted and the packet is put in memory. The first bit of the incoming address must be 1 for a logical address. If the first bit is 0, it is a physical address and is compared against the physical address that was loaded through the Initialization Block.

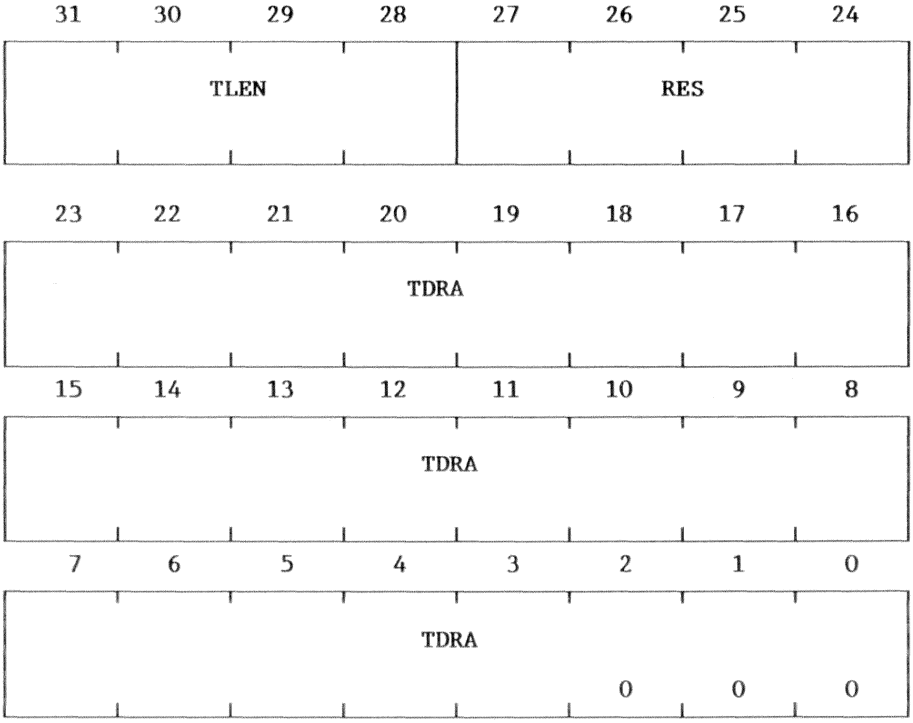
The Broadcast address, which is all ones, does not go through the Logical Address Filter and is always enabled. If the Logical Address Filter is loaded with all zeros, all incoming logical addresses except Broadcast are rejected.

Receive Descriptor Ring Pointer Field (Initialization Block, Offset 10H)



Bit	Description (Receive Descriptor Ring Pointer Field)
31-29	RLEN - RECEIVE RING LENGTH The number of entries in the Receive Ring expressed as a power of two. RLEN Number of Entries 0 1 1 2 2 4 3 8 4 16 5 32 6 64 7 128
28-24	RES (Reserved)
23-0	RDRA - Receive Descriptor Ring Address RDRA is the base address (lowest address) of the receive descriptor ring. Because the receive rings must be aligned on quadword boundaries, bits 2-0 must be zeros.

Transmit Descriptor Ring Pointer Field (Initialization Block, Offset 14H)



Bit	Description (Transmit Descriptor Ring Pointer Field)
31-29	TLEN TLEN is the number of entries in the Transmit Ring expressed as a power of two. TLEN / Number of Entries 0 / 1 2 / 4 3 / 8 4 / 16 5 / 32 6 / 64 7 / 128 8 / 256
28-24	RES - Reserved
23-0	TDRA - Transmit Descriptor Ring Address TDRA is the base address (lowest address) of the Transmit Descriptor Ring. Because the transmit rings must be aligned on quadword boundaries, these bits 2-0 must be zeros.

Buffer Management

Buffer descriptors, organized as ring structures in memory, are used for buffer management. The buffer descriptors, also called message descriptors, are four words long and point to a buffer. Two ring structures are allocated for the LANCE: a ring of receive message descriptors (RMDs) and a ring of transmit message descriptors (TMDs). To transmit or receive packets, the LANCE polls each ring structure. The LANCE also enters status information in the descriptor entry. The LANCE is limited to looking only one descriptor entry ahead of the one with which the LANCE is currently working.

The location of the descriptor rings and their length are found in the initialization block, which the LANCE accesses during the initialization procedure. Writing a 1 into the STRT bit of CSR0 causes the LANCE to start accessing the descriptor rings and enables the sending and receiving of data packets.

The LANCE communicates with the data link layer program through the ring structures in memory. Each entry in the ring is owned either by the LANCE or the data link layer program. The ownership bit (OWN) in the message descriptor entry determines who owns the entry. Therefore, ownership of a descriptor entry is mutual exclusive. To gain ownership of a descriptor entry, the communications partner (LANCE or data link layer program) must wait until the owner gives ownership to the communications partner. Between the time a communications partner relinquishes ownership of a descriptor entry and the time that communications partner regains ownership, it must not change any data in the descriptor entry.

Descriptor Rings in Memory

Figure 13-1 shows receive and transmit descriptor rings with 128 message descriptors each.

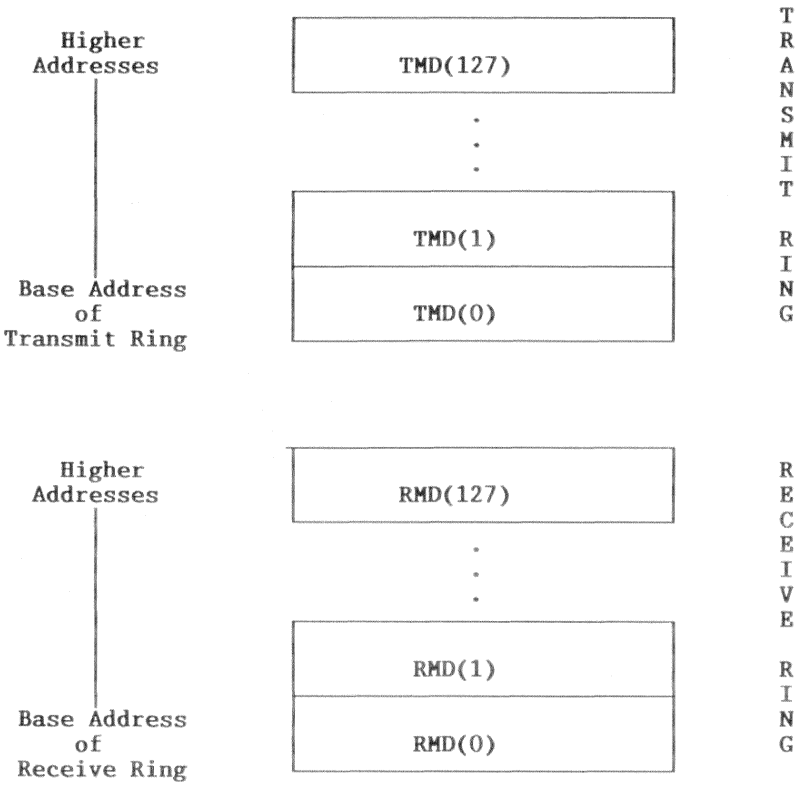
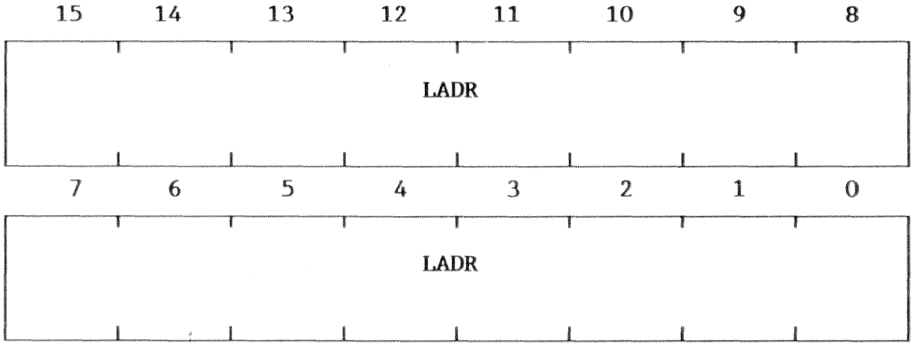


Figure 13-1 Descriptor Rings

Receive Descriptor Rings

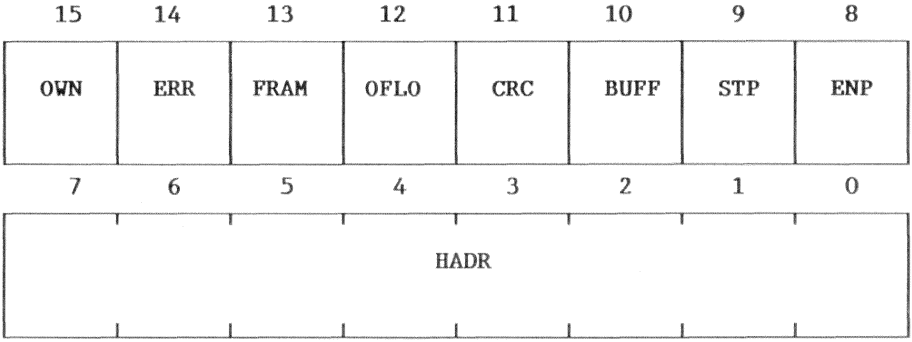
Each descriptor in a ring in memory is a 4 word entry. The format of the receive descriptor follows.

Receive Message Descriptor 0 (RMD0)



Bit	Description
15-0	LADR - Low Order Address The LADR of the buffer pointed to by this descriptor. LADR is written by the software and unchanged by the LANCE. This is the memory location to place the next received message.

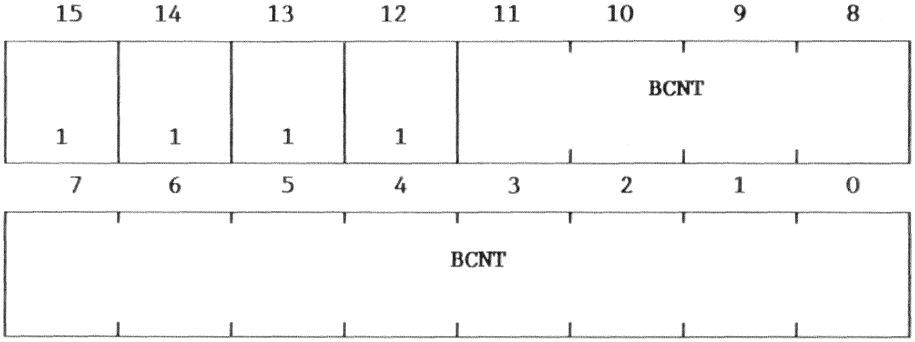
Receive Message Descriptor 1 (RMD1)



Bit	Description
15	OWN 0 = Descriptor entry owned by the data link layer software 1 = Descriptor entry owned by the LANCE The LANCE clears the OWN bit after filling the buffer pointed to by the descriptor entry. The software sets the OWN bit after emptying the buffer. Once the LANCE or software has relinquished ownership of a buffer, it may not change any field in the four words that comprise the descriptor entry.
14	ERR - Error 0 = The four bits, FRAM, OFLO, CRC or BUFF, are all equal to zero. 1 = One or more of the four bits, FRAM, OFLO, CRC or BUFF, is equal to one. ERR is set by the LANCE and cleared by the software.
13	FRAM - Framing Error 0 = No framing error 1 = The incoming packet contained a non integer multiple of eight bits and there was a CRC error. A CRC error on the incoming packet is a necessary condition for FRAM to equal 1 (even if there was a non integer multiple of eight bits in the packet). FRAM is set by the LANCE and cleared by the software.

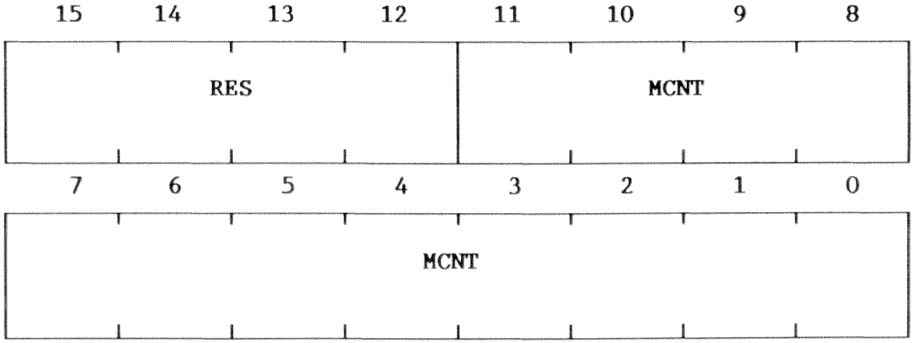
Bit	Description (RMD1 - cont.)
12	OFLO - Overflow Error 0 = No overflow error 1 = The receiver has lost all or part of the incoming packet because it cannot store the packet in a memory buffer before the internal SILO has overflowed. OFLO is set by the LANCE and cleared by the software.
11	CRC - Cyclic Redundancy Check 0 = No CRC error 1 = The receiver detected a CRC error on the incoming packet. CRC is set by the LANCE and cleared by the software.
10	BUFF - Buffer Error 0 = No buffer error 1 = The LANCE does not own the next buffer while data chaining a received packet. This condition could occur if the OWN bit of the next buffer was zero or the LANCE could not acquire the next status before silo overflow occurred. BUFF is set by the LANCE and cleared by the software.
9	STP - Start of Packet 0 = This is not the first buffer the LANCE uses for this packet. 1 = This is the first buffer LANCE uses for this packet. It is used for data chaining buffers. STP is set by the LANCE and cleared by the software.
8	ENP - End of Packet 0 = This is last buffer LANCE uses for this packet. 1 = This is the last buffer used by the LANCE for this packet. It is used for data chaining buffers. Both STP and ENP set indicate that the packet fit into one buffer and there was no data chaining. ENP is set by the LANCE and cleared by the software.
7-0	HADR - High Order 8 Address Bits The HADR of the buffer pointed to by this descriptor. This field is written by the software and unchanged by the LANCE.

Receive Message Descriptor 2 (RMD2)



Bit	Description
15-12	Must be ones. This field is written by the software and unchanged by the LANCE.
11-0	BCNT - Buffer Byte Count BCNT is the length of the buffer pointed to by this descriptor expressed as a two's complement number. This field is written by the software and unchanged by the LANCE. The minimum buffer size is 64 bytes.

Receive Message Descriptor 3 (RMD3)

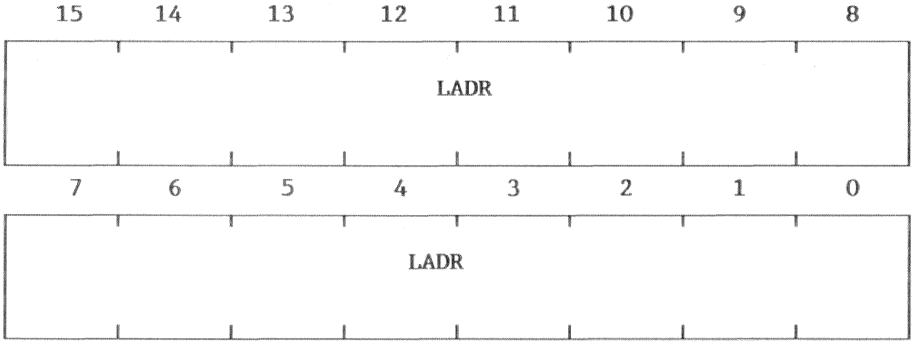


Bit	Description
15-12	RES Reserved and read as zeros.
11-0	MCNT - Message Byte Count MCNT is the length in bytes of the received message. MCNT is valid only when ERR is clear and ENP is set. MCNT is written by the LANCE and cleared by the software. When data chaining, RMD3 is only loaded by the LANCE when the status of last buffer is updated. Only the status word is updated for the other buffers in the data chain.

Transmit Descriptor Ring

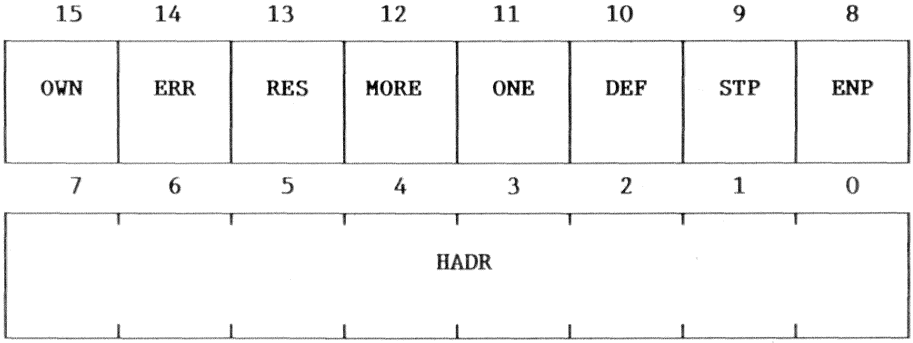
Each descriptor in a ring in memory is a 4 word entry. The format of the transmit descriptor follows.

Transmit Message Descriptor 0 (TMD0)



Bit	Description
15-0	LADR - Low Order 16 Address Bits The LADR of the buffer pointed to by this descriptor. LADR is written by the software and unchanged by the LANCE.

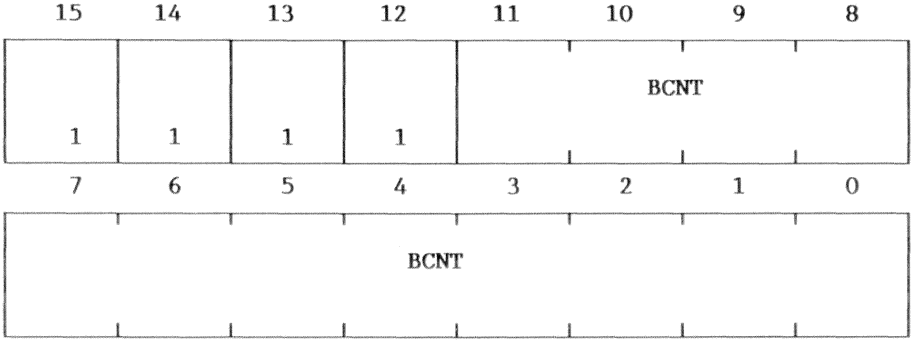
Transmit Message Descriptor 1 (TMD1)



Bit	Description
15	OWN 0 = Owned by the LANCE 1 = Owned by the data link layer software OWN indicates whether the descriptor entry is owned by the software (OWN = 0) or by the LANCE (OWN = 1). The software sets the OWN bit after filling the buffer pointed to by this descriptor. The LANCE clears the OWN bit after transmitting the contents of the buffer. Neither the software nor the LANCE may alter a descriptor entry after relinquishing ownership.
14	ERR - Error Summary 0 = All of LCOL, LCAR, UFLO, and RTRY equal 0. 1 = One or more of LCOL, LCAR, UFLO, or RTRY equal 1. ERR is set by the LANCE and cleared by the software.
13	RES- Reserved (Always 0)
12	MORE 0 = One retry or less was needed to transmit a packet. 1 = More than one retry was needed to transmit a packet. MORE is set by the LANCE and cleared by the software.

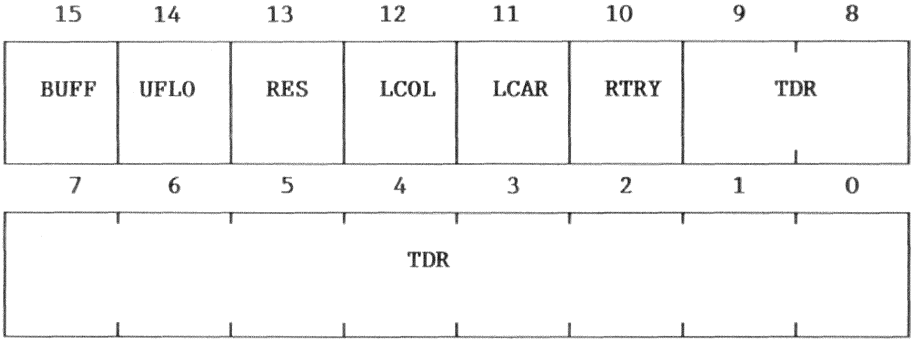
Bit	Description (TMD1 - cont.)
11	ONE 0 = The number of retries need to transmit a packet is not equal 1. 1 = Exactly one retry was needed to transmit a packet. ONE is set by the LANCE and cleared by the software.
10	DEF - Deferred 0 = The channel is ready for LANCE to transmit a packet. 1 = The channel is busy when the LANCE is ready to transmit a packet. DEF set by the LANCE and cleared by the software.
9	STP - Start of Packet 0 = Not the first buffer LANCE uses for this packet. 1 = The first buffer to be used by the LANCE for this packet. It is used for data chaining buffers. STP is set by the software and unchanged by the LANCE.
8	ENP - End of Packet 0 = Not the last buffer LANCE uses for this packet 1 = The last buffer to be used by the LANCE for this packet. It is used for data chaining buffers. If both STP and ENP are set, the packet fit into one buffer and there was no data chaining. ENP is set by the software and unchanged by the LANCE.
7-0	HADR - High-Order 8 Address Bits HADR is the high-order 8 address bits of the buffer pointed to by this descriptor. HADR is written by the software and unchanged by the LANCE.

Transmit Message Descriptor 2 (TMD2)



Bit	Description
15-12	Always 1. This field is set by the software and unchanged by the LANCE.
11-0	BCNT - Buffer Byte Count BCNT is the number of bytes LANCE transmits of the buffer to which TMD2 points. BCNT is expressed in 2's complement. BCNT is written by the software and not affected by the LANCE. The minimum size of the buffer is 64 bytes.

Message Descriptor 3 (TMD3)



Bit	Description
15	BUFF - Buffer Error 0 = No buffer error 1 = LANCE, during transmission, does not find the ENP flag in the current buffer and does not own the next buffer. BUFF = 1 also if the LANCE owns the next buffer but can not read the next buffer's status parameters before the silo underflowed. BUFF is set by the LANCE and cleared by the software. If a Buffer Error occurs, an Underflow Error also occurs because the transmitter continues to read data from the silo until empty.
14	UFLO - Underflow Error 0 = No underflow error 1 = The transmitter has truncated a message due to lack of data from memory. UFLO indicates that the Silo has emptied before the end of the packet was reached. UFLO is set by the LANCE and cleared by the software.
13	RES - Reserved (Always 0)
12	LCOL - Late Collision 0 = No collision 1 = A collision occurred after the slot time of the channel has elapsed. The LANCE does not retry on late collisions. LCOL is set by the LANCE and cleared by the software.

Bit	Description (TMD3 - cont.)
11	LCAR - Loss of Carrier 0 = RENA does not go false during transmission 1 = The carrier presence (RENA) input to the LANCE goes false during a transmission initiated by the LANCE. The LANCE does not retry upon Loss of Carrier. LCAR is set by the LANCE and cleared by the software.
10	RTRY - Retry Error 0 = No retry error 1 = The transmitter has failed in 16 attempts to transmit a message due to repeated collisions on the medium. If DRTY = 1 in the MODE register, RTRY sets after 1 failed transmission attempt. RTRY is set by the LANCE and cleared by the software.
9-0	TDR - Time Domain Reflectometry The TDR shows the state of an internal LANCE counter that counts from the start of a transmission to the occurrence of a collision. This value is useful in determining the approximate distance to a cable fault. The TDR value is written by the LANCE and is valid only if RTRY is set.

TMD3 is valid only if the LANCE has already set the ERR bit of TMD1.

Network Interface External Interconnect

The VAXmate network hardware connects to the network coaxial cable through a BNC T-connector. The VAXmate network hardware complies with IEEE 802.3 10BASE2 specifications. The cable shield is isolated from the VAXmate logic and chassis ground, and must be externally grounded by the interconnect equipment. The VAXmate workstation does not contain a 50-ohm line-termination. The 50-ohm line terminator must be supplied externally, as required by the connection topology.

Network Interface System Bus Interconnect

The hardware interface uses the 16-bit and 8-bit system bus for data transfers. It operates in both bus master and bus slave modes. In bus slave mode, the registers respond to I/O cycles. The LANCE registers require 16-bit I/O and the NI CSR register requires an 8-bit I/O. All of the network interface registers are located in proprietary I/O address space of the VAXmate workstation. Accessing the LANCE registers can cause extra I/O wait states.

In the bus master mode, the LANCE initiates 16-bit memory cycles to transfer data, commands and parameters to and from main system memory. The VAXmate workstation supports both 16-bit and 8-bit memory cycles, but the network interface supports only 16-bit memory cycles. Although the VAXmate uses the READY signal to support asynchronous memory cycles, the network interface supports only synchronous memory cycles. The network interface performs DMA transfers at the rate of 600 ns per cycle.

The network hardware works with standard system memory resident on the VAXmate CPU module, and with the VAXmate memory option located in the workstation main box. Accesses to 8-bit memory in the VAXmate expansion box are not supported by the network hardware. If the memory complies with the network hardware timing requirements, third-party 16-bit memory in the expansion box may work correctly.

The interface does not support DMA to the VAXmate video display memory. The interface uses the full 24-bit physical memory address space and operates with physical addresses only.

The network hardware uses interrupt line IRQ10, and a special LAN DMA request line, provided by the CPU Module. The special LAN DMA request line has higher priority than all DRQs lines on the system bus.

Index

E

F

J

K

T

W

Technical Reference Manual
Volume 1
AA-JP76A-TH

Reader's Comments

Your comments on this manual will help improve our product quality and usefulness.

Please indicate the type of reader you most closely represent.

☐ First-time user ☐ Programmer ☐ Experienced user

☐ Application user ☐ Other (please specify)_____

How would you rate this manual for:

	Excellent	Good	Fair	Poor
Completeness of Information	☐	☐	☐	☐
Accuracy of Information	☐	☐	☐	☐
Easy to Read/Use	☐	☐	☐	☐
Usefulness of Examples	☐	☐	☐	☐
Number of Examples	☐	☐	☐	☐
Illustrations	☐	☐	☐	☐
Table of Contents	☐	☐	☐	☐
Index	☐	☐	☐	☐
Format	☐	☐	☐	☐
Binding Style	☐	☐	☐	☐
Print Quality	☐	☐	☐	☐

Did you find any errors in this manual? Please specify by page and paragraph.

Incorrect information: _____

_____

Information left out: _____

_____

Hard to understand: _____

_____

What suggestions do you have for improving this manual? Attach a second sheet if necessary.

_____

_____

_____

Name_____ Title_____

Company_____ Dept._____

Street_____ City_____

State/Country_____ Postal/Zip Code_____

Telephone_____ Date_____

Do Not Tear - Fold Here and Tape

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1–2 L12
MARLBOROUGH, MA 01752

Do Not Tear - Fold Here

Cut Along Dotted Line

Technical Reference Manual
Volume 1
AA-JP76A-TH

Reader's Comments

Your comments on this manual will help improve our product quality and usefulness.

Please indicate the type of reader you most closely represent.

☐ First-time user ☐ Programmer ☐ Experienced user

☐ Application user ☐ Other (please specify)_____

How would you rate this manual for:

	Excellent	Good	Fair	Poor
Completeness of Information	☐	☐	☐	☐
Accuracy of Information	☐	☐	☐	☐
Easy to Read/Use	☐	☐	☐	☐
Usefulness of Examples	☐	☐	☐	☐
Number of Examples	☐	☐	☐	☐
Illustrations	☐	☐	☐	☐
Table of Contents	☐	☐	☐	☐
Index	☐	☐	☐	☐
Format	☐	☐	☐	☐
Binding Style	☐	☐	☐	☐
Print Quality	☐	☐	☐	☐

Did you find any errors in this manual? Please specify by page and paragraph.

Incorrect information: _____

_____

Information left out: _____

_____

Hard to understand: _____

_____

What suggestions do you have for improving this manual? Attach a second sheet if necessary.

_____

_____

_____

Name_____ Title_____

Company_____ Dept._____

Street_____ City_____

State/Country_____ Postal/Zip Code_____

Telephone_____ Date_____

Do Not Tear - Fold Here and Tape

digital

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1-2 L12
MARLBOROUGH, MA 01752

Do Not Tear - Fold Here

Cut Along Dotted Line